Collaborative SOA Prototyping with the *Delve SOA Fast-Prototyping Toolkit*

By Paul Toth, CEO, Cretaceous Software

Introduction

Delve is a collaborative fast-prototyping SDK for Tectonic, a service-oriented language. It is intended for use in the early stages of an SOA project, when architects, business analysts, and designers are focused on brainstorming prospective XML data formats and service interfaces. Working in Tectonic allows those considering the validity and practicality of composing, deploying, and consuming a set of services to move from whiteboard to functional prototype in a very short time, without the need to employ dedicated developers. For instance, it is often possible to write the code required to precisely represent an XML-Schema data type in less time than it takes to verbally describe that type.

Once prototypes are deployed, Delve gives architects the ability to modify schemas and service interfaces quickly and with minimal impact to other prototype components. This means that planners can rapidly iterate through many prospective versions of a structured data type or functional element without becoming bogged down in implementation details. The ability to execute short, focused iterations, which is key to modern software development methodologies such as *RUP*, *Agile,* and *Extreme Programming,* is at the core of Delve's value proposition.

At the completion of each stage in the development process, Delve helps generate essential work products. Candidate schemas are created for use in draft requirements documents. Sample SOAP messages and WSDLs can be incorporated into architectural specifications. Finally, design documents including XPath statements and references to live service prototypes are passed on to developers. Delve makes it easy to generate rich deliverables that consist of more than the simple prose and diagrams with which most teams make due.

Use of Delve makes it possible to employ an architectural strategy known as *Immersive SOA.* Practitioners of this strategy make service oriented concepts first and foremost when approaching requirements and design tasks. This means, for instance, that a newly proposed service is conceptualized, described, and designed in an "interface-first" manner, with definition of XML-Schema types and a service WSDL preceding any development work. Tectonic's service oriented nature makes its use an ideal component of such a strategy, allowing users to interact with constructs such as XML-Schema types and structured documents as essential components of the language itself.

In this article, we will explore a number of ways in which Delve enables SOA fast-prototyping and, consequently, utilization of Immersive SOA. A number of code samples are provided, all of which are intended to be run in Delve.

Composing Schemas

Suppose one prospective component of a solution architecture consists of a Web service for managing business development activities. A quick round of brainstorming reveals that one of the most commonly used domain objects is a "person," which can in turn be specialized to represent contacts, internal bizdev specialists, and so on. Since we're employing Immersive SOA, the next step is to quickly draft a person type using Tectonic.

```
schema bizdev "http://www.proto.com/schemas/bizdev"
 type person
  fname;
  lname;
  mid;
 end person
end bizdev

write bizdev;
```

Example 1, Simple Schema Definition

Just like a schema definition composed using XML, our schema definition begins with the word "schema." At this point, it consists of a single type, named "person," defined by the word "type" and a semicolon-delimited list of element names. The last line of the code block expresses the schema in document form.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="www.proto.com/schemas/bizdev"
 <xs:complexType name="person" >
  <xs:sequence>
   <xs:element type="xs:string"  name="fname" ></xs:element>
   <xs:element type="xs:string"  name="lname" ></xs:element>
   <xs:element type="xs:string"  name="mid" ></xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

That's it. In no more time than it would take to sketch the schema on a white board we've expressed it in code and XML. It can now be used to define service interfaces, validate documents, and create document instances. The schema immediately becomes URL-accessible, so other participants in the design process can utilize it as well.

As simple as this may seem, there is an even easier way to define our bizdev "person." Tectonic maintains a default schema used to house all constructs not placed in named schemas.

```
type person
  fname;
  lname;
  mid;
end person

write person;
```

Example 2, Type Definition in Default Schema

Output from example 2 is as follows.

```
    <xs:complexType name="person" >
     <xs:sequence>
      <xs:element type="xs:string"  name="fname" ></xs:element>
      <xs:element type="xs:string"  name="lname" ></xs:element>
      <xs:element type="xs:string"  name="mid" ></xs:element>
     </xs:sequence>
    </xs:complexType>
```

Because XML-Schema conventions and constructs are a fundamental part of Tectonic, making use of them is straightforward. When we define a schema, element, or type, we're not generating an OOP class that represents that construct; we're defining the construct itself. There's no autogenerated class named "person" lurking on the machine after this code executes. Tectonic is not an object-oriented wrapper around some service-oriented functionality, it is truly service oriented, from the ground up.

By default, elements in a type definition are considered strings, but Tectonic supports the whole gamut of possibilities defined by XML-Schema. Elements may be constrained using any of the built-in XML-Schema data types, or any user-defined type (complex or simple). An element declaration may even consist of a reference to a top-level element defined elsewhere, just as is the case with XML-Schema.

Getting back to our hypothetical brainstorming session, let's assume some of the participants want to enhance "person" by applying somewhat more realistic constraints and adding a few more members.

```
 type person
  xs:normalizedString fname;
  xs:normalizedString lname;
  xs:normalizedString mid;
  xs:unsignedByte age;
  xs:integer emp-id-num;
 end person

 write person;
```

Example 3, Enhanced Type Definition

Example 3 output:

```
<xs:complexType name="person" >
 <xs:sequence>
  <xs:element type="xs:normalizedString"  name="fname" ></xs:element>
  <xs:element type="xs:normalizedString"  name="lname" ></xs:element>
  <xs:element type="xs:normalizedString"  name="mid" ></xs:element>
  <xs:element type="xs:unsignedByte"  name="age" ></xs:element>
  <xs:element type="xs:integer"  name="emp-id-num" ></xs:element>
 </xs:sequence>
</xs:complexType>
```

So far, all of our “person” member elements use the default properties, meaning they occur one and only once, have no value space constraints beyond those specified for their type, and so on. What if a data architect wants to enhance “person” by including optional and multiply occurring members? Tectonic supports this using a slightly more verbose declaration form.

```
type person
  xs:normalizedString fname;
  xs:normalizedString lname;
  xs:normalizedString mid;
  elementd age type xs:unsignedByte; minOccurs 0; end age;
  xs:integer emp-id-num;
  elementd affiliation
   type xs:normalizedString;
   maxOccurs 3;
   maxLength 60;
  end affiliation
 end person

 write person;
```

Example 4, Verbose Declaration

Example 4 output:

```
<xs:complexType name="person" >
 <xs:sequence>
  <xs:element type="xs:normalizedString"  name="fname" ></xs:element>
  <xs:element type="xs:normalizedString"  name="lname" ></xs:element>
  <xs:element type="xs:normalizedString"  name="mid" ></xs:element>
  <xs:element type="xs:unsignedByte"  name="age" ></xs:element>
  <xs:element type="xs:integer"  name="emp-id-num" ></xs:element>
 </xs:sequence>
</xs:complexType>
```

At this point, it becomes interesting to take a look at an instance of the type. In Tectonic, this is achieved using the keyword “instance.”

```
        pi = instance person;
        write(pi!);
```

Example 5, Type Instancing

The character ‘!’ is known as the elaboration operator, or “bang” for short (the expression pi! is, for example, pronounced “pi bang”), and causes a variable’s structured content to be expressed as a string. Output is as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<person xmlns="http://www.openuri.org/" >
       <fname>string</fname>
       <lname>string</lname>
       <mid>string</mid>
       <age>1</age>
       <emp-id-num>1</emp-id-num>
       <affiliation>string</affiliation>
</person>
```

Viewing type instances is a useful part of brainstorming and design exercises in general and Delve usage in particular. Because session participants can generate instances simply by entering (or pasting in) a couple lines of code, each team member can do so personally and at their leisure. This makes it easy for individuals to experiment with different types and formats, bringing interesting or useful additions/revisions to the group's attention.

The (often simplistic) schemas produced early on can serve as valuable artifacts in later discussions and exercises. To a large extent, this is due to the reusable nature of XML-Schema constructs. For example, once defined, a type or element declaration can be used as the basis for deriving additional types, and can serve in declaring a member of newly created type. Delve drives the collaborative process by fully supporting these aspects of XML-Schema.

Let us suppose that some time after the brainstorming session that produced examples four and five an analyst wishes to create a type representing a business development manager. He can do so by extending the "person" type.

```
type bd-manager from person
 @man-id;
 location;
 elementd report type person; minOccurs 0; maxOccurs 20; end report
 attribute title
  type xs:normalizedString;
  default "business development manager";
 end title
end bd-manager

write bd-manager;
```

Example 6, Type Derivation

In example six, we see two different styles of attribute declaration, one using the concise form

```
       @man-id;
```

and another using a more detailed form

```
        attribute title
         type xs:normalizedString;
         default "business development manager";
        end title
```

that allows for explicit attribute declaration.

Example six output:

```
<xs:complexType name="bd-manager" >
 <xs:complexContent>
  <xs:extension base="person" >
   <xs:sequence>
    <xs:element type="xs:string"  name="location" ></xs:element>
    <xs:element type="person"  minOccurs="0"  maxOccurs="20"
     name="report" >
    </xs:element>
   </xs:sequence>
  <xs:attribute type="xs:string"  name="man-id" ></xs:attribute>
  <xs:attribute type="xs:normalizedString"
   default="business development manager"  name="title">
  </xs:attribute>
 </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Authoring Services

Once a round or two of data model refinements have taken place, the next logical step is generally construction of a rudimentary service prototype. This can often be accomplished in the space of a few minutes.

```
service BDMaint
 operation GetBDManager
  in(placeholder);
  out bdm format bd-manager;

  bdm = instance bd-manager;
 end GetBDManager
end BDMaint
```

Example 7, BDMaint Service, Iteration One

Example seven provides the first cut of a service to be used for maintaining a collection of business development manager profiles. The service "BDMaint," starts out with a single operation "GetBDManager," to be used for obtaining information pertaining to a particular manager. The statement

```
in(placeholder);
```

declares the identity of the operations input message, set to the unused identifier "placeholder" since we don't intend to do anything with the request message just yet. By using the statement

```
out bdm format bd-manager;
```

to declare the output message, we indicate that it should be formatted using the "bd-manager" type defined in example six. A single line of code…

```
        bdm = instance bd-manager;
```

… defines the operation's behavior, setting the output message content to a "blank" instance of "bd-manager."

These seven lines of code are sufficient to completely define the service. There's no framework data or code files linked or modified, no autogenerated source files, no manifests or descriptors or other deployment artifacts, no dependencies of any kind—nothing but the small, self-contained block of code shown in examples six and seven. This code block could be IMed from one team member, who could then paste it directly into a blank Delve code page and immediately execute it.

Services defined in Tectonic are true SOAP Web services, and can be remotely invoked in the same manner as any other service, but they may also be invoked directly, in the same code blocks in which they are defined.

```
        invoke BDMaint/GetBDManager response mgr;
        write("\n\nManager:\n------------------\n");
        write(mgr!);
```

Example 8, Local Service Invocation

In example eight, BDMaint is invoked, with the response message content immediately displayed (as shown below).

```
Manager:
------------------
<?xml version="1.0" encoding="utf-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Body>
  <bd-manager title="business development manager"
    man-id="string" xmlns="http://www.openuri.org/" >
   <fname>string</fname>
   <lname>string</lname>
   <mid>string</mid>
   <age>1</age>
   <emp-id-num>1</emp-id-num>
   <affiliation>string</affiliation>
   <location>string</location>
   <report>
    <fname>string</fname>
    <lname>string</lname>
    <mid>string</mid>
    <age>1</age>
    <emp-id-num>1</emp-id-num>
    <affiliation>string</affiliation>
   </report>
  </bd-manager>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The operation "GetBDManager" is functional but simplistic. There are two obvious steps that should be taken to make its behavior more realistic: 1) have it accept an input parameter, something upon which to select a specific manager representation, and 2) fill in the response message with distinct values, overriding the defaults produced by instancing.

```
service BDMaint
 operation GetBDManager
  in(man-id-to-match);
  out bdm format bd-manager;

  bdm = instance bd-manager;
  bdm/fname = "Harry";
  bdm/lname = "Bendis";
  bdm/mid = "K";
  bdm/age = 42;
  bdm/affiliation = "Financial institutions, like UBank";
  bdm/@man-id = man-id-to-match;
  bdm/location = "Beaverton, OR";
  bdm/@title = "Business Development Principle";
  bdm/report/fname = "Alice";
  bdm/report/lname = "Campbell";
  bdm/report/mid = "F";
  bdm/report/age = 23;
  bdm/report/affiliation = "UBank";
 end GetBDManager
end BDMaint

invoke BDMaint/GetBDManager(A2334) response mgr;
write("\n\nManager:\n------------------\n");
write(mgr!);
write("\n\nfull name is " + mgr/Body/bd-manager/fname/text()
 + " " + mgr/Body/bd-manager/mid/text() + " " +
 mgr/Body/bd-manager/lname/text());
```

Example 9, Local Service Invocation

The revised service code shown in example nine accomplishes both of these goals. A manager ID is passed in; ostensibly to be used in obtaining the profile the manager assigned that ID (though we sill don’t actually make use of the input message). In addition, all of the “bd-manager” instance’s members are assigned values (note the XPath-style syntax used in the assignment statement lvals).

Development team members may immediately begin utilizing the newly authored service (no explicit deployment is necessary) and may do so using exactly the same line of code used to invoke the service locally (after registering the WSDL URL with Delve). While the behavior of the service is still fairly trivial, the mere fact that it is up and running will allow those who wish to create prototye composite services that incorporate BDMaint to begin doing so. Making a BDMaint prototype available quickly means that it is possible for development of service producers and consumers to proceed in parallel—one of the keys to executing a development process that is both collaborative and iterative.

Another activity taking place in parallel with other efforts is continued refinement of the data model and associated prototypical schema members. The types defined in examples four and six, composed in our hypothetical early-stage brainstorming session, are just a starting point. Traditional software development tools behave poorly in this scenario, often requiring that autogenerated code modules be rebuilt from scratch (and thereby making a bit of a mockery of the “loose coupling” concept), but Delve, due to its inherently service oriented nature, minimizes the impact of such changes.

Let’s take a look at the impact of changing the “person” type defined in example four. We’ll add two new elements; a social security number and a business phone number.

```
type person
  xs:normalizedString fname;
  xs:normalizedString lname;
  xs:normalizedString mid;
  elementd age type xs:unsignedByte; minOccurs 0; end age;
  xs:integer emp-id-num;
  elementd affiliation
   type xs:normalizedString;
   maxOccurs 3;
   maxLength 60;
  end affiliation
  xs:normalizedString ssn;
  xs:normalizedString biz-phone-num;
 end person
```

Example 10, Revised Person Type

Making this change does not precipitate modifications elsewhere in the code. Both the service invocation and the subsequent write statements can function unchanged. The service itself will automatically be refreshed when the code defining it is executed. In addition, remote Delve clients making use of the service via SOAP/HTTP will not require modification (unless, of course, they need to make use of the new fields). Even more disruptive schema modifications such as field removal, name changes, and structural alterations can often be made without creating a ripple effect across either the service consumers or service producers that utilize that schema.

Generating Sample Data

Prototyping exercises are often well served by the ability to readily compose significant amounts of sample data that is representative of what is expected in a production environment. Tectonic includes a feature intended to serve this purpose in the form of the “executable element,” a functional entity dedicated to the generation of structured data.

```
element(fn, ln, mi, ag, eid, af, loc, mid, tl) bd-man-gen
 tagname "bd-manager";
 elementi fname tnwrite(fn); end fname
 elementi lname tnwrite(ln); end lname
 elementi mid tnwrite(mi); end mid
 elementi age tnwrite(ag); end age
 elementi emp-id-num tnwrite(eid); end emp-id-num
 elementi affiliation tnwrite(af); end affiliation
 elementi location tnwrite(loc); end location
 attribute man-id = mid;
 attribute title = tl;
end bd-man-gen
```

Example 11, Manager Generator

The executable element “bd-man-gen,” shown in example eleven above, is used to create a bizdev manager profile instance. Doing so provides more flexibility than using an “instance” statement because an executable element can utilize parameters and incorporate arbitrarily complex logic.

Let’s suppose an architect wants to expand the BDMaint service to include an operation used to obtain an entire list of business development managers.

```
        operation GetAllBDManagers
          in(include-reports);
          out bdms format bd-mgr-collection;

          bdms = bd-men();
         end GetAllBDManagers
```

The new operation sets its output format to “bd-mgr-collection” a newly defined type.

```
        type bd-mgr-collection
         elementd manager type bd-manager; maxOccurs 100; end
        manager
        end bd-mgr-collection
```

The operation’s body consists of a single line of code. This statement sets the output message content to the result of calling an executable element that serves to generate a manager collection instance.

```
element bd-men
 tagname "bd-mgr-collection";

 bd-man-gen("Sally", "Denton", "V", 51, 8892,
 "Aerospace, Southern CA", "Los Angeles", "C3323",
 "Senior Business Development Manager");
 bd-man-gen("Daniel", "Broward", "T", 33, 2332,
 "IBM Global Services, Cisco", "Los Angeles", "N23",
 "Business Development Manager");
 bd-man-gen("Alice", "Grabel", "Y", 43, 5670,
 "Various startups, Boston", "Boston", "N67",
 "Business Development Manager");
end bd-men
```

Example 12, Manager Collection Generator

Within “bd-men,” calls to “bd-man-gen” create child elements representing individual bizdev managers.

Using the statement

```
invoke BDMaint/GetAllBDManagers(false) response mgrs;
```

to invoke the newly written operation produces the following result.

```
<?xml version="1.0" encoding="utf-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 <SOAP-ENV:Body>
  <bd-mgr-collection>
   <bd-manager title="Senior Business Development Manager"
     man-id="C3323" >
    <fname>Sally</fname>
    <lname>Denton</lname>
    <mid>V</mid>
    <age>51</age>
    <emp-id-num>8892</emp-id-num>
    <affiliation>Aerospace, Southern CA</affiliation>
    <location>Los Angeles</location>
   </bd-manager>
   <bd-manager title="Business Development Manager"  man-id="N23" >
    <fname>Daniel</fname>
    <lname>Broward</lname>
    <mid>T</mid>
    <age>33</age>
    <emp-id-num>2332</emp-id-num>
    <affiliation>IBM Global Services, Cisco</affiliation>
    <location>Los Angeles</location>
   </bd-manager>
   <bd-manager title="Business Development Manager"  man-id="N67" >
    <fname>Alice</fname>
    <lname>Grabel</lname>
    <mid>Y</mid>
    <age>43</age>
    <emp-id-num>5670</emp-id-num>
    <affiliation>Various startups, Boston</affiliation>
    <location>Boston</location>
   </bd-manager>
  </bd-mgr-collection>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The sample data, and a SOAP operation that accesses it, become immediately available to the analyst's colleagues. They may even invoke it using exactly the same line of code used by the analyst for his internal test. Furthermore, the "bd-men" executable element can readily be upgraded to use techniques such as random number generation to create sample data in a more dynamic fashion.

The examples in this section help demonstrate what we mean by "Immersive SOA." Instead of thinking and planning based on factors such as the structure of existing databases and file formats, considerations which often drive early-stage efforts, requirements and architecture are framed in terms of services, interfaces, and structured data. This enables team members to focus on domain objects and behaviors in the manner required by RUP and other modern methodologies.

Persistence Considerations

Having a stock of canned data to access through our prototype is all well and good, but it's often the case that different team members wish to add and retrieve their own sample data, making it available to other team members as the development process progresses. Enhancing the BDMaint service such that it is able to add and retrieve specific “person” data elements provides this sort of capability.

We’ll add two operations to BDMaint, one which is used to add a single person’s profile, and one which use to retrieve a profile by social security number. The first step is to add some additional schema components. For convenience sake, we’ll wrap our “person” data in a structure that also provides an indication of whether or not the required profile was available. A couple of top-level element declarations provide a convenient means of referring to elements of the required types.

```
elementd person-el type person; minOccurs 0; end person-el

type GetPersonResult
 xs:boolean success;
 elementd ref person-el;
end GetPersonResult

elementd person-result type GetPersonResult; end person-result;
```

Next, we need to add the new service code to BDMaint. The keywords “save” and “retrieve” are used to access Delve’s built-in persistence store and “exists” is used to check the store for a particular key.

```
operation AddPeep
  in to-add format person-el;
  out(true-if-not-redundant);

  write("begin add");
  write(to-add!);
  true-if-not-redundant = true;

  if (to-add//ssn/text() exists) then
   write("\nalready there\n");
   true-if-not-redundant = false;
  else then
   write("\nnot there: adding\n");
   save to-add//person-el, to-add//ssn/text();
  end then
 end AddPeep

 operation GetPeep
  in(ssn-to-get);
  out peep-got format person-result;

  peep-got = instance person-result;
  write("\nlooking for: " + ssn-to-get + "\n");
  if (ssn-to-get exists) then
   retrieve person-from-store, ssn-to-get;
   peep-got/success = true;
   peep-got/propfile = person-from-store;
  else then
   peep-got/success = false;
  end then
 end GetPeep
```

Finally, we'll add a block of code that tests both of the newly added operations, named "AddPeep" and "GetPeep."

```
p = instance person-el;
p/fname = "Sam";
p/lname = "Bungie";
p/emp-id-num = 557;
p/ssn= "774-58-4231";
p/biz-phone-num = "(650)555-1212";
p/affiliation = "unknown";
p/age = 22;

invoke BDMaint/AddPeep p response sr;
write("added " + p/fname);

if (lnot sr) then
 write(" was already there");
else then
 write(" was new record");
end then
write("\n\nretrieving...\n\n");

invoke BDMaint/GetPeep("774-58-4231") response q;
if (q//success/boolean()) then
 write("success:\n");
 write(q!);
else then
 write("could not find profile");
end then
```

Example 13, Persistence Enhancements

The new operation implementations may look like pseudocode, but they're entirely functional, and will allow team members to add and retrieve person profiles to and from a common repository at will, building up a common store of sample data to be used in building out the prototype.

Because the persistence store is not format-sensitive, persistence code can generally ignore changes to schema components.   Suppose, for instance, that data modelers decide to drop the "biz-phone-num" element from the "person" type, replacing it with a child element containing a variety of contact information. No changes would be required to the BDMaint service whatsoever—it could continue to function unchanged. This is in stark contrast to the manner in which a solution based on say, a relational database, would be affected.

The fact that Tectonic programs often take on the appearance of pseudocode can be advantageous. It means that the Tectonic programs can themselves constitute useful requirements and design deliverables. A developer taking part in the construction phase of a project (in RUP parlance) can look at a line of Tectonic code reading "save to-add//person-el, to-add/ssn/text()" and know that she needs to insert a record into a database such that the destination table is used to store "person" records and the "ssn" row of that table is indexed.