Teaching Web Services with Water

Matthew D. Kendall   Edward F. Gehringer

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206

matt_kendall@iname.com, efg@ncsu.edu

ABSTRACT

Web services have gained popularity and importance to the point that all software professionals should know something about them. However, it can be challenging to fit Web services into a crowded curriculum. Common approaches require teaching a host of standards and APIs that all but obscure the simplicity of the concepts. The object-oriented Water™ language offers a way around these difficulties. Originally designed for rapidly prototyping XML-based Web services, it provides a very concise encoding of Web-services functionality. Students can learn to write real Web-services programs in as little as two or three weeks. Moreover, Water facilitates the use of several patterns that are important to understanding object-oriented design but lacking—or not implemented well—in common o-o languages such as Java and C++. Among these are delegation and multiple inheritance.

Categories and Subject Descriptors

D.1.5 [Software] Programming Techniques – Object-Oriented Progrmamming, H.3.5 [**Information Storage & Retrieval]** Online Information Services – Web-Based Services, K.3 [**Computers & Education**]: Computer & Information Science Education – Computer Science Education

General Terms

Design, Languages

Keywords

Water, Web services, XML

1. INTRODUCTION

Web services are a powerful new computing paradigm that provides a standardized way for applications to exchange information over a network. Since their introduction in 2001, Web services have rapidly gained popularity in the software development industry [Ric 04]. Due to this combination of utility and popularity, Web services should be a part of all computer science curricula. Unfortunately, traditional methods of implementing Web services require learning a host of standards and APIs, for example, the Simple Object Access Protocol (SOAP), the Web Services Description Language (WSDL), Java servlets, the .NET platform. The amount of infrastructure associated with Web services obscures the simplicity of the concept. We propose an alternative: the Water programming language [Wate 05]. Water is a simple, self-contained, object-oriented programming language specifically designed for rapidly developing Web services. By using Water to teach Web services, we significantly reduce the time spent studying infrastructure and allow students immediately to experience the Web-services model.

While the Web-services model is not inherently object oriented, it is an example of a Service-Oriented Architecture (SOA). An SOA is an architectural design pattern that partitions a system into a set of services. Each service performs a set of related functions that are revealed to other services through a well-defined interface that describes its inputs and outputs. Work is accomplished by passing messages between services using these interfaces. Each service is dependent only on the interfaces of the services that it utilizes. This means that the internal implementation of a service can change without affecting the system as a whole. This property, known as loose coupling, is a common goal of object-oriented design and is the reason for information hiding [IBM 05]. SOA is unique because its services usually consist of applications, rather than objects within a system. Communication between applications makes use of message passing, interfaces, and information hiding, in the context of distributed systems. Teaching Web services in this way helps drive home the point that object-oriented concepts have application far beyond the realm of programming languages.

Water allows students to avoid the steep learning curve associated with Web services and focus on the theory of an SOA. This is desirable for several reasons. First, it is easier to find space in a curriculum for three weeks of Water instruction than for a semester of Web-services infrastructure. Second, the standards that Web services employ are not yet stable [Ort 04, See 04]. Finally, students are able to see the advantages of Web services quickly through examples and programming exercises.

In order to understand the simplicity of implementing Web services with Water, consider the following Water program. Executing this program creates a Web server at http://localhost that responds to any HTTP request with the current date:

```
<server date.<current/> />
```

This program is much simpler than equivalent implementations on platforms such as Java or .NET, and it is completely self-contained (it can be run without any other Web server present).

Water possesses several other interesting features in addition to its treatment of Web services. Since the syntax of the language is XML based, Water incorporates XML parsing as a language feature. Students who are not familiar with XML will be quite familiar with its structure after working with Water. In addition, Water supports at least two interesting object-oriented patterns, delega-

tion and multiple inheritance, that are rarely seen in popular object-oriented languages such as Smalltalk and Java.

2. Web services

Web services are a collection of standards that enable platform- and language-independent machine-to-machine communication. This powerful implementation of distributed computing means that two or more applications can exchange information autonomously, regardless of the hardware, operating systems, or programming languages involved. The key to the operation of Web services is the standards that they employ to create an SOA.

The first of these standards is XML, the eXtensible Markup Language. XML provides a format for the messages passed between services and, since XML is parse-able by any language that can interpret plain text, it provides Web services with superior interoperability. XML messages are passed to service interfaces over traditional Internet protocols, e.g. HTTP, FTP, and SMTP. The interfaces of each service are described in another XML language, the Web Services Description Language (WSDL). Each Web service maintains a WSDL definition that describes its inputs and outputs. Finally, WSDL definitions can be collected in Universal Description, Discovery and Integration (UDDI) catalogs. Since UDDI is also an XML language, these catalogs can be easily updated and queried by machines. A simple diagram of an exchange between two Web services is presented in Figure 1.

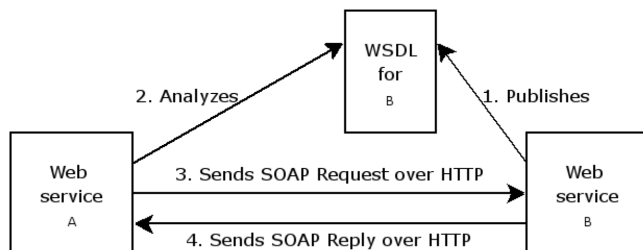

Figure 1: Typical Web service operation. (1) Web service B publishes a WSDL description of its interface. (2) Web service A analyzes it and (3) generates a request using the specification. (4) Web service B sends a response formatted using the specification.

The idea of autonomously passing messages between two entities over a network is not unique to Web services. Technologies such as Remote Procedure Calls (RPC) and the Common Object Request Broker Architecture (CORBA) accomplished a similar goal, often with better performance. Unfortunately, these techniques require using a uniform platform, language, or both. Web services not only obviate this need, they use accepted standards such as XML and proven Internet protocols to accomplish the task.

Although the standards supporting Web services have not fully solidified, the concept of Web services is well established. More than 50,000 developers are registered with Amazon Web Services, and eBay's database receives more than 30 million Web-services requests per day [Ric 04]. In addition, companies such as IBM and Microsoft have shown increasing interest in the technology. For example, Microsoft restructured the .NET platform to provide explicit support for Web services.

This industry support, combined with the fluidity of Web services standards, is another compelling reason to teach the theory of Web services while hiding the details. Students should know enough about Web services to understand Figure 1, but they do not need to know the intricacies of each supporting technology. Water serves this end by mapping the concepts of Web services to familiar object-oriented constructs.

3. How Web services are taught today

Many Computer Science departments have begun to teach courses in Web services. The most common approach seems to be to use .NET. Four of five instructors responding to a question posed on the SIGCSE-members reported that they used C# or .NET in their courses. Some courses, such as the Service-Oriented Computing [NCSU 05] and Web Services [NCSU 05b] courses at NCSU, use both Java or J2EE and .NET. Last year, a paper at this symposium [KG 04] compared the advantages of teaching Web services in J2EE and .NET, and concluded that, for a program with a Java-centric curriculum, the platform-independent J2EE was a natural choice. But .NET courses seem to predominate. Web services are taught at all levels, from CS1 [LJM 05] to advanced graduate [NCSU 05a], with the latter focusing on the broader issues of service-oriented architectures. Well over a thousand books on Web services are available, including many texts, such as Deitel et al. [DDDT 02] and the forthcoming Møller and Schwartzbach [MS 05].

4. Water

To facilitate the adoption of Water and Web services into computer science curricula, we have created a series of six 50-minute lectures. These lectures introduce the concept of Web services, the Water language, and the implementation of Web services with Water. These lectures and a companion set of programming exercises are available on-line at: http://people.engr.ncsu.edu/efg/water.

The best way to demonstrate the power of Water is through example. Therefore, in the next two sections we present examples of Water's unique Web services and object oriented features. In order to conserve space, we include only the code necessary to understand the examples. The complete programs are available online with our lecture series (and in Appendix B).

Perhaps the most characteristic feature of Water is its syntactic similarity to XML. Although this makes Water extremely adept at executing XML taken from other source, it tends to lead to very verbose code. Therefore, Water employs a extension of XML called ConciseXML, which introduces the following features to XML:

1. **Closing Tag Shorthand** – closing tags can be abbreviated as **</>**.
   XML: **<foo>bar</foo>**
   ConciseXML: **<foo>bar</>**
2. **Positional Arguments** – Names of keys can be omitted from attributes whenever the key of an attribute can be inferred. This elision reflects the fact that XML tags are used for both method invocation and object creation in Water. Specifying both the key and value for each parameter in a predefined method is needless verbosity.
   XML: **<foo attr1="value" attr2="value"/>**

```
<set flickr = <web
    <uri "http://www.flickr.com/services/rest/"
      query="method=flickr.photos.search&api_key=a04183d905bac769bddb9f5a78c4aa24"
    />
    contract=<defmethod text=required=string/>
  />
/>
```

Water's XML syntax can be daunting at first, so we have divided the code into functional units using borders. When a unit is referenced in the text, it is outlined using its corresponding border, e.g. <web/>.

Listing 1: Creating a Web service method for Flickr

ConciseXML: **<foo "value" "value"/>**

3. **Path notation** – Subelements of an element can be accessed via a dotted path notation.
   XML: **<foo><bar>text</bar></foo>**
   ConciseXML: to access "text": **foo.bar**

Water's object model is both simple and powerful. It follows four simple rules [Plus 02]:

1. Everything is an object.
2. An object has fields.
3. Each field has a key and a value.
4. The key and value can be any object.

Section 4.2 provides more information about the Water object model.

The code examples provided on the lectures page can be executed using the trial edition of the Steam IDE, which is available from www.waterlanguage.org.

4.1 Water and Web services

One popular type of Web service is an application that consumes data from one or more Web services and outputs a new view of the collected data. We present such an example here to demonstrate Water's ability to collect data from remote services and show how easily it serves data up over the Web.

Our example utilizes the Web services at Flickr (www.flickr.com), which is an on-line service for storing, organizing, and sharing digital photographs. Although Flickr's Web services are expansive, we will only be using a single function that searches the image database using keywords. The service is controlled by parameters in a URL:

http://www.flickr.com/services/rest/?**method**=flickr.photos.search&**api_key**=a04183d905bac769bddb9f5a78c4aa24&**text**=keywords

The above URL has three important parameters in the query: **method**, **api_key**, and **text**. The method is the function from Flickr's Web services that we want to invoke. The **api_key** is a unique key that allows Flickr to track usage of the service. The **text** parameter contains a space-delimited list of keywords.

Executing a query of this kind returns a complex XML document that contains no URLs for the photographs that match the keywords. The user must construct the URL from information contained in the response. The XML response is of the following form:

```
<rsp stat="ok">
  <photos page="1" pages="57" perpage="100"
total="5673">
  <photo id="6702150"
owner="49503205076@N01" secret="0e98f0349d"
server="8" title="Maxwell" ispublic="1" is-
friend="0" isfamily="0"/>
  ...more photo tags...
  </photos>
</rsp>
```

We will create a Water Web service that constructs the URL and simplifies this XML output to:

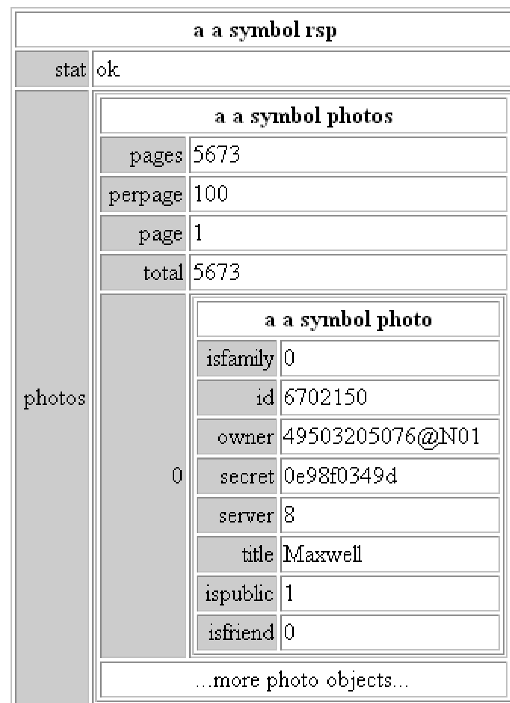

Figure 2: A Water XML Object

```
<vector>
  <simple_flickr
url="http://photos8.flickr.com/6702150_0e98f
0349d_m.jpg" title="Maxwell"/>
  ...more simple_flickr tags...
</vector>
```

We begin by using one of Water's most powerful features, which allows us to define a Web service as a method (Listing 1). Essentially, a Web service method (the **<web/>** object) is created and stored into the variable **flickr**. If we inspect the **<web/>** object, we can see that it is created with two fields. The first field contains a **<uri/>** object that represents Flickr's Web service. Notice that the **<uri/>** object does not include the **text** parameter. We tell Water about the Web service's **text** parameter with the second of **<web/>**'s parameters, the **contract** parameter.

The **contract** parameter is used by Water to express that the Flickr Web service has a single required parameter, called **text**, of type string. We can now invoke the Web service method:

```
<flickr "water"/>
```

The result is a Water object that represents Flickr's XML response to a search for the term "water". The object that Water returns by default has a structure that is identical to the response's XML structure. We can simplify the structure of the response object using Water's **<normalize/>** method (Appedix A).

```
<normalize <flickr text=keyword/>.<to_xml/>
/>
```

<normalize/> condenses the response object and returns the Water object shown in Figure 2. The process of creating an object from an XML spcification is much more complex in other languages, such as Java, which require XML binding files or navigation of a Document Object Model (DOM).

The next step towards a simple Flickr output is creating an object to hold the title and URL for each photo object.

```
<defclass simple_flickr title url/>
```

Now we have a **<simple_flickr/>** class to store each title and URL from the **<flickr/>** method. All that remains is to create a method that converts from Flickr's **<photo/>**s to our **<simple_flickr/>**s.

We define a method named **<flickr_reducer/>** that takes a single parameter: **keyword** (Listing 2). The first line of the method is familiar; we call the Flickr Web service using the provided **keyword** parameter and store the returned object in **result**. Then, **<for_each/>** photo in the photos field, we create a **<simple_flickr/>** object. The **value** variable is set to the current photo object in each iteration. We use the **title** of the **value** for **<simple_flickr/>**'s **title** field, but the **url** field is more complicated.

To build the **url**, we create a Water **<vector/>** of the known and variable parts of the **<photo/>**'s URL and then we concatenate all of the array values together using the **<join/>** method. The result is a single string that represents the URL.

```
<defmethod flickr_reducer keyword>
 <set result=<normalize
   <flickr text=keyword/>.<to_xml/>
 /> />
 result.photos.<for_each combiner=insert
                         include=vector_key>
   <simple_flickr value.title
     <vector
       "http://photos" value.server
       ".flickr.com/" value.id "_"
       value.secret "_m.jpg"
     />.<join/>
   />
 </>
</>
```

Listing 2: The <flickr_reducer/> method.

As each **<simple_flickr/>** object is created, it is inserted into a **<vector/>** using Water's **<for_each/>** construct. **<for_each/>** is Water's generalized looping statement; the **combiner=insert** parameter causes the simple Flickr object to be inserted into a vector. This vector is returned by the **<flickr_reducer/>** method because the **<for_each/>** loop is the last statement in the method. The **include=vector_key** parameter of the **<for_each/>** loop causes the loop to include fields with numeric keys and ignore fields in the **<photos/>** object with non-numeric keys, such as **per_page** and **total**. The result is that only fields containing photo objects are processed by the loop.

The advantage that Water affords to this mundane processing task is the ease with which we can iterate over the objects in an XML collection.

The final step is to allow the **<flickr_reducer/>** method to be accessed over a network. As we have seen, a single Water statement accomplishes the task:

```
<server root=thing port=8080/>
```

This statement creates an HTTP server that is accessible over port 8080. The **root** of the **<server/>** (http://localhost:8080) points to the **thing** class, which is the ancestor of all objects in Water and is similar to Java's Object class. By setting the **<server/>**'s **root** to **thing**, we open the entire Water environment to the Web. Any object or class in Water can now be created by accessing the correct URL.

The various URLs that access our service are shown in Figure 3. The first URL produces the expected XML output (see the above example).

Water also provides alternate formatting outputs. If we want to produce human-readable output, we can modify the URL to create HTML or SOAP output. Without altering our implementation, we are able to support a variety of clients.

URL	Type
http://localhost:8080/flickr_reducer.**xml**?keyword=water	XML
http://localhost:8080/flickr_reducer.**htm**?keyword=water	HTML
http://localhost:8080/flickr_reducer.**soap**?keyword=water	SOAP

Figure 3: URLs that access the Water service and their corresponding return types.

The entire program (see Appendix B) is only 23 lines of code. This illustrates that Water provides access to Web services at a very high level.

4.2 Water's Object-Oriented Patterns

Water supports several object-oriented patterns, including delegation and multiple inheritance. We cover these concepts in our lectures because they expose students to object-oriented systems beyond the strict, single-inheritance behavior-sharing models of Java and Smalltalk.

Delegation is a way of message passing that is common in prototype-based languages such as Self [US 87]. Prototype-based languages are designed according to the theory that humans learn by first mastering a specific example and then expanding their understanding to similar instances. In a prototype language, a user defines a concrete object, the prototype, to fill a specific role. As similar roles are discovered within the system, the prototype is copied and the copies are modified to suit their individual functions. [Lieb 86] This is in stark contrast to class-based languages, where all the roles must be understood *a priori* so that a set of classes and subclasses can be created.

Although Water is a class-based language, it also supports the concept of delegation. In a prototype language, when an object receives a message it determines whether or not it can answer the message. If it cannot, it *delegates* the message to its prototype. When the prototype handles the message it uses the context of the original object. Therefore, prototypes answer delegated messages on behalf of the original recipient. This process differs from the *consultation* message-passing scheme of class-based languages, where the object consults its class definition to determine whether it can handle the message. If it cannot, it consults an ancestor, who responds to the message as if it were the original recipient of the message. [DaP]

Semantically, the difference lies in the binding of the "**this**", or "**self**", variable, which is called **_subject** in Water. Under delegation, **_subject** is always bound to the original message recipient. In consultation, **this** is bound to the final message handler.

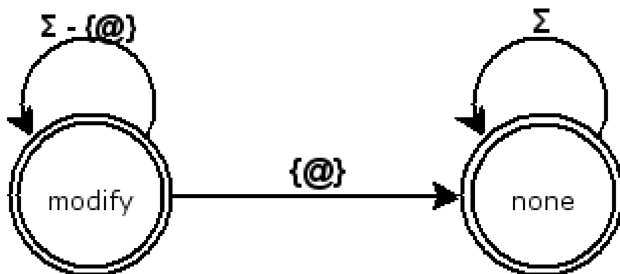

Figure 4: : Finite State Machine for <obfuscator/>. Σ is the set of all valid string characters.

Water object instances only have field values that are set at object creation or by calling **<set/>**. All other fields are accessed via delegation to the class definition. This trick keeps objects small and simple. Water also benefits from delegation because it allows dynamic runtime specialization [Rai 94]. Since the object consults its instance, and not its class, to determine whether or not it can handle a message, we can change the behavior of object instances at run time. This feature can be extremely useful when implementing state machines.

For example, assume we need to build a simple state machine that obfuscates an e-mail address in an obvious way so that it cannot be recognized by spam bots. We use the machine in Figure 4 to convert the addresses. When in the **modify** state, we add the tilde character before each letter in the e-mail address. After we encounter the at sign, we transition to the **none** state and stop making changes. For example:

water@code.com ⇒ ~w~a~t~e~r~@code.com

We begin with a Water object that iterates over an input string.

```
<defclass obfuscator>
  <defmethod run addr>
    addr.<for_each combiner=insert>
      _subject.<apply value/>
    </>.<join/>
  </>
  obfuscator.<set apply=modify/>
</>
```

The **<obfuscator/>** class contains two methods, **<run/>** and **<apply/>**. **<run/>** simply calls the **<apply/>** method on each character in the **value** parameter and **<join/>**s the result of all the **<apply/>** invocations to create the final string. The **apply** field holds a method that implements the current state.

We represent the states of the machine in Figure 4 using two methods:

```
<defmethod nothing letter>
  letter
</>
<defmethod modify letter>
  <if> letter.<is <char "@"/> />
    _subject.<set apply=nothing/>
  </>
  "~".<concat letter/>
</>
```

When a newly created **<obfuscator/>** instance receives an **<apply/>** message, it searches its instance for an **apply** field. The instance does not have an **apply** field, so the message is delegated to **<obfuscator/>**, which uses the **<modify/>** method to add a tilde before the current letter. When the at sign is encountered in the **modify** state, we **<set/>** the instance's **<apply/>** field to use the **<nothing/>** method instead, effectively transitioning to the nothing state. Successive searches for **<apply/>** in the instance will be successful and **<nothing/>**

is executed rather than delegating the message to the **<obfuscator/>** class.

Using delegation to implement a state machine is simpler than equivalent methods in consultation languages, where a subclass must be created for each state [Rai 94]. See the State pattern in the Gang of Four book for an extended example [GHJ 02].

Water uses the delegation model to implement multiple inheritance by allowing multiple objects to be included in the delegation search. This approach is similar to the proposed multiple inheritance implementation for Self in [US 87]. The utility of multiple inheritance is a heavily debated topic [SCC 93], but it is an obvious outgrowth of the inheritance architecture, and we believe that students should be introduced to the concept. In our lecture series, we introduce multiple inheritance as a way to reuse common constructs, such as logging and security, across unrelated objects.

For example, if we want to include logging functionality in a Water program, we can use multiple inheritance to give each object native logging capabilities. First we define a logging class:

```
<defclass logger>
  <defmethod log msg>
    <echo msg/>
  </>
</>
```

Then we use the **<union/>** method to create an object that has more than one parent.

```
<set oblog=<union obfuscator logger/> />
```

The object returned by the **<union/>** method is new class that delegate all messages to obfuscator, then to logger, until a match is found. **<union/>** can take more than two arguments; delegation propagates down each provided class until the first match is found. We can use the new **oblog** class to obfuscate an e-mail and log the results:

```
<set example=<oblog/> />
example.<log
  example.<run "logging@code.com"/>
/>
```

First, we create a new oblog object and store it into the variable **example**. Then, we log the result of the **<run/>** call. The output remains the same, since our current logging function only logs to standard output, but we could easily alter the logger to output to a file:

```
<defclass logger>
  logger.<set logfile=<file
    <uri
      "logical://user/log"
    /> />.<create/>
  />
  <defmethod log msg>
    _subject.logfile.<insert msg.<concat
"\n"/> />
    msg
  </>
</>
```

Now each call to the **<log/>** method appends the message to the end of a file named log in the user's home directory. All classes that are **<union/>**ed with logger now log to a file instead of standard output.

5. Lecture Usage

We have used a preliminary version of our lectures to teach Water in three weeks as part of a graduate-level survey course in object technology. Students in the course were familiar with both Java and Smalltalk, and were also introduced to the Java implementation of Web services after the Water lectures were completed.

At the conclusion of the three-week module, the students were asked to create a Web service that collected book prices from several on-line bookstores. The original bookstores targeted in the assignment were not actual Web services, so the students were required to search for the price in the HTML content. The original assignment can be seen at http://courses.ncsu.edu/csc517/common/homework/program4.html.

6. Conclusion

Using Water to teach Web services has several advantages. First, Water is a high-level way of getting the concepts across. Code is very compact, and common operations are built in, like reading XML to create objects and normalizing the representation of those objects. These allow the student to focus on what is being done, rather than how to do it. We have given some trivial examples, and two more detailed examples, that illustrate the power of Water. Second, the high-level nature of Water helps a programmer visualize a service-oriented architecture in terms of object-oriented concepts such as message passing, interfaces, and loose coupling. Third, Water's object model is quite rich. It supports different styles of programming, such as inheritance vs. prototype-based programming, multiple inheritance, and user-defined control structures using the **for_each** statement. Thus, Water allows us not only to teach Web services quickly, but simultaneously to enhance students' skills in object orientation.

7. References

[DaP] Darwin Project, The. *What is (Not) Delegation*. http://javalab.cs.uni-bonn.de/research/darwin/delegation.html

[DDDT 02] Deitel, Harvey M., Deitel, Paul J., DuWaldt, B., Trees, L. K., *Web Services: A Technical Introduction*, Prentice-Hall, 2002

[GHJ 02] Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, October, 2002.

[IBM 05] IBM developerWorks. SOA and Web services: New to SOA and Web services. http://www-128.ibm.com/developerworks/webservices/newto/.

[KG 04] Kachru, Sandeep and Gehringer, Edward F., On the relative advantages of teaching Web services in J2EE vs. .NET, *OOPSLA 2004 Educators' Symposium*.

[Lieb 86] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of*

the 1st Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86) (Portland, Oregon) 214-223, 1986.

[LJM 05] Lim, Billy B. L., Jong, Chu, and Mahatanakoon, Pruthikrai, On integrating Web services from the ground up into CS1/CS2, *Proc. 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, 241–245, Feb. 2005.

[MS 05] Møller, A., Schwartzbach, M, *An Introduction to XML and Web Technologies*, forthcoming in October 2005

[NCSU 05a] CSC 750, Service-Oriented Computing, http://www.csc.ncsu.edu:8080/courses/graduate/index.php#CSC750

[NCSU 05b] CSC 450, Web Services http://www.csc.ncsu.edu:8080/courses/undergrad/index.php#CSC450

[Ort 04] Ortiz, Sixto Jr. *Web Services: The Next Phase*. http://www.processor.com/editorial/article.asp?article=articles/p2628/34p28/34p28.asp, July 9, 2004.

[Plus 02] Plusch, Mike. *Water: Simplified Web Services and XML Programming*. http://waterlanguage.org/water_book_2002/index.htm.

[Rai 94] Räihä, Liisa. Delegation: dynamic specialization. In *Proceedings of the conference on TRI-Ada '94* (Baltimore, Maryland) 172 - 179, 1994.

[Ric 04] Ricadela, A., Foley, J. *New Face Of E-Commerce*, *InternetWeek.com*, http://www.internetweek.com/story/showArticle.jhtml?articleID=25600346, July 27, 2004.

[SCC 93] Shan, Yen-Ping, et al. Is multiple inheritance essential to OOP? In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. (Washington, D.C.) 360-363, 1993.

[See 04] Seely, R. *Competing standards hurt Web services growth*. http://www.adtmag.com/article.asp?id=7571, July 16, 2004.

[US 87] D. Ungar and R.B. Smith. SELF: The Power of Simplicity. In *Proceedings of 2nd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)* (Orlando, FL) 227-242, 1987.

[Wate 05] Water: The clear solution for Web programming, http://www.waterlanguage.org

Appendix A: Normalization

Water is adept at importing XML documents into object, but unfortunately, the result is poorly formatted. An HTML representation of an unformatted **<flickr/>** object is shown in Figure 5. We can force Water to return a useful object by using the **<normalize/>** method:

```
<normalize <flickr text="water"/>.<to_xml/> />
```

a rsp				
stat	ok			
content	a thing			
	0	a photos		
		pages	5673	
		total	5673	
		perpage	100	
		page	1	
	content	a thing		
		0	a photo 6738571	
			isfamily	0
			id	6702150
			owner	49503205076@N01
			secret	0e98f0349d
			server	8
			title	Maxwell
			ispublic	1
			isfriend	0
		...more photo objects...		

Figure 5: A Water XML Object before Normalization

a a symbol rsp			
stat	ok		
photos	a a symbol photos		
	pages	5673	
	perpage	100	
	page	1	
	total	5673	
	0	a a symbol photo	
		isfamily	0
		id	6702150
		owner	49503205076@N01
		secret	0e98f0349d
		server	8
		title	Maxwell
		ispublic	1
		isfriend	0
	...more photo objects...		

Figure 6: A Normalized Water XML Object

<normalize/> converts an XML string to its intuitive object representation. The resulting object is shown in Figure 6. Notice the correlation between the structure of this object and the original Flickr XML output. The normalized object is more compact than the original object in Figure 5 and the key names are more sensible. For example, the original object stored the first <photo/> in the **content.0.content.0** field, while the normalized object stores the first **<photo/>** object in the **photos.0** field.

Appendix B: Flickr Example

```
<!--
  Flickr Web service reducer. This code can be executed in the Steam
  IDE from http://www.waterlanguage.org.
-->

<!--
  Create a <flickr/> method that uses the Flickr Web service:

http://www.flickr.com/services/rest/?method=flickr.photos.search&api_key=a04183d905bac769bddb
9f5a78c4aa24
  and appends the text parameter with each call.
-->
<set flickr=<web
    <uri "http://www.flickr.com/services/rest/"
      query="method=flickr.photos.search&api_key=a04183d905bac769bddb9f5a78c4aa24"
    />
    contract=<defmethod text=required=string/>
  />
/>


<!--
  Create a class to hold the simplified Flickr information.
-->
<defclass simple_flickr title src/>

<!--
  When we call the <flickr/> Web service method, the XML response will
  be executed as Water code. Therefore, we need classes to represent each
  tag that is contained in the response.
-->
<defclass rsp stat/>
<defclass photos pages total perpage page/>
<defclass photo isfamily id owner secret server title ispublic isfriend/>

<!--
  The main body of the Web service.
-->
<defmethod flickr_reducer keyword>
  <set result=<normalize <flickr text=keyword/>.<to_xml/> /> />
  result.photos.<for_each combiner=insert include=vector_key>
    <simple_flickr value.title
      <vector
        "http://photos" value.server ".flickr.com/"
        value.id "_" value.secret "_m.jpg"
      />.<join/>
    />
  </>
</>

<!--
  Create a server to access the Web service.
-->
<server thing port=8080/>
```