Chapter 4. Sharing Data through Web Services

Data sharing would seem to be a simple task. Agencies have been making their data publicly available through the Internet since the 1980s. The World Wide Web in its early form can be thought of as a big, read-only file sharing network. High-speed networks allow gigabytes of data to be moved from one place to another in very little time, and the cost of these networks keeps decreasing. So why is sharing data still a problem?

In the buildout analysis, a host of data sources are used. In the case of zoning, the primary challenge was translating each town's zoning categories into matching categories. With land use, the big problem was finding and acquiring the most up-to-date data sources, systematizing their inclusion into the analysis. The latest data is usually the most disaggregated, and in the hands of the smallest organizations with the least incentive to participate in a larger system. In this case these are the developers who are building the newest residential subdivisions.

Sharing data with government, and supporting planning support systems are not the primary mission of developers, yet highly detailed data sets are critical in an urban information infrastructure. They are usually created and maintained by small, local organizations, so there must be a mechanism for data publishing that conforms to their level of technological sophistication. However, at the other end of the spectrum the system must be sophisticated enough to support complex analyses. This chapter lays out a Web services strategy for meeting these seemingly conflicting goals.

WSDL

We start with a very simple example, because an important design element is the ability to offer simple solutions for simple requirements. In this case, the requirement is to enable an organization to publish spatial data as easily as they publish Web pages. The most common spatial data format is the ESRI Shapefile. Shapefiles are like Adobe PDF files in that the data format is public and free to use, and the files are small and easily emailed, making the Shapefile the *de facto* standard in the GIS world. Instead of simply placing these files on a Web site, publishing them through a Web service interface allows the data to be more tightly integrated into information processing systems, hopefully in a more fully automated manner.

First of all, it is important to emphasize the similarities between a Web site and a Web service. In the strictest sense, any part of a Web site can be a Web service if it is described formally. For example, a Web page is a text file containing data in Hypertext Markup Language (HTML). It is accessed using the Hypertext Transfer Protocol (HTTP) by sending a GET request to a particular Universal Resource Locator (URL). If the previous two sentences are written formally in a particular dialect of XML called Web Services Description Language (WSDL), the Web page becomes a Web service.

Code Listing 4-1 presents a simple WSDL file that serves to publish a Shapefile as a Web service. A WSDL file has four sections, `service`, `binding`, `interface`, and `types`. The service section tells a user what Web address to access in order to invoke the Web service. The interface sections tells the user what commands the service understands, and the types section describes the format of these commands and the responses that may be returned. The binding section has technical details relating to how the commands described in the interface section must be expressed in a particular language. A service could have one interface and many bindings, meaning that the same command can be expressed in many different languages. Another important concept is that the WSDL expression of a service is an abstraction. There could be other Shapefiles on this Web site, and they may or may not be "published." There could also be other services that "publish" the same data, but use a different WSDL file—meaning that the data is published in a different way to a different audience.

In this way the Web service can be crafted to meet the exact requirements of an organization. This can be a useful concept if we think of the WSDL file as bridging the gap between organizational and technical concerns. In formally describing the data sources, and the means of accessing them in a highly structured manner, WSDL becomes not only a technical solution to data sharing, but a *contract* between the data provider and the data user. This is the contract that trading partners require to ensure a stable relationship in regards to information exchange.

Basic Data Sharing: one Shapefile

In order to publish a Shapefile as a Web service, three things must be put on a Web server:

1. The data files being published.
2. A WSDL file describing certain generic aspects of a Shapefile.
3. An XML file describing the specific Shapefile being published.

The generic aspects of a Shapefile are described in the `types` section of Code Listing

4-1. We see there an XML Schema element called `ShapefileWriter`, named so to distinguish between a service message that outputs, or writes, a Shapefile, and one that ingests, or reads one. Note the XML attribute `srsName`. All spatial data has a particular spatial reference system (SRS)—a way of referencing locations on the earth. Cartographers have hundreds of different ways of doing this, based on tradeoffs between accuracy, scale and other considerations. These different systems have all been given a name, and that is what would be stored in the `srsName` attribute. of the `ShapefileWriter` element. Shapefiles store their data in three files having .shp, .dbf, and .shx suffixes. The locations of these files are specified in the `ShpFile`, `DbfFile`, and `ShxFile` elements as URLs.

The interface, binding, and service sections combine to say that the Web request, `http://www.city.us/wetlandsShapefile.xml`, will be answered with an XML file conforming to the XML Schema defined by the `ShapefileWriter` element. In this case, a possible response is shown in Code Listing 4-2. A small, unsophisticated agency could put the two XML files on their Web site along with the three Shapefile components, and consider the data published by giving interested parties the URL to the WSDL file. This is the bare minimum required to participate in the collaborative framework envisioned in this paper.

Code Listing 4-1: WSDL file for Shapefile publishing

```
<definitions name="DataPublishing”>

<types>
   <xs:schema targetNamespace="http://web.mit.edu/pamml.wsdl">
      <xs:element name=”ShapefileWriter” type=”ShapefileWriterType”/>
      <xs:complexType name=”ShapefileWriterType”>
         <xs:sequence>
            <xs:element name=”ShpFile” type=”xs:anyURI”/>
            <xs:element name=”DbfFile” type=”xs:anyURI”/>
            <xs:element name=”ShxFile” type=”xs:anyURI”/>
         </xs:sequence>
         <xs:attribute name=”srsName” type=”xs:string”/>
      </xs:complexType>
      <xs:element name=”NullMessage” nillable=”true”/>
   </xs:schema>
</types>

<interface name="PublishDataInterface">
   <operation name="GetData" pattern="http://www.w3.org/2003/11/wsdl/in-out">
      <input message="tns:NullMessage"/>
      <output message="tns:ShapefileWriter"/>
   </operation>
</interface>

<binding name="PublishDataBinding" type="tns:PublishDataInterface">
   <http:binding verb=”GET”/>
   <operation name=”HTTPBindingGetDataOperation>
      <http:operation location=”/wetlandsShapefile.xml”/>
      <input>
         <http:urlReplacement/>
      </input>
      <output>
         <mime:content type=”text/xml”/>
      </output>
   </operation>
</binding>

<service name="PublishDataService">
   <documentation>Geospatial data accessible from this server</documentation>
   <endpoint name="DataServiceURL" binding="tns:PublishDataBinding">
      <http:address location="http://www.city.us"/>
   </endpoint>
</service>

</definitions>
```

Code Listing 4-2: XML instance document for Shapefile publishing

```
<ShapefileWriter srsName=”EPSG:26986”>
   <ShpFile dataFile=”http://www.city.us/wetlands.shp”/>
   <DbfFile dataFile=”http://www.city.us/wetlands.dbf”/>
   <ShxFile dataFile=”http://www.city.us/wetlands.shx”/>
</ShapefileWriter>
```

A full explanation of the WSDL specification is beyond the scope of this work. However it is important to note a few characteristics of this approach. A WSDL file is quite complex, and a small organization would probably need to contract out for its development. But still it is only a text file, so no additional software or hardware, beyond what is required to publish Web pages, is needed to participate in what will be shown to be a sophisticated system. This point is so important because it matches so well the way organizations function. Most organizations—even small non-profits—are able to initiate large, complex projects because it is at the beginning when the project's champions are still in place and there is usually some commitment of resources. Problems usually arise over time, or after the project is "officially" over (meaning no longer explicitly funded), when time, maintenance and upkeep must be incorporated into a general operational cost structure. With finite resources and turnover in leadership, old projects tend to lose funding and time commitments and cease to operate if their upkeep requires any extraordinary effort. In publishing this Web service we have a complicated project initiation stage, where the data and XML files must be created and posted on the Web site, but a simple maintenance stage that only requires the upkeep of a Web server, which is probably critical to other organizational initiatives as well.

Professional Data Sharing

The previous section focused on the requirements of small agencies whose technology infrastructure was limited to a Web server. This is a sensible baseline technology, considering that even millions of individuals in the U.S. have their own Web site. The implementation strategies outlined above do not meet the needs of professionals, however. GIS agencies, planners, assessors, and the like have broader requirements, and a more sophisticated technology infrastructure, than a basic Web server. In this section we address the needs of these more traditional spatial data providers. Generally, these are municipal, regional and state agencies that publish numerous data sets, often in multiple formats. Sometimes these data sets do not reside on disk, but in a database, or are generated on request. Another important characteristic of these kinds of organizations is that they often update their data, so their customers must be made aware of this fact and consider the update event in managing their own business processes. Finally, these agencies are concerned about their data's provenance. Making sure their users know when a data set was created, last updated, or its level of accuracy are concerns that have significant organizational, if not legal, ramifications.

Metadata

Information about a data set is generally referred to as *metadata*. The subject of what should be recorded in metadata is an active field of inquiry. In the U.S., the Federal Geographic Data Committee (FGDC) has for over a decade championed the FGDC Metadata Standard. Internationally, the International Standards Organization (ISO) has issued a standard called Geographic Information — Metadata, which is commonly referred to by its document identification number, ISO19115. What these organizations are trying to do is to capture, in broad terms, the general characteristics of geographic information so that potential users can search for information relevant to their task, and quickly decide whether that information meets their needs. This involves capturing spatial metadata, such as the geographic extent of a data set, attribute metadata, such as the names and data types of attributes, and administrative metadata, such as the responsible agency, date of creation, and update frequency.

Metadata is not the focus of this research, but it certainly plays a complementary role. The latest metadata standardization efforts of organizations like the FGDC, ISO, and OpenGIS rely on XML technologies, so the XML-focused work presented here can easily incorporate metadata by simply using XML's built-in extensibility mechanisms. Code Listing 4-3 supplements the XML definition of `ShapefileWriter` from Code Listing 4-1 to support metadata. A new element, `Metadata`, is added to the object, and it is defined in a very general way in the `MetadataType` object. This is simply an object that can have any XML content in it, allowing an organization to incorporate their metadata efforts with their distributed planning support systems work.

Code Listing 4-3: Adding metadata to data

```
<xs:element name="ShapefileWriter" type="ShapefileWriterType"/>

<xs:complexType name="ShapefileWriterType">
   <xs:sequence>
      <xs:element name="Metadata" type="MetadataType"/>
      <xs:element name="ShpFile" type="xs:anyURI"/>
      <xs:element name="DbfFile" type="xs:anyURI"/>
      <xs:element name="ShxFile" type="xs:anyURI"/>
   </xs:sequence>
   <xs:attribute name="srsName" type="xs:string"/>
</xs:complexType>

<xs:complexType name="MetadataType">
   <xs:sequence>
      <xs:element name="Publisher" type="xs:string"/>
      <xs:element name="Date" type="xs:date"/>
      <xs:any minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>
```

Object inheritance, and sharing multiple files through a single service

The number of common spatial data formats seems endless. Organizations that publish spatial data often make it available in multiple formats, to support the various software environments of their users. There are a number of file-based formats that are similar to Shapefiles in that they are defined by the locations of their component files. Another big class of spatial data format is the spatial relational database. This includes Oracle Spatial, IBM DB2, PostGIS, and MySQL. Accessing data in these formats generally involves making a database connection, which requires some authentication and network location information. An example of how a PostGIS data source could be modeled is shown in Code Listing 4-4.

Code Listing 4-4: Accessing spatial data in PostGIS

```
<xs:element name="PostGISWriter" type="pamml:PostGISWriterType"/>

<xs:complexType name="PostGISWriterType">
   <xs:sequence>
      <xs:element name="User" type="xs:string"/>
      <xs:element name="Passphrase" type="pamml:PassphraseType"/>
      <xs:element name="Host" type="xs:anyURI"/>
      <xs:element name="Port" type="xs:int"/>
      <xs:element name="Driver" type="xs:string"/>
   </xs:sequence>
   <xs:attribute name="srsName" type="xs:string"/>
</xs:complexType>
```

Notice that, like *`ShapefileWriter`*, *`PostGISWriter`* **has the** *`srsName`* **attribute. It would also have the** *`Metadata`* **element, if fully defined, but instead of repeatedly defining objects that are common to many other objects, XML allows objects to *inherit* the characteristics of another. What we would like to say is that every data model in our system may have metadata, and must have a spatial reference system definition. Code Listing 4-5 expresses this.**

Notice that *Metadata* is defined in the *ModelType* object. Here we introduce the concept that some data models might not represent spatial data. Every model may have metadata, but those that represent spatial data also have a spatial reference system (the *srsName* attribute modeled in the *GeoData* object). The concept of inheritance will be used extensively in this work. It not only provides clarity to an information model, but offers practical benefits in system implementations.

Code Listing 4-5: An object-oriented model of spatial data

```
<xs:element name="Model" type="ModelType"/>
<xs:complexType name="ModelType">
   <xs:sequence>
      <xs:element ref="Metadata" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="GeoDataType">
   <xs:complexContent>
      <xs:extension base="ModelType">
         <xs:attribute name="srsName" type="xs:string" use="required"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:element name="ShapefileWriter" type="ShapefileWriterType"/>
<xs:complexType name="ShapefileWriterType">
    <xs:complexContent>
      <xs:extension base="GeoDataType">
         <xs:sequence>
            <xs:element name="ShpFile" type="xs:anyURI"/>
            <xs:element name="DbfFile" type="xs:anyURI"/>
            <xs:element name="ShxFile" type="xs:anyURI"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:element name="PostGISWriter" type="PostGISWriterType"/>
<xs:complexType name="PostGISWriterType">
    <xs:complexContent>
      <xs:extension base="GeoDataType">
            <xs:sequence>
            <xs:element name="User" type="xs:string"/>
            <xs:element name="Passphrase" type="PassphraseType"/>
            <xs:element name="Host" type="xs:anyURI"/>
            <xs:element name="Port" type="xs:int"/>
            <xs:element name="Driver" type="xs:string"/>
         </xs:sequence>
      </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:element name="GeoDataModels" type="GeoDataModelsType"/>
<xs:complexType name="GeoDataModelsType">
   <xs:sequence>
      <xs:element name="GeoDataModel" type="GeoDataType" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>
```

Developing a better object-oriented data model also provides flexibility when we look at publishing more complex data services. In theory, multiple data sets could be published using the strategy recommended above (for a small agency publishing one Shapefile). In practice, however, this system could be difficult to maintain for the publisher, because it requires each data set to have its own WSDL file, and since they will all be very similar, making a small change, such as updating the agency's phone number, requires changes to many files. The way organizations have traditionally published data has been to advertise one Web site with data download functionality. Perhaps this "data warehouse" paradigm is less compelling in a Web services framework, and it is better to use the one data set per service concept, but that is a debate for another time. Here we simply show that the data warehouse idea can be supported.

Code Listing 4-6 describes a Web service that publishes multiple data sets in multiple formats. The main difference between this service and the basic one is that there must be a "conversation" between the client and the service to determine which data set to give the client and in what format. In the most general sense, this is a search task. The client is searching for data of a particular type, and will be able to identify it by some characteristic, like its name, subject matter or geographic region. Searching and cataloging will probably only be done well by specialized services. This is the case with the Web in general. Individual Web sites used to all have their own internal search engine, but nowadays most sites let Google handle search.

While a handful of the largest spatial data libraries may implement their own search and cataloging functionality, most will only need to publish a short list of data sets in their holdings. This is best accomplished by creating an object that lists spatial data models. The *GeoDataModelsType* object shown in Code Listing 4-5 fills this role. Code Listing 4-6 shows how that list of available data sources is accessed by making a

GetDataListing request to the service (in this example, the service is invoked using a SOAP binding). From this list, the user can choose the data set they desire. The final problem to solve is how services uniquely identify data sets. The most common way of doing this is to give every object a unique ID. While this requires some mechanism to ensure that the ID is unique, in the Internet space this is usually made easier by the ability of an organization to prefix the identification token with their Internet domain name, avoiding cross-organization naming problems. In order to employ this strategy a new attribute must be added to all of our model objects, so we add an *id* attribute to the *ModelType* object. This allows the requesting client to get at the *id* attribute of the model, which is needed to make a full model request using the *GetDataSourceByID* message of the *GetDataSource* operation.

Code Listing 4-6: Service description (WSDL) for data publishing

```
<definitions name="DataPublishing”>
<xs:import namespace="http://web.mit.edu/pamml"
          location="http://web.mit.edu/pamml.xsd"/>
<types>
   <xs:schema targetNamespace="http://web.mit.edu/pamml.wsdl">
      <!–- insert elements from Code Listing 4-5 -->
      <xs:element name="GetDataListing" nillable=”true”/>
      <xs:element name="GetDataSourceByID" type="xs:string"/>
   </xs:schema>
</types>

<interface name="PublishDataInterface">
   <operation name="QueryData" pattern="http://www.w3.org/2003/11/wsdl/in-out">
      <input message="tns:GetDataListing "/>
      <output message="tns:GeoDataModels"/>
   </operation>
   <operation name="GetDataSource" pattern="http://www.w3.org/2003/11/wsdl/in-out">
      <input message="tns:GetDataSourceByID"/>
      <output message="tns:GeoDataModel"/>
   </operation>
</interface>

<binding name="PublishDataSOAPBinding" type="tns:PublishDataInterface">
   <soap:binding style="document"
              transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="QueryService">
      <soap:operation soapAction="http://www.scituate.ma.us/QueryService"/>
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
   </operation>
   <operation name="GetData">
      <soap:operation soapAction="http://www.city.us/DataService"/>
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
   </operation>
</binding>

<service name="PublishDataService">
   <documentation>Geospatial data accessible from this server</documentation>
   <endpoint name="DataServiceURL" binding="tns:PublishDataSOAPBinding">
      <soap:address location="http://www.city.us/DataService"/>
   </endpoint>
</service>

</definitions>
```

Sharing data in multiple formats

In Code Listing 4-6 we did not explicitly define a mechanism for publishing the same data set in multiple formats. We only devised a way to publish multiple data sets. Those data sets *could* represent the same data, but it would be nice to have a way to make this relationship explicit. The concept that an output data source is really one concrete representation of some abstract data object is an important one, though. The unique ID just discussed pertains to one particular concrete instance of the data—a Shapefile, PostGIS source, etc.—not the underlying data model, which should be described aside from its output format. For our data modeling efforts, this means that any object that outputs data should have some internal representation of spatial data, as shown in Code Listing 4-7, where `ShapefileWriter` and `PostGISWriter` now have an internal `GeoDataType` object. If an organization published a data set in Shapefile and PostGIS formats, this internal object could be the same (have the same ID), although the `ShapefileWriter` and `PostGISWriter` objects would have different IDs (and would rightly be semantically different objects). The information modeling tools required to design this structure are readily available in the XML language, making it easy to add this level of inheritance, indirection and nesting.

Code Listing 4-7: Modeling spatial data output

```
<xs:complexType name="ShapefileWriterType">
    <xs:complexContent>
     <xs:extension base="GeoDataType">
       <xs:sequence>
          <xs:element name="ShpFile" type="xs:anyURI"/>
          <xs:element name="DbfFile" type="xs:anyURI"/>
          <xs:element name="ShxFile" type="xs:anyURI"/>
          <xs:element name="DataSource" type="GeoDataType"/>
       </xs:sequence>
     </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:complexType name="PostGISWriterType">
    <xs:complexContent>
     <xs:extension base="GeoDataType">
          <xs:sequence>
          <xs:element name="User" type="xs:string"/>
          <xs:element name="Passphrase" type="PassphraseType"/>
          <xs:element name="Host" type="xs:anyURI"/>
          <xs:element name="Port" type="xs:int"/>
          <xs:element name="Driver" type="xs:string"/>
          <xs:element name="DataSource" type="GeoDataType"/>
       </xs:sequence>
     </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Some practical considerations

In addition to creating a Web services framework, the research agenda included prototyping applications that implement the services (presented in Chapter 7). From this experience, a number of issues emerged that did not arise in the pure data modeling exercise. These do not have a direct significance to any planning problem, but were crucial in designing a language from which applications could be developed. These features must be presented now for the upcoming code examples and graphics to make sense.

Spatial data typing issues

Most important is that the abstract concept of spatial data has little use in application development. GIS software is designed to work primarily with one of two types of spatial data, vector and raster. Furthermore, the overwhelming majority of vector data formats model spatial objects as a set of geometry objects (one of the seven "simple features" defined in the OpenGIS Consortium Abstract Specification) linked to an attribute table. Raster data sets are even simpler, with each cell having only one attribute. The common models for vector and raster data sets are shown in Code Listing 4-8, Code Listing 4-9, and Figure 4-1, along with the rest of the spatial data model hierarchy used in this work.

Efficient design of a data processing application *requires* that the type of data be known beforehand. It also *helps* to know what attributes the data set has, as well as their types. Therefore we include attribute information in the `VectorDataType`'s `AtributeInfo` object. For example, a client may want to access wetlands data in conjunction with a habitat model. One simple application would be to summarize the different types of wetlands present. This would require knowing what data attribute contained the information describing the wetland type, so it is extremely helpful to advertise these features of the data set. This concept is discussed in more detail later. In fact, only the most important modeling concepts are discussed in this text. Many decisions made to facilitate practical implementations are only detailed in the full, working XML Schema in Appendix A.

Code Listing 4-8: The complete spatial data model hierarchy

```
<xs:complexType name="ModelType">
  <xs:sequence>
    <xs:element ref="Metadata" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="GeoDataType">
  <xs:complexContent>
    <xs:extension base="ModelType">
      <xs:attribute name="srsName" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="VectorDataType">
  <xs:complexContent>
    <xs:extension base="GeoDataType">
      <xs:sequence>
        <xs:element ref="AttributeInfo" minOccurs="0">
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="RasterDataType">
  <xs:complexContent>
    <xs:extension base="GeoDataType">
      <xs:attributeGroup ref="rasterAttributes"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="ShapefileWriterType">
   <xs:complexContent>
    <xs:extension base="VectorDataType">
      <xs:sequence>
        <xs:element name="ShpFile" type="xs:anyURI"/>
        <xs:element name="DbfFile" type="xs:anyURI"/>
        <xs:element name="ShxFile" type="xs:anyURI"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="ASCIIIntegerGridReaderType">
  <xs:complexContent>
    <xs:extension base="RasterDataType">
      <xs:sequence>
        <xs:element name="DataFile" type="DataFileCompressable"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

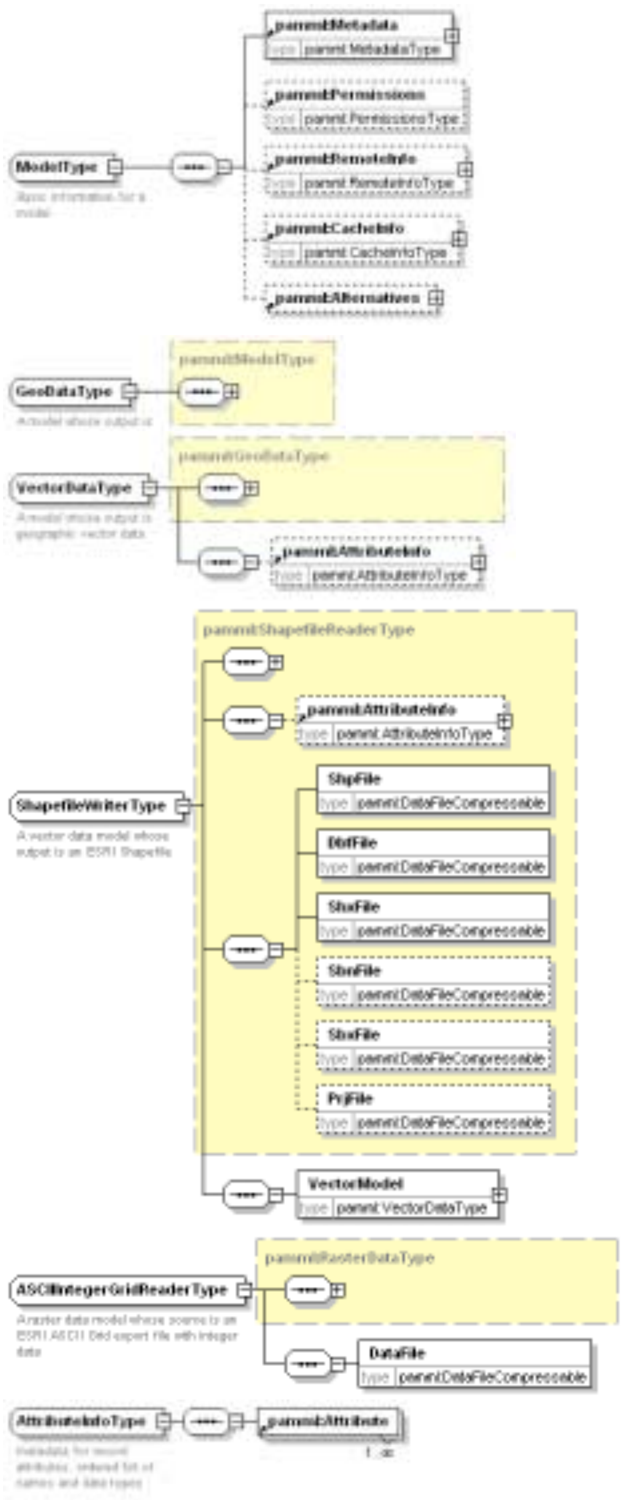
Code Listing 4-9: Spatial and tabular data feature definition

```
<xs:element name="AttributeInfo" type="pamml:AttributeInfoType"/>

<xs:complexType name="AttributeInfoType">
   <xs:sequence>
      <xs:element ref="pamml:Attribute" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>

<xs:element name="Attribute">
   <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="dataType" type="xs:anySimpleType" use="required"/>
      <xs:attribute name="minVal" type="xs:string" use="optional"/>
      <xs:attribute name="maxVal" type="xs:string" use="optional"/>
      <xs:attribute name="query" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

Figure 4-1: Common spatial data objects



ModelType
Basic information for a model
pammt:Metadata
type pammt:MetadataType
pammt:Permissions
type pammt:PermissionsType
pammt:RemoteInfo
type pammt:RemoteInfoType
pammt:CacheInfo
type pammt:CacheInfoType
pammt:Alternatives
GeoDataType
pammt:ModelType
VectorDataType
pammt:GeoDataType
pammt:AttributeInfo
type pammt:AttributeInfoType
ShapefileWriterType
A vector data model whose output is an ESRI Shapefile
pammt:ShapefileReaderType
ShpFile
DbfFile
ShxFile
SbnFile
SbxFile
PrjFile
type pammt:DataFileCompressable
VectorModel
type pammt:VectorDataType
ASCIIIntegerGridReaderType
A raster data model whose source is an ESRI ASCII Grid export file with integer data
pammt:RasterDataType
DataFile
type pammt:DataFileCompressable
AttributeInfoType
pammt:Attribute
1..∞

Performance issues

One of the biggest shortcomings to distributed systems is the tremendous difference in performance between a desktop application using hard drive-bound data, and an Internet-based application. Whether or not this is actually the case, people seem to be uncomfortable with the idea that the data underlying their work is out of their control. They may not articulate their feelings in this way, but it was felt that to have widespread acceptance, a key design feature of this Web service-based framework would be to offer the benefits of both systems. At the simplest level, the language describes information processing in a fully distributed manner. However, there are objects built into the language that provide “hooks” that software developers can use to implement the system in such a way that all data and models are stored locally on the user’s computer. We can still take advantage of the distributed framework, by making sure the software stays synchronized with the original data sources, but users get the performance benefits of using data on their hard drive, and the peace of mind of knowing that no one can arbitrarily cut off their access to the data. This last feature does in fact have a direct planning application, in that one of our target audiences is small, community-based non-profit organizations, who often have a (real or perceived) adversarial relationship with government agencies, and are therefore not likely to adopt a system that relies completely upon a constant level of cooperation with city hall and the state house.

In order to provide users with these benefits, a few additional objects must be added to the language that will only be used by software implementers, not end users. `RemoteInfo`, in Code Listing 4-10 is the construct that provides the language hooks that software can use to implement local caching schemes. Consider that the model being read is potentially a copy whose origin is unknown. The model may have been acquired by a Web search, or someone may have emailed it to you. In that case, you have a file sitting on your computing device (which could be a computer, mobile phone, etc). You know that your computer can not execute this model, so it must have a means of telling you how it can be executed, and this requires semantics describing the original location of the model description (the *ModelLoc* object), and the location of a computer that is able to execute the model (the *ModelRunnerLoc* object). Those two objects make distributed computing more flexible. The next object, *LocalCache*, is the one that enables the local storage of data. Notice that *LocalCache* is itself a Model, which does not need to be of the same type as the original model. This allows the implementing software to, for example, cache a complex spatial operation as a simple Shapefile, while still having the option to re-compute the analysis from the remote source when desired. This example underscores the importance of PAMML's highly decomposable design. The abstraction of a spatial processing operation into a function that outputs a vector data set, combined with the fact that any PAMML operation will output only one data set, creates a very simple basic structure, which greatly facilitates the loose coupling of distributed resources.

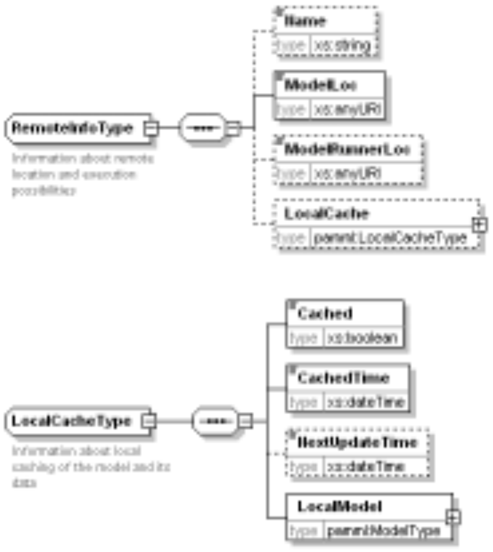
Code Listing 4-10: Objects that make distributed computing perform like desktop computing

```
<xs:element name="RemoteInfo" type="RemoteInfoType"/>

<xs:complexType name="RemoteInfoType">
   <xs:sequence>
      <xs:element name="Name" type="xs:string" minOccurs="0"/>
      <xs:element name="ModelLoc" type="xs:anyURI"/>
      <xs:element name="ModelRunnerLoc" type="xs:anyURI" minOccurs="0"/>
      <xs:element name="LocalCache" type="LocalCacheType" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="LocalCacheType">
   <xs:sequence>
      <xs:element name="Cached" type="xs:boolean"/>
      <xs:element name="CachedTime" type="xs:dateTime"/>
      <xs:element name="NextUpdateTime" type="xs:dateTime" minOccurs="0"/>
      <xs:element name="LocalModel" type="ModelType"/>
   </xs:sequence>
</xs:complexType>
```

Figure 4-2: RemoteInfoType and LocalCacheType object diagrams



This chapter has laid out a strategy for addressing one of the primary causes of high information management costs, the process of moving data sets from producers to users and into analysis systems with a minimum of human intervention. In the past decade or so, we have made great strides in our ability to distribute data efficiently. Most data are stored in electronic format, and content encoding formats are standardized enough so that translation is more of an annoyance than a real barrier to use. What we have not addressed until now is the orchestration of the process to the level of detail where human intervention can be replaced by computer-to-computer negotiation. This not only achieves significant cost reductions through automation—replacing expensive human resources with cheap computing cycles—but also creates the opportunity for new levels of efficiency, and better systems. For example, this architecture permits software to be developed that runs a quick analysis based on locally cached information resources, or a slower, more thorough one that reaches out to remote warehouses to make sure it is using the most up-to-date data. In the next chapter we build upon this methodology, adding analysis to the data sharing framework.