Chapter 5. Web Services for Collaborative Modeling and Decision Making

The data publishing framework presented in the last chapter is one example of a more general strategy, which in computer programming is called the ***adapter design pattern***. When computing systems need to interoperate with each other, they usually only need to know a few things about each other; they do not need to understand each other's entire realm of functionality. Therefore, it often makes sense to create a connection object that ***encapsulates*** a program's functionality into the few key parameters that other systems might be interested in. This connection object is called an adapter. This chapter relies heavily on the concepts of adapters and encapsulation to extend the data sharing framework into a system that integrates that data across systems for analysis, decision support and participation.

Computing design patterns for distributed Web services

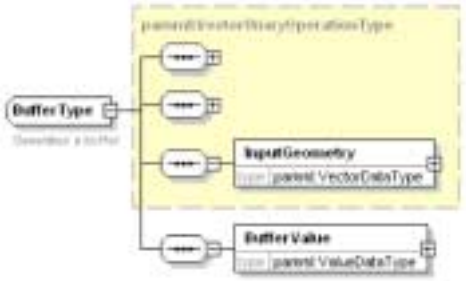
In the previous chapter, data were presented as abstracted, stylized ***models*** of real-world phenomena and processes (Keller 1999). Data and analytic models are often thought of as being different concepts, but this distinction is false. More generally, a model is a simplified description of a complex entity or process. It could represent a number, a spatial data set, or an analysis that predicts population growth. The practical difference between data models and analytic models is that the latter can usually be described algorithmically and therefore be reproduced by computers. One might even

say that what we call data are models for which we either have not yet discovered the algorithm, or algorithmic detail is not relevant to the problem at hand.

One example is rainfall. Say that the federal government wants to modernize the National Weather Service (`http://www.nws.noaa.gov`). A gardener might be interested in having access to a service that told them how much rain is likely to fall, but they would have little use for the meteorological model that underlies the rainfall forecasts. In this case, the gardener only needs the rainfall model to be published as data. A corporate farmer, however, might be very interested in the details of the model, and have the resources to integrate it into an internal production forecast model. But they in turn probably would not need access to the level of detail that a NOAA scientist would want whose job was to re-calibrate the model.

This can be better explained with a simple municipal planning exercise. Our task is to design a linear park that runs along an urban river. We need to give the landscape designers a plan for allocating space to various activities, including walking and bicycling. As a starting point, our plan is to preserve a 50-foot buffer of natural vegetation alongside the water, then have a 15-foot wide path for pedestrians, and finally a 25-foot wide path for cyclists. A generic model of a buffer is shown in Figure 5-1 and Code Listing 5-1. This plan can be described by three spatial buffer operations. In this model the computational process of creating buffer areas around geometries is hidden, or encapsulated. All that is made explicit are the required inputs—a spatial data set and a buffer distance—and the single output—a new spatial data set representing the buffer areas. The actual plan is shown as a map in Figure 5-2, in XML form in Code Listing 5-2 (some attributes, like `srsName` and `id`, are omitted in this example for illustrative

Figure 5-1: Model of a spatial buffer operation



Code Listing 5-1: Model of a spatial buffer operation

```
<xs:element name="Buffer" type="pamml:BufferType"/>

<xs:complexType name="BufferType">
   <xs:complexContent>
        <xs:extension base="pamml:VectorDataType">
           <xs:sequence>
              <xs:element name="InputGeometry" type="VectorDataType"/>
              <xs:element name="BufferValue" type="ValueDataType"/>
           </xs:sequence>
        </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

purposes), and as a diagram in Figure 5-3. The adapter design pattern allows the linear park plan to be ***composed*** of three buffer models, with each being an input to the next.

Composability is the ability to put together a piece of software from several components. Think of Lego™ building blocks and you have a good idea of how powerful and intuitive are systems exhibiting strong composability. In computer science, this is generally thought to be an essential property for building large and complex systems as it enables modularization and separation of concerns. Composability is made possible by the use of design patterns discussed earlier, such as ***inheritance*** (all vector spatial data types are descendants of a single generic type), ***adapters***, and ***encapsulation***. In systems distributed across computers and organizations, modularization and separation are more than critical; they are basic requirement. It is extremely elegant, therefore, that

composability not only enables the development of a distributed system, but also can be made to mirror organizational specialization.

Take the linear park model as an example. Code Listing 5-2 and Figure 5-3 present a site planning model composed out of three buffer models, with spatial data passing from one buffer model's output to the next one's input. But what format is the data in? This is not specified, so this model must be executed on a single computer system. In that case, there is no need to specify concrete data formats, allowing the software to choose its own preferred internal format (this is an important feature for commercialization, as it allows software companies to differentiate themselves, and charge a premium, based on their ability to execute algorithms well, even if all software packages are using a common language to describe the algorithm). In many cases, however, the system will not reside on a single computer, but will be distributed across multiple agencies. For argument's sake, let us assume that the river boundary data comes from the USGS; the extent of the natural buffer around the river is determined by the state department of environmental protection, and the recreational paths are set by the municipal parks department.

Figure 5-2: Linear park model, cartographic visualization

Code Listing 5-2: Linear park planning model

```
<Buffer name="Pedestrian path">
   <InputGeometry name="Bike path" type="BufferType">
      <InputGeometry name="Natural area" type="BufferType">
         <InputGeometry name="River" type="ShapefileReaderType">
            <ShpFile
               dataFile="http://www.usgs.gov/data?type=hydro&amp;fips=4&part=shp"/>
            <DbfFile
               dataFile="http://www.usgs.gov/getdata?type=hydro&fips=4&part=dbf"/>
            <ShxFile
               dataFile="http://www.usgs.gov/getdata?type=hydro&fips=4&part=shx"/>
            <SbnFile
               dataFile="http://www.usgs.gov/getdata?type=hydro&fips=4&part=sbn"/>
         </InputGeometry>
         <BufferValue name="Natural area extent" type="SimpleIntValue"
            units="feet" value="50"/>
      </InputGeometry>
      <BufferValue name="Bike path right-of-way" type="SimpleIntValue"
         units="feet" value="25"/>
   </InputGeometry>
   <BufferValue name="Pedestrian path right-of-way" type="SimpleIntValue"
      units="feet" value="15"/>
</Buffer>
```

Figure 5-3: Linear park model, diagrammatic visualization

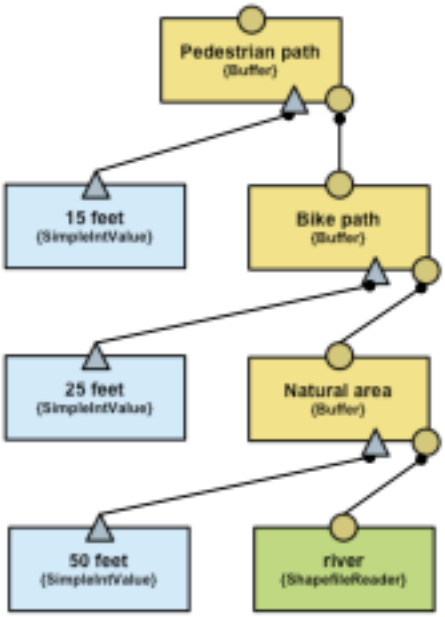


Figure 5-4: Linear park model, distributed across agencies

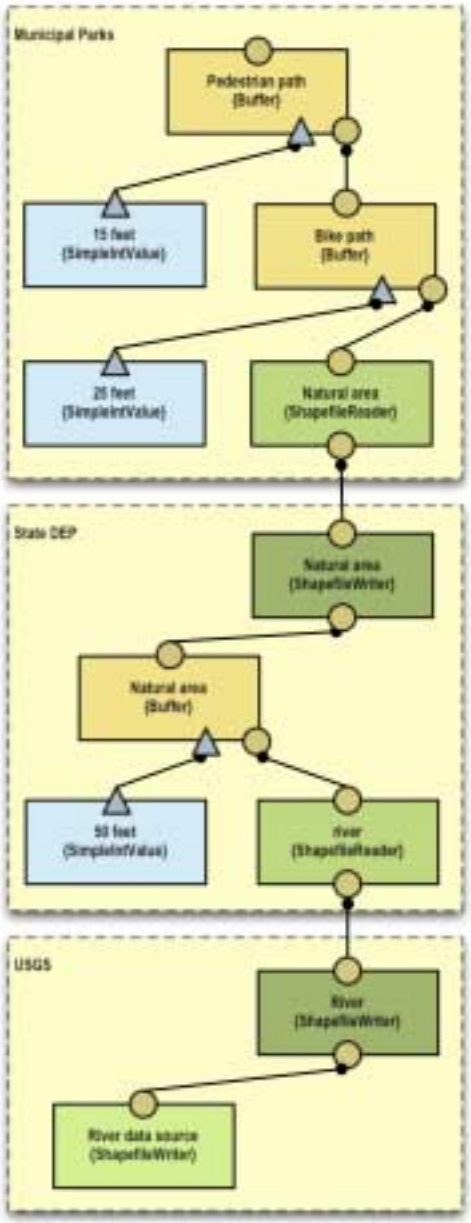


Figure 5-5: Some basic data models

In this scenario, we must use the adapter design pattern to guarantee interoperability across potentially heterogeneous systems in the various agencies. Figure 5-4 illustrates how our language implements the adapter pattern through data "readers" and "writers." While data is processed within a system, operations can be described abstractly, as in Figure 5-3 above. But whenever data moves from one system to another, a Writer adapter must be used on exit, and a Reader adapter must be used on entrance. In this way, most systems can be integrated into a distributed computing environment, as long as we can agree on a few basic data types. Some are shown in Figure 5-5. Note that even a simple data type like an integer is descended from the `ModelType` object. This has practical benefits in that this allows the value to acquire all the nice features of a model object, like metadata. More importantly is the semantic meaning. Even simple numbers are "abstracted, stylized models of real-world phenomena and processes." When the U.S. Environmental Protection Agency says that X parts per billion of heavy metals in a fish is not hazardous to an adult's health, X is not simply a number. It's a complex model.

Supporting legacy, or "black box" systems

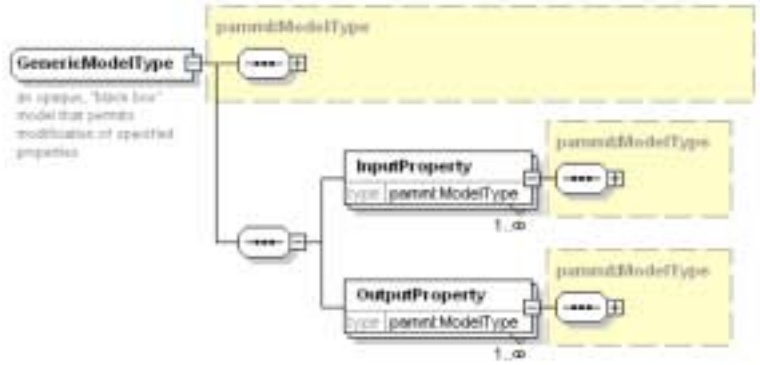
Not all models are as simple to describe as a buffer. It probably does not make sense to create a universal language that describes complex, scientific models in minute detail. There is little to gain and much to lose as there are probably many good reasons why expert domains have their own, unique discourse. What is more useful is to recognize that while core parts of an analytic model might always be a "black box"—indecipherable to all but a small group of experts—significant parameters may still be exposed to the computer systems of other modelers (like the corporate farmer mentioned earlier), and to the intellects of human decision makers. Therefore, the most important model in our system is the `GenericModel` (Figure 5-6). The `GenericModel` is important because it provides a quick way for organizations with "legacy" systems to participate in the new distributed framework without making major changes to their business processes. The drawback is that a system defined using generic models has less semantic meaning inherent in its description than others, and is therefore more difficult to integrate into collaborative analysis or decision support systems.

Code Listing 5-3: GenericModel XML Schema

```
<xs:element name="GenericModel" type="GenericModelType"/>

<xs:complexType name="GenericModelType">
   <xs:complexContent>
      <xs:extension base="ModelType">
         <xs:sequence>
            <xs:element name="InputProperty" type="ModelType" maxOccurs="unbounded"/>
            <xs:element name="OutputProperty" type="ModelType" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Figure 5-6: GenericModel, integrating legacy models



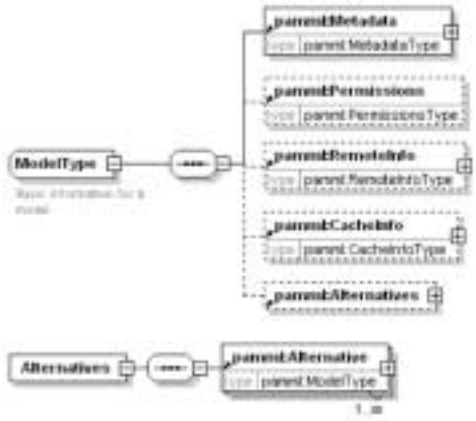
Collaborative planning: linking models with decision makers

There is a large body of work within the PSS field on participatory decision making, but the systems proposed are rarely integrated with the system used by the experts. This calls into question those systems' ability to truly capture feedback. In fact, the fault lies at both ends of the process because information systems rarely have the ability to capture and store any kind of debate around an analysis' results or techniques. So while researchers have experimented with effective systems that help explain complex phenomena to non-expert audiences, and enabled these audiences to play out scenarios that use alternative weights and values, the results of these experiences are generally captured ***outside*** of the original information system, in some form of human communication to the "experts." Although it would be hard to find a researcher in this field who did not think "feedback loops" were of critical importance, it would be just as hard to find software that actually implemented what could be called a feedback information system—a system for the storage, retrieval and analysis of discussion and debate around a planning model.

While the concept of a feedback information system is compelling, and is probably a necessary step in the evolution of participatory PSS, a thorough treatment of the topic is beyond the scope of this paper. For example, the types of feedback are numerous, from anecdotal commentary to survey instruments and voting. Each type of system might suggest a different information model, and might also require a different way of describing the participants, because the dynamics of public meetings are such that the type of participant matters as much as the issues being discussed. Here we seek to take a small step towards such a system by defining some structure in which to express the idea that what often happens in participatory PSS is that stakeholders want to explore "what-if" scenarios by substituting alternative values for a model's initial parameters. This one example will illustrate that the language has the ability, in general, to work in concert with potential participatory planning information systems.

Modeling systems usually do a good job of allowing the ***analyst*** to explore different scenarios by changing parameters, but they pay little attention to preserving this information. And if they do, they usually take an engineering approach, saving model runs in a scripting language or some other shorthand geared to be an efficient means of

Figure 5-7: Capturing feedback in the information system



restarting the modeling process. The approach espoused here is more of an information modeling approach, geared towards storing different opinions in a manner suited to visualization and analysis of the debate. Figure 5-7 shows the `ModelType` object with a new object called `Alternatives`. This object may contain an unlimited number of models that may be considered alternatives to the enclosing object. Software that implemented the `Alternatives` object would need to make sure that the output data type of the alternative models matched that of the enclosing object (XML Schema has no elegant way of articulating this constraint), but aside from that there is wide range of possibilities a software package could exploit using the `Alternatives` object. There are many other important issues to consider when designing a participatory information system, but they are not unique to the language being developed here, and are better handled in a more general collaborative computing research agenda.

This chapter has presented some features of PAMML most relevant to the challenges found in the buildout analysis. PAMML includes a number of additional objects and operations, which can be examined in the full XML Schema in Appendix A. Some of these, like `Union`, `Intersection`, `Difference`, and `Dissolve`, fill out the language's library of spatial operations. We have mainly discussed operations involving vector data, but the schema includes enough basic raster data types and operations to implement map algebra. Finally, some others add "inline" data types, which are XML-based descriptions of a data set, such as a table or a number. This simply allows the data to be included in the model, instead of requiring a remote reference.

Now that the PAMML framework has been developed, in the next chapter we return to the information management challenges that motivated this work, and we use these tools to reconstruct the buildout analysis, and show how the new framework reduces the cost and complexity of information processing.