A Generic Framework for Engaging Online Data Sources in Introductory Programming Courses

Nadeem Abdul Hamid
Berry College
Mount Berry, GA 30149
nadeem@acm.org

ABSTRACT

This paper presents work on a code framework and methodology to facilitate the introduction of large, real-time, online data sources into introductory (or advanced) Computer Science courses. The framework is generic in the sense that no prior scaffolding or template specification is needed to make the data accessible, as long as the source uses a standard format such as XML, CSV, or JSON. The implementation described here maintains minimal syntactic overhead while relieving novice programmers from low-level issues of parsing raw data from a web-based data source. It interfaces directly with data structures and representations defined by the students themselves, rather than predefined and supplied by the library. Together, these features allow students and instructors to focus on algorithmic aspects of processing a wide variety of live and large data sources, without having to deal with low-level connection, parsing, extraction, and data binding. The library, available at http://cs.berry.edu/big-data, has been used in an introductory programming course based on Processing.

Keywords

open data; XML; introductory CS; library; web services

1. INTRODUCTION

The average student in an introductory programming course today has grown up interacting with large and real-time sources of data from the Internet on a daily basis. Programming exercises and projects they encounter in a traditional CS1/2 course, however, are often limited to contrived input data, often student-generated and supplied through "standard input" or a simple GUI mechanism. This dichotomy, in terms of the data that students interact with, may contribute to a disconnect between students' experience with mainstream computing and their perception of Computer Science as they work through their first programming course.

As Bart et al. discuss in [5], there are a couple of compelling reasons for instructors to consider incorporating the use of real-time web-based data in courses early in the Computer Science (CS) curriculum. Primarily, this might contribute to meeting students' expectations of a CS1/2 experience that is engaging and relevant and through which they can draw some connection to the web services and mobile apps that they are used to interacting with every day.

ITiCSE '16, July 09 - 13, 2016, Arequipa, Peru

ISBN 978-1-4503-4231-5/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2899415.2899437

Instances may be found of exercises for introductory courses (e.g. [3]) involving, for example, social media, weather, or financial data, but the frameworks developed for such assignments is often ad hoc. Instructors usually prepare a code module ahead of time that carries out the tasks of connecting to a web service and parsing data (into an instructor-defined data structure), or they provide pre-processed downloaded sets of data for students to use in an assignment.[1] When attempting to incorporate "live," online data sources into an assignment there are a few major difficulties that an instructor may face. In particular, every web service provides a different API for accessing data. Furthermore, the raw data, whether directly accessed through a web service API or read from locally downloaded data files, must be parsed, and often only a subset of the information associated with each record in the data is of interest.

In the project described here, we develop a code framework and methodology to facilitate the incorporation of real, online data sets into traditional introductory programming courses. The framework is generic in the sense that (in most instances) no prior scaffolding or template specification is needed to make the data accessible. Our implementation aims to maintain minimal syntactic overhead while relieving novice programmers from issues of parsing and extracting raw data from a web-based data source. This allows students and instructors to focus on algorithmic aspects of engaging with arbitrary data sources, without having to deal with low-level issues related to accessing the data.

Our motivation and approach is similar to that of [5] with a couple of significantly novel features, highlighted in Section 2. In the sections that follow, we detail the particular requirements and rationale for our framework followed by concrete examples of its use. We discuss our initial implementation of a library in Java and its use (limited so far) in an introductory programming course. Finally, we wrap up with an evaluation of the current implementation, future directions and related work.

2. REQUIREMENTS AND RATIONALE

The overarching motivation for this project (like [5]) is to provide an interface to a wide variety of web-based data

[1] The Nifty Assignments archive, nifty.stanford.edu, has multiple examples of these approaches.

sources suitable for use by novice programmers. This leads to a set of requirements between which some tension exists.

Firstly, we seek to access data directly through a web URL (though locally-downloaded files are also supported) and relieve the programmer (i.e. student) from low-level issues of reading and parsing raw data formats like CSV, JSON, or XML. Simultaneously, we would like to require minimal syntactic overhead for use of the framework. Loading and parsing primitive or simple structured data should not involve much more than what is involved in reading from standard or file input (e.g. using a `Scanner` object in Java). The library should incorporate a cache to reduce load on data services and improve local runtime performance.

Another key requirement we aim to support (and that is a significant difference from the work of Bart et al. [5], discussed further in Section 6) is the ability to instantiate data objects based on student-defined data structures and representations. In Java, for example, students should be able to develop their own `class` definitions for the data of interest, independently of the data source or the library. Data accessed through our framework should then use the student-supplied definitions to construct objects in the program when it is run. Our anecdotal experience is that this enhances students' feeling of achievement. They see that they are really defining the structure and organization of the data that is eventually fetched, rather that having data structures already defined and fixed for them in the black box provided by some library. This serves as an active and motivating context to practice developing and defining data representations, rather than passively dealing with pre-defined structures.

Most of the time, data sets will contain many more fields of information than are of interest to the student, who should be able to identify and select data fields of interest in a straightforward manner. Additionally, the framework should enable retrieval of entire lists or arrays of objects (again, based on student-defined classes/structures) from the data source whenever appropriate. Missing data should be handled elegantly, e.g. by filling in with sensible default values.

In addition to having data binding occur dynamically at runtime, as described in the preceding paragraphs, we would also like to eliminate the need (as much as possible) for presupplied data schemas or templates to parse and process the data. This is another significant novelty with respect to [5]. We anticipate that this will make it even more feasible and attractive for instructors and students to engage with a wide variety of data sources since, as long as a service supplies data in a standard format, there is no need for prior generation of a client library or preprocessing template. While carrying out automated structure inference of data in common formats like XML, the framework should nonetheless provide mechanisms to specify a schema for the data, as well as enable custom extensions for connecting to and parsing additional data formats.

Finally, to enhance usability by novice programmers, the framework should incorporate built-in help/usage functionality and robust error-handling and reporting.

3. EXAMPLES OF USE

3.1 Basic Usage

To access any data source using our framework, there are three basic steps to be carried out: *connect* to the data source via URL or filename (includes setting options/parameters), *load* the data, and *fetch* elements of interest. The following complete, minimal example connects to the WorldCat online library catalog and fetches the title of a particular book.

```
import easy.data.*;

public class WorldCatSimple {
  public static void main(String[] args) {
     DataSource ds = DataSource.connect("http://xisbn.worldcat.org"
         + "/webservices/xid/isbn/9780201038019"
         + "?method=getMetadata&fl=*");              // step 1
     ds.load();                                      // step 2
     String title = ds.fetchString("isbn/title");    // step 3
     System.out.println(title);
  }
}
```

Note that other than downloading the JAR file for the library and including it in one's classpath, no other prior setup specific to the WorldCat data source is required. The `connect` method automatically infers the data format (XML in this case) and prepares to download it or access a cached copy. (If necessary, connection methods to force a specific format, such as `connectXML`, `connectJSON`, etc., can be used.) The `load` method carries out some simple structure inference on the raw data and exposes a set of labeled fields and structures that can be accessed. The actual XML returned by the URL in this example is:

```
<?xml version="1.0" encoding="UTF-8"?>
<rsp xmlns="http://worldcat.org/xid/isbn/" stat="ok">
      <isbn   oclcnum="311084141 ..." form="BA" year="1975"
      lang="eng" ed="2. ed., 1. print."  author="Donald E. Knuth."
      title="The art of computer programming"
      ...>9780201038019</isbn>
 </rsp>
```

The set of inferred data fields is available by invoking the method `ds.printUsageString()`. It is up to the user to fetch the data as an appropriate type (`String`, `int`, etc.).

Our framework requires dealing with URLs of web services, as may be observed in the example above. This requires some discussion of concepts such as URL paths and query parameters. To obviate the need to construct raw query strings, i.e. `"?method=..."`, one may use `set` methods on the `DataSource` object, as shown in the left half of Figure 1. The nature of the exposed data also needs to be explained since it often involves nested structures. As in this example, the fields are accessed by paths like `"isbn/author"`. (The examples in Figure 1 compose, or chain, method calls for brevity of presentation.)

3.2 Structured Data and Collections

In the `APStatus` program on the right side of Figure 1, the student has defined their own class, `AP`, with several fields to represent airport information. The data source is loaded and the `fetch` method is provided the name of the class along with the paths of fields to extract from the data, in the order they should be provided to the `AP` constructor.

If the data source provides a list of data elements (e.g. a list of airport statuses), one could simply use a `fetchList` (or `fetchArray`) method instead of `fetch`. The library intelligently unifies the available data with the requested object type(s). Again, no prior preparation is required to access this particular data source using the `easy.data` library. Thus, in a few statements of code, a student is able to load, parse, and instantiate hundreds or thousands of objects (of a user-defined class) from the data.

```
import easy.data.*;

public class WorldCat {
   public static void main(String[] args) {
      String isbn = "9780201038019";   // could be user input
      DataSource ds = DataSource.connect(
         "http://xisbn.worldcat.org/webservices/xid/isbn/"
         + isbn);
      ds.set("method", "getMetadata").set("fl", "*");
      ds.set("format", "xml").load();
      String title = ds.fetchString("isbn/title");
      String author = ds.fetchString("isbn/author");
      int year = ds.fetchInt("isbn/year");
      System.out.printf("%s, %s, %d.", title, author, year);
   }
}
// OUTPUT:
// The art of computer programming, Donald E. Knuth., 1975.
```

```
public class APStatus {
   public static void main(String[] args) {
      DataSource ds = DataSource
        .connect("http://services.faa.gov/airport/status/BOS")
        .set("format", "application/xml").load();
      AP x = ds.fetch("AP", "Name", "State", "Delay",
                            "Weather/Weather");
      System.out.println(x);
   }
}

class AP {
   ...
   public AP(String name, String place,
            boolean delay, String weather) { ... }
   ...
}
```

Figure 1: Sample Java programs.

3.3 Data Source Specifications

While having an automated framework is convenient in many instances, there are times when it might be more practical for an instructor to provide a data schema ahead of time and/or be able to set various options or parameters for a particular data source. Our framework provides a facility to do so in the form of *specification files*. These are XML format files that may contain the data source URL and format, a human-friendly description and data schema with annotations, pre-supplied and user-supplied (required and optional) query parameters or path parameters, options specific to a particular type of data source, cache settings, etc. Details and examples are omitted here for lack of space.

4. DESIGN AND IMPLEMENTATION

As demonstrated in the preceding section, we have currently implemented a Java library to satisfy the requirements of the framework envisioned in Section 2. A call to `connect()` prepares the URL (or filename) path, accumulating parameters and options via the `set()` methods, and attempts to infer the data format based on the path extension and potentially the data itself. When `load()` is invoked, the library actually accesses the data source, downloads the data, and analyzes it to infer its structure, producing a *field schema*. A call to `fetch*()` provides the library with the type of object that the programmer wishes to have data instantiated as - either as a primitive, using an arbitrary class constructor with an ordered list of fields names of interest, or as a list of either of the previous. The library uses the lightweight annotation provided by `fetch*()` to build a *type signature* for the desired object(s) to be produced from the data. It then instantiates it by unifying the signature against the field schema to determine how exactly to extract the data and produce an object(s) to be returned to the programmer.

The initial version of our library uses XML as its internal format for data representation. A field schema is either a primitive field, or a list or structure of arbitrarily nested field schemas. Structure inference is performed by a fairly straightforward function that recursively analyzes the XML document node. A node with no tagged sub-nodes is inferred to be a primitive field. In the event sub-tags occur, each tag name may occur once or multiple times. The entire node is then mapped to a structure field schema with each child tag associated with the field schema inferred for the subnode. The inferred field schema for tags that occur multiple times is wrapped as a list. Throughout this process, field schema elements are annotated with XML tag names (or paths, in general) to enable navigating through the data later based on the schema. To account for XML attributes, the data is preprocessed to turn all attributes of elements into child nodes (as in the example of Section 3.1).

Type signatures are similar to field schemas except the signature for a structure is annotated with a Java `Class` object, and primitive signatures are associated with a specific Java base type (`int`, `String`, etc.). While signatures may be arbitrarily nested internally, the library's `fetch` methods currently only enable specification of flat objects (i.e. objects with primitive type fields, or lists of those).

The process of instantiating a type signature against a field schema and associated raw data is more complicated. In effect, the supplied signature must be unified against the field schema while simultaneously extracting data and constructing Java objects. The simplest case is when the shape of the signature precisely matches that of the schema. However, the library also attempts to handle other cases as well. For example, if the field schema is that of a list, but a primitive or structure type signature is supplied, then the subschema of the list elements is unified against the first element in the actual data and returned as a Java object. On the other hand, if the field schema is that of a structure, but a list type signature is provided, then the library attempts to unify the structure against the sub-signature of the list, and wraps the result as a one-element `ArrayList`.

For brevity, the complete details of structure inference and object instantiation are omitted here. It bears mentioning, however, that use of the Java Reflection library (`java.lang.reflect`) is critical to achieve tasks such as reifying a `Class` object from a name (a `String` such as `"AP"`) and invoking arbitrary Java constructors at runtime. As this library has been initially used in a course based on the Processing [2] development environment, some additional management of runtime configuration had to be implemented. Processing allows developers to write *sketches*, using a subset of Java, which are then embedded into a hidden Java applet template that does various behind-the-scenes manipulation (e.g. of environment and directory paths).

The library handles other standard data formats (currently, comma-separated and tab-separated values, and JSON)[2]

[2]An extension to scrape tables or other arbitrary data from HTML pages via regular expressions is also in development.

by converting them to an XML object before processing them, but future work involves developing more direct methods of parsing and binding the data, to improve performance on large data sets.

5. EVALUATION

We evaluate the library of this paper from two perspectives: one in terms of its use in an introductory programming course, and secondly in terms of how well its current implementation meets the requirements and goals described earlier in the paper.

5.1 In the Classroom

No formal study has been performed on the library's use in a course, but the author has used this library for two years in an introductory programming course based on Processing [2], a Java-based IDE and set of libraries geared towards programming in a visual arts context. In the most recent offering of the course (Fall 2014), the library was introduced in the last month of the semester. (In the future, we anticipate integrating the material earlier on.)

Students were provided a set of tutorial-style labs to work through (available on the project GitHub site). The tutorials gradually progress through a hierarchy of data complexity from accessing primitive data, to simple objects, then lists of primitive types, and finally lists of objects. They introduce concepts such as URLs, the nature of data schemas, caching, and dealing with incomplete data (many real-world data sets have records that are missing fields). For the final project in the course, students were asked to find a data source on their own and develop an interactive GUI visualization of the data. Figure 2 lists some data sources that were presented in lecture and chosen by students for their projects. (A large list of sites to browse for data sources was provided by the instructor.) Almost all the sources were automatically processed by the library directly via the data source URL. In a couple of cases, students manually created a CSV file from an HTML table on a web site.

One issue that arose in some instances was finding a link to the raw data provided by a service. The primary URL for many web services often provides an interactive interface or visualization of the data. It takes some digging around to find a direct URL to the underlying XML data or a JSON rendering. Related to this, some students had initial difficulty understanding the distinction between data rendered in a human-friendly form (such as is visible when a web page is viewed in a browser), and the raw data in a format amenable to machine-processing by their program. When they found a web page that displayed a set of data in tabular form, they would attempt to use that page's URL to `connect` to the data source. Nonetheless, this serves an opportunity for a "learning moment."

Another concern that did not seem to be a major problem, but may have limited the choice of data sources, was that some sites require "developer" registration and an API key to be obtained in order to access data. Some students were hesitant to register for such. Also, this required various query parameters to be `set` on the `DataSource` object before it is loaded. Determining the necessary parameters involved reading into the developer documentation of a particular web service, an experience which varies in pleasantness from site to site.

Overall, insofar as operation of the library itself was concerned, things seemed to go fairly smoothly. As noted earlier, introducing the library earlier on in the course, and integrating it more thoroughly with the course content would make the students more comfortable with it and allow them a better sense of how to access data for the purpose of analysis or visualization. And, of course, more extensive field studies, using the library in other courses and institutions, are necessary to formally gauge its true usefulness.

5.2 Technical Challenges

Based on the limited testing and use so far, our current implementation appears to meet many of the goals laid out in Section 2. However, there are a few areas where improvement is clearly needed. One is to provide a more robust approach to error handling and reporting. Currently, exceptions are raised when things do not match up as expected at various stages of loading and instantiating data but the error messages can be improved. This could be achieved partly by providing a high-level description of the source of a problem, rather than just a line number in a file, to help a novice programmer understand what happened.

As alluded to in the previous section, one complication in accessing data sources is providing a user-specific API key or access token. While this can be achieved using the existing `set` methods, it can be frustrating for students to figure out how to specify the necessary parameters. Also, rather than having user API tokens exposed directly in source code of a program, which might be shared with others, it would be better to have private user data stored in a separate file. We need to investigate a mechanism for doing this in a simple manner, combined with a curated set of data source specifications for common APIs (Ebay, Twitter, etc.) that require special parameters, to make it easier for students to access these. Also, a GUI tool for managing data source-related parameters, as mentioned in the next section, might be very helpful.

In terms of performance, the library handles many small to medium-size data sets well. However, the current implementation loads the entire data set into memory. Along with the adoption of XML as a common intermediate format, this results in significant delays when loading large data sets (tens of thousands of records). Even with caching, there is at least a two-fold delay loading a CSV file, for example. First the data must be loaded and converted to XML; then the entire converted data is scanned again to infer its schema. Depending on the `fetch` operation that is then executed, there may be further delay in processing. Even with intelligent caching, XML is not the most efficient representation of data and in many cases it would be much more efficient (time and space) to access CSV data, for example, using a data structure other than an XML object.

A complete reorganization of the internals of the library is currently in progress. It abandons XML as the common intermediate format and adopts more abstract mechanisms to encapsulate arbitrary data format objects, allowing them to be traversed using a uniform interface. The initial focus of our project was on dealing with XML-based data, as it seemed the most general and challenging representation to handle. Now that we have established the viability of the approach presented in this paper, we are in the process of refining the architecture of our initial implementation. A

Name (Asterisk * indicates data set discovered and/or used by students)	Source	Type	Records
*1000 songs to hear before you die	opendata.socrata.com	XML	1,000
Abalone data set	UCI Machine Learning Repository	CSV	4,177
*Airport Weather Mashup	NWS + FAA	XML	fixed
*Chicago life expectancy by community	data.cityofchicago.org	XML	˜80
Earthquake feeds	US Geological Survey	JSON	variable
*Fuel economy data	US EPA	XML	35,430
*Jeopardy! question archive	reddit	JSON	216,930
Live auction data	Ebay	XML	100/page
Magic the Gathering card data	mtgjson.com	JSON	variable
Microfinance loan data	Kiva	XML	variable
*SEC Rushing Leaders 2014	ESPN	CSV (manual)	variable

Figure 2: Examples of data sources used in lecture examples and student final projects.

complementary feature that also remains to be explored is to provide methods for streaming and/or paginating data.

5.3 Additional Future Directions

In addition to addressing the more immediate technical challenges of the previous section, there are some other ideas for future development that would improve the library's usefulness. One is the ability to invoke a GUI interface to configure various settings for the library and parameters for a particular data source. Upon loading a data source, it could also allow the user to view usage info (as an alternative to `printUsageInfo`), preview the available data, provide help for invoking methods of the library, and perhaps even generate code snippets.

As mentioned in Section 4, the `fetch()` methods currently only support the specification of flat signatures. It is possible to directly construct and provide a type signature object to the library that involves, for example, lists of structures with nested lists or structures. Providing a simplified way to supply such signatures via `fetch()` remains future work, and the process of data instantiation against completely general signatures is an interesting challenge that needs to be studied further.

We chose Java as the initial language in which to implement the ideas of this paper. Since Java is widely used in introductory programming courses, this should make the library accessible to a large number of instructors if they are interested in trying it out. Nonetheless, we hope to also provide realizations of the framework in other languages, such as Racket [9] and Python. The initial inspiration for our work came from the author's reimplementation [11] of a Racket "teachpack" (initially developed by Shriram Krishnamurthi and Kathi Fisler) to access Kiva data through its XML API. In Racket, a language derived from Scheme, macro facilities were used, instead of reflection, to realize "dynamic" binding of data to user-defined structures. Macros allow expansion of code prior to execution. The details of the implementation thus have a different flavor than that of the Java version, and even allow some compile-time checks (such as ensuring, for example, that the number of fields specified in a `fetch()` expression match the number of parameters of a constructor). Another interesting feature of the Kiva library was that it provided a small, fixed subset of the data for testing purposes. This would be a useful feature to incorporate in our Java implementation– a way to cache a small portion of data and make it available for development and testing, eliminating the need to load the entire data set for large sources of data until the final run of the program.

6. RELATED WORK

As cited in the Introduction, our motivation mirrors that of RealTimeWeb [5], a framework that makes real-time web data accessible for introductory programming projects, although the approach differs substantially. RealTimeWeb (now renamed the CORGIS [4] project) utilizes a JSON-formatted "client library specification" to generate specialized code in one of several languages - Java, Python, and Racket. The generated library code (which requires additional manual tweaking) can then be packaged, imported into a student program, and used to access data from web sources. This approach is similar to that of mainstream tools, discussed later in this section, in that a schema of the data source as well as the client language API (what methods the generated code will provide to access selected information) must be manually specified (e.g. by an instructor) and then compiled to produce actual code for the target language. While the CORGIS project envisions development of a curated gallery where many client libraries may be shared (and a number have indeed been produced to date), we believe this approach limits the flexibility of the framework because a new data source cannot be accessed until a specification is defined and a client library generated and tested.

While our framework also supports use of a data source specification file to provide a simple schema for accessing a data source, our primary motivation and ideal is to develop a library that dynamically and automatically infers the structure of data simply upon connection to the data source. We believe this improves the usability and applicability of such a library because instructors can choose to access any online data source, not just ones for which a client library exists. Furthermore, students are able to access data sources of their own choosing if they so wish, for example, if working on a project idea developed by themselves. At the same time, the syntactic complexity of invoking our library remains barely more than that of a CORGIS library. For example, to implement the same example as in Figure 2 of [5], we replace the first two statements of the `main` method with:

```
DataSource ds = DataSource.connectJSON("http://earthquake.usgs"
       + ".gov/earthquakes/feed/v1.0/summary/all_hour.geojson");
```

and replace the statement containing the `getEarthquakes` method call with:

```
ds.load();
List<Q> qs = ds.fetchList("Q", "features/properties/title", ...);
// Q is a user-defined class with fields of interest for a quake...
```

The only arguable complexity introduced here is the exposure of the underlying URL for the data source and the paths to the fields of interest. We do not believe this is excessively burdensome on students, and in fact it may increase the sense of "realness" of their engagement with the data.

Beyond the pedagogical context, the approach this paper takes towards interfacing between (primarily) XML and Java falls in the category of *vocabulary-specific data access interfaces* (DAIs) [13]. These tools map data from XML to application-specific data structures. Such an approach enables developers to work with data in a concise manner that fits into the semantics of their particular application and language. Among the most popular "real world" tools of this nature for Java currently are the Java Architecture for XML Binding (JAXB) [1] and Castor [8]. These and all similar libraries currently available, to our knowledge, provide an automated way to generate Java class definitions from an XML schema and/or vice versa, and then facilities for serializing/deserializing Java objects to/from XML data. By design, the data binding in these libraries is two-way, and they all incorporate both static (compile-time) as well as dynamic (run-time) aspects. In particular, there is a very tight correspondence between the automatically-generated Java or XML schema definitions, and customization involves adding various types of annotations, which can become complex.

In contrast to such frameworks, the work presented in this paper focuses on a constrained, one-way flow of data: from XML to Java objects. Also, we relax the tight coupling between the Java and XML representations, allowing the developer (student) to define Java classes more or less as they wish, and then later specify how to bind XML data to them. Furthermore, our process is entirely dynamic - we do not require any static generation of class or schema definitions prior to binding data.

As the framework of this paper purposefully avoids the necessity of a predefined schema or XML data type definition, it must perform inference of XML data structure at runtime. There has been much prior work on inference and extraction of XML document structure and schema, e.g. [10, 6, 7], and there are mature commercial tools for doing so, e.g. [12]. Again, however, for our context we do not need to support the full generality of inference of arbitrary structure and properties of XML data. It is sufficient to determine only enough structure to be able to bind data to programmer-defined object types based on some simple hints.

7. CONCLUSION

A number of contextualized approaches to teaching introductory Computer Science courses have been developed in the past decade, catering to students with different interests and backgrounds. For instance, entire courses have been developed around media computation or robots (real and virtual). There is, however, one context which, to our knowledge, has not been exploited in a systematic fashion – that of "live data," or widely available, large and live online datasets from a wide variety of sources. Engaging novice programming students with such data sources may be a powerfully engaging experience that adds relevance to their introductory experience in Computer Science. In order to develop a course centered around this idea, however, a software toolbox is needed that enables instructors and students to rapidly access data of interest from a web service without becoming bogged down in low-level details of I/O, parsing, etc. We hope that the ideas of the framework described in this paper, and the implementation presented, are a further step towards achieving this vision. In addition to the interesting software architecture challenges that remain open, the next step is to develop a complete set of materials (lecture examples, labs, programming projects, etc.) to support such a course.

8. REFERENCES

[1] Java architecture for XML binding (JAXB). https://jaxb.java.net. September 2014.

[2] Processing. http://processing.org/overview/. June 2015.

[3] R. E. Anderson, M. D. Ernst, R. Ordóñez, P. Pham, and B. Tribelhorn. A data programming CS1 course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 150–155, New York, NY, USA, 2015. ACM.

[4] A. C. Bart. Situating computational thinking with big data: Pedagogy and technology (abstract only). In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 719–719, New York, NY, USA, 2015. ACM.

[5] A. C. Bart, E. Tilevich, S. Hall, T. Allevato, and C. A. Shaffer. Transforming introductory computer science projects via real-time web data. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 289–294, New York, NY, USA, 2014. ACM.

[6] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 115–126. VLDB Endowment, 2006.

[7] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 998–1009. VLDB Endowment, 2007.

[8] ExoLab Group. The castor project. http://castor.codehaus.org. September 2014.

[9] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. http://racket-lang.org/tr1/.

[10] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. Xtract: Learning document type descriptors from xml document collections. *Data Min. Knowl. Discov.*, 7(1):23–56, Jan. 2003.

[11] N. A. Hamid. Kiva teachpack for DrRacket. http://github.com/nadeemabdulhamid/Kiva-Teachpack. June 2015.

[12] Microsoft Corporation. Inferring an XML schema. http://msdn.microsoft.com/en-us/library/b6kwb7fd(v=vs.110).aspx. September 2014.

[13] J. White, B. Kolpackov, B. Natarajan, and D. C. Schmidt. Reducing application code complexity with vocabulary-specific XML language bindings. In *Proceedings of the 43rd Annual Southeast Regional Conference - Volume 2*, ACM-SE 43, pages 281–287, New York, NY, USA, 2005. ACM.