Item 37 Validate Inside Your Program with Schemas

Rigorously testing preconditions is an important characteristic of robust, reliable software. Schemas make it very easy to define the preconditions for XML documents you parse and the postconditions for XML documents you write. Even if the document itself does not have a schema, you can write one and use it to test the documents before you operate on them. It is quite hard to attach a DTD to a document inside a program. Fortunately, however, most other schema languages are much more flexible about this.

For example, let's suppose you're in charge of a system at *TV Guide* that accepts schedule information from individual stations over the Web. Information about each show arrives as an XML document formatted as shown in Example 37–1.

Example 37–1 | An XML Instance Document Containing a Television Program Listing

```
<Program xmlns="http://namespaces.example.com/tvschedule"
  <Title>Reality Bites</Title>
  <Description>
   Elimination tournament in which contestants eat a
   succession of gross items until only one is left standing.
   Tonight's episode features rancid apples, insects, and
   McDonald's Happy Meals.
  </Description>
  <Date>2003-11-21</Date>
  <Start>08:00:00-05:00</Start>
  <Duration>PT30M</Duration>
  <Station>KFOX</Station>
</Program>
```

Every day, around the clock, stations from all over the country send schedule updates like this one that you need to store in a local database. Some of these stations use software you sold them. Some of them hire interns to type the data into a password-protected form on your web site. Others use custom software they wrote themselves. There may even be a few hackers typing the information into text files using emacs and then telnetting to your web server on port 80, where they paste in the data. There are about a dozen different places where mistakes can creep in. Therefore, before you even begin to think about processing a submission, you want to verify that it's correct. In particular, you want to verify the following.

- The root element of the document is `Program`.
- All required elements are present.
- No more than one of each element is present.
- The `Title` element is not empty.
- The date is a legal date in the future.
- The `Start` element contains a sensible time.
- The duration looks like a period of time.
- The station identifier is a four-letter code beginning with either K or W.
- The station identifier maps to a known station somewhere in the country, which can be determined by looking it up in a database running on a different machine in your intranet.

You could write program code to verify all of these statements after the document was parsed. However, it's much easier to write a schema that describes them declaratively and let the parser check them. The W3C XML Schema Language, RELAX NG, and Schematron can all handle about 85% of these requirements. They all have problems with the requirement that the date be in the future and that the station be listed in a remote database. These will have to be checked using real programming code written in Java, C++, or some other language after the document has been parsed. However, we can make the other checks with a schema. Example 37–2 shows one possible W3C XML Schema Language schema that tests most of the above constraints.

Example 37–2 A W3C XML Schema for Television Program Listings

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="Program">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="Title">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="1"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Description" type="xsd:string"/>
        <xsd:element name="Date"        type="xsd:date"/>
        <xsd:element name="Start"       type="xsd:time"/>
        <xsd:element name="Duration"    type="xsd:duration"/>
        <xsd:element name="Station">
          <xsd:simpleType>
            <xsd:restriction base="xsd:token">
              <xsd:pattern value="(W|K)[A-Z][A-Z][A-Z]"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

For simplicity, I'll assume this schema resides at the URL `http://www.example.com/tvprogram.xsd` in the examples that follow, but you can store it anywhere convenient.

There are several different ways to programmatically validate a document, depending on the schema language, the parser, and the API. Here I'll demonstrate two: Xerces-J using SAX properties and DOM Level 3 validation.

Xerces-J

The Xerces-J SAX parser supports validation with the W3C XML Schema Language. By default, it reads the schema with which to validate documents from the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes in the instance document. However, you can override these with the `http://apache.org/xml/properties/schema/external-schemaLocation` and `http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation` SAX properties. In this example, the documents being validated have namespaces, so we'll set `http://apache.org/xml/properties/schema/external-schemaLocation` to `http://www.example.com/tvprogram.xsd`. Then, we'll turn on schema validation by setting the `http://apache.org/xml/features/validation/schema` feature to true.

```
XMLReader parser = XMLReaderFactor.createXMLReader(
 "org.apache.xerces.parsers.SAXParser");
parser.setProperty(
"http://apache.org/xml/properties/schema/external-
schemaLocation",
  "http://namespaces.example.com/tvschedule"
  + " http://www.example.com/tvprogram.xsd");
parser.setFeature(
  "http://apache.org/xml/features/validation/schema",
  true);
```

We'll also have to register an `ErrorHandler` to receive any validation errors that are detected. Because validity errors aren't necessarily fatal unless we make them so, we'll rethrow the `SAXParseException` passed to the `error()` method. Example 37–3 shows an appropriate `ErrorHandler` class.

Example 37–3 A SAX `ErrorHandler` That Makes Validity Errors Fatal

```
import org.xml.sax.*;

public class ErrorsAreFatal implements ErrorHandler {

  public void warning(SAXParseException exception) {
    // Ignore warnings
  }

  public void error(SAXParseException exception)
   throws SAXException {

    // A validity error; rethrow the exception.
    throw exception;

  }

  public void fatalError(SAXParseException exception)
   throws SAXException {

    // A well-formedness error
    throw exception;

  }

}
```

This `ErrorHandler` also needs to be installed with the parser.

```
parser.setErrorHandler(new ErrorsAreFatal());
```

Finally, the document can be parsed. The parser checks it against the schema as it parses. At the same time, the `ContentHandler` methods accumulate the data into the fields. Since SAX parsing interleaves parser operation with client code, all the data collected should be stored until the complete document has been validated. Only then can you be sure the document is valid and the information should be committed. Example 37–4 demonstrates one way to build a `TVProgram` object that stores this data. The constructor is private, so the only way to build such an object is by passing an `InputStream` containing a `TVProgram` document to the `readTVProgram()` method. The `TVProgram` object is actually created before the parsing starts. However, it's not returned to anything outside this class until the input document has been parsed and any constraints verified. If a constraint is violated, then an exception is thrown.

Example 37–4 | A Program That Validates against a Schema

```
import java.util.*;
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class TVProgram extends DefaultHandler {

  private String title;
  private String description;
  private Date   startTime; // includes both date and time
  private int    duration;  // rounded to nearest second
  private String station;   // rounded to nearest second

  private TVProgram() {
    // Data will be initialized in the readTVProgram() method
  }

  private static XMLReader parser;

  // Initialization block. No need to load a new parser for
  // each document.
  static {
    try {
      parser = XMLReaderFactory.createXMLReader(
       "org.apache.xerces.parsers.SAXParser");
      parser.setProperty(
"http://apache.org/xml/properties/schema/external-schemaLocation",
       "http://namespaces.example.com/tvschedule"
       + " http://www.example.com/tvprogram.xsd");
      parser.setFeature(
       "http://apache.org/xml/features/validation/schema",
       true);
      parser.setErrorHandler(new ErrorsAreFatal());
    }
    catch (SAXException e) {
     throw new RuntimeException(
       "Handling exceptions in static initializers is tricky");
    }
  }
  public static TVProgram readTVProgram(String systemID)
   throws SAXException, IOException {

    TVProgram program = new TVProgram();
    parser.setContentHandler(program);
    parser.parse(systemID);

    // If no exception has been thrown yet, then the document
    // must be valid. However, we still have to check the
    // constraints the schema couldn't:
    checkDateInFuture(program.startTime);
    checkStationExists(program.station);

    // If we get here, everything's fine.
    return program;

  }

  private static void checkDateInFuture(Date date)
   throws SAXException {

    // Java code to compare the date to the current time

  }

  private static void checkStationExists(String station)
   throws SAXException {

    // JDBC code to look up the station call letters in our
    // database

  }

  // Various ContentHandler methods that will fill in the fields
  // of this object. This could be a separate class instead...

  // Various setter and getter and other methods...

}
```

Presumably, after such an object has been read, other code will store it in a database or otherwise work with it. And, of course, building an object that exactly matches the data in the document is far from the only way to model the data. All these details will depend on the business logic in the rest of the program. However, the input checking through validation will normally be similar to what's shown here.

DOM Level 3 Validation

DOM Level 3 provides a detailed API for validation. This API can be used to validate against any schema language the parser supports, although DTDs and W3C XML Schema Language schemas are certainly the most common options.

Caution *This section is based on working drafts of the relevant specifications and experimental software. The broad picture presented here is correct, but a lot of details are likely to change before DOM Level 3 is finalized.*

Unlike SAX, DOM objects can be validated when the document is first parsed or at any later point. You can also validate individual nodes rather than validating the entire document. To validate while parsing, you set the following features on the document or document builder's `DOMConfiguration` object.

- `schema-type`: A URI identifying the schema language used to validate. Values include `http://www.w3.org/2001/XMLSchema` for the W3C XML Schema Language and `http://www.w3.org/TR/REC-xml` for DTDs.
- `schema-location`: A white-space-separated list of URLs for particular schema documents used to validate.
- `validate`: If true, all documents should be validated. If false, no documents should be validated unless `validate-if-schema` is true.
- `validate-if-schema`: Validate only if a schema (in whatever language) is available, either one set by the `schema-location` and `schema-type` parameters or one specified in the instance document using a mechanism such as a `DOCTYPE` declaration or an `xsi:schemaLocation` attribute.

For example, here's the DOM Level 3 code to parse the document at `http://www.example.net/kfox.xml` while validating it against the schema at `http://www.example.com/tvprogram.xsd`.

```
DOMImplementation impl = DOMImplementationRegistry
   .getDOMImplementation("XML 1.0 LS-Load 3.0");
if (impl == null || !impl.hasFeature("Core", "3.0") {
  throw new Exception("DOM Level 3 not supported");
}
```

```
DOMImplementationLS implLS = impl.getInterface("LS-Load", "3.0");
DOMBuilder builder = implLS.createDOMBuilder(
  DOMBuilder.MODE_SYNCHRONOUS,
  "http://www.w3.org/2001/XMLSchema");
DOMConfiguration config = builder.getConfig();
config.setParameter("validate", Boolean.TRUE);
config.setParameter("schema-location",
  "http://www.example.com/tvprogram.xsd");
config.setParameter("schema-type",
  "http://www.w3.org/2001/XMLSchema");
builder.setErrorHandler(new DOMErrorHandler() {
  public boolean handleError(in DOMError error) {
    System.err.println(error.getMessage());
  }
});
Document doc = builder.parseURI(
  "http://www.example.net/kfox.xml");
```

Currently, this API is only experimentally supported by Xerces and the Xerces-derived XML for Java, but more parsers should support it in the future.

If you make modifications to a document, DOM3 allows you to revalidate it to make sure it's still valid. This is an optional feature, and not all DOM Level 3 implementations support it. If one does, each `Document` object will be an instance of the `DocumentEditVal` interface as well. Just cast the object to this type and invoke the `validateDocument()` method as shown below.

```
if (doc instanceof DocumentEditVal) {
  DocumentEditVal docVal = (DocumentEditVal) doc;
  try {
    boolean valid = docVal.validateDocument();
  }
  catch (ExceptionVAL ex) {
    // This document doesn't have a schema
  }
}
```

You can even continuously validate a document as it is modified. If any change makes the document invalid, the problem will be reported to the registered `DOMErrorHandler`. Just set the `continuousValidityChecking` attribute to true.

```
docVal.setContinuousValidityChecking(true);
```

This is particularly useful if the modifications are not driven by the program but by a human using an editor. In this case, you can even check the data input for validity before allowing the changes to be made.

If you need to change the schema associated with a document, set the `schema-location` and `schema-type` parameters on the document's `DOMConfiguration` object.

```
DOMConfiguration config = doc.getConfig();
config.setParameter("schema-type",
                    "http://www.w3.org/2001/XMLSchema");
config.setParameter("schema-location",
                    "http://www.example.com/schema.xsd");
```

To validate this document, you would then call `validateDocument()` as described above.

Validation with DOM differs from validation with SAX in that you don't actually begin working with the document until after it has been validated. Thus there's no need to worry about committing the data in pieces. This is a common difference between SAX and DOM programs. A second advantage is that DOM validation can be reversed so that you build the document in memory and then check for validity before outputting it. You can even check every node you add to the `Document` object for adherence to a schema immediately and automatically.

Whether you validate with SAX or DOM, whether you do so continuously or just once when the document is first parsed, and whether the schema is a DTD, a W3C XML Schema Language schema, or something else, validation is an extremely useful tool. Even if you don't reject invalid documents, you can still use the result of validity checking to determine what to do with any given document. For instance, you might validate documents against several known schemas to identify the document's type and dispatch the document to the method that processes that type. Validation is an essential component of robust, reliable systems.