# Here’s how to allocate your budget for your data efforts

Roger L. Costello
March 15, 2017

 *Spend less of your budget on XML Schemas. Spend less on UML diagrams.*

*Spend more on documenting and expressing domain knowledge.*

*Don’t use JSON.*

## Here’s why

XML Schemas contain structural rules: element A contains element B; element B contains element C; and so forth. Likewise, UML diagrams contain structural rules.

Structural rules are mostly just made-up, subjective, fleeting stuff.

The good stuff is in knowledge about the domain.

Constraints express knowledge.

Here are several examples of constraints (knowledge):

* Susan cannot be married to both John and Bill at the same time.
* The four nucleotides that make up DNA strands can only combine in particular sequences.
* The angles of a triangle must sum to 180 degrees.

Schematron is a machine-processable, declarative language for expressing constraints. In the UML realm, Object Constraint Language (OCL) is a language for expressing constraints. In other words, Schematron and OCL are languages for expressing knowledge.

**Recommendation #1. Allocate your budget this way**: Allocate minimal time, energy, and money to creating XML Schemas and UML diagrams. Allocate maximal time, energy, and money to documenting and expressing domain knowledge (constraints). Here’s how I recommend allocating your budget:



 **Recommendation #2. Do not use JSON**: JSON just contains structural rules, which, as we’ve seen, is of lesser importance. The JSON technology stack does not have an equivalent to Schematron. That is, the JSON stack does not provide a machine-processable, declarative language to express knowledge. As a consequence of this, programmers bury knowledge in Java code (or JavaScript code or Python code or C# code). For instance, they write Java code to constrain the sum of angles in a triangle to 180 degrees. That is terrible. *Therefore*, do not use JSON (except possibly for superficial browser-server exchanges).

## Justification

Every data problem has two parts:

* Structure (made-up stuff)
* Constraints (knowledge)

To see this, consider the following data problem.

 *Radio stations broadcast on frequency bands.
 Within a geographic area no two radio stations
 may broadcast on the same frequency band.*

That contains several nuggets of knowledge:

1. There are things called radio stations
2. There are things called frequency bands
3. Radio stations broadcast on different frequency bands (within a geographic area)

Consider the creation of an XML Schema for radio stations. (The development of a UML diagram is analogous.)

How do we create the XML Schema? We make stuff up! Like this:

Create an element and call it, well, let’s call it RadioStation.
 And then create a child element and call it, well, how about
 band. Finally, we make up a type and call it Freq.

We express that using the XML Schema language, resulting in something like this:

<xs:element name="GeographicArea">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="RadioStation" maxOccurs="unbounded">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="band" type="Freq" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:sequence>
 </xs:complexType>
</xs:element>

<xs:simpleType name="Freq">
 <xs:restriction base="xs:decimal" />
</xs:simpleType>

But that’s only one way to structure and name things. There’s nothing “correct” or “sacred” about that XML Schema. There are a million other ways to do it.

*XML Schemas express subjective, fleeting stuff.*

*XML Schemas contain little, if any, knowledge. It’s mostly just made-up stuff.*

Earlier we identified this nugget of knowledge:

Radio stations broadcast on different frequency bands.

That expresses a constraint on frequency bands used by radio stations. In other words, it expresses a nugget of knowledge.

Here’s how to express the constraint using Schematron:

<sch:pattern id="Knowledge-about-radio-stations">
 <sch:rule context="GeographicArea">
 <sch:let name="stations" value="RadioStation"/>
 <sch:assert test="
 every $s1 in $stations, $s2 in $stations satisfies
 if (pred:Disjoint($s1, $s2)) then
 number($s1/band) ne number($s2/band)
 else true()
 ">
 Radio stations broadcast on different frequency bands (within an area).
 </sch:assert>
 </sch:rule>
</sch:pattern>

Nice! It is a declarative, machine-processable expression of knowledge.

**Summary**: There is made-up stuff. It’s boring, subjective, fleeting, and can be implemented in a million different ways. Then there is knowledge. It’s exciting, objective, universal, and eternal (or, at least, more long-lived than structural stuff).

**Lessons Learned**: In any project, focus on documenting and expressing domain knowledge. If you’re creating XML Schemas be sure that you also create Schematron schemas to capture knowledge. If you’re creating UML diagrams be sure that you also create Object Constraint Language (OCL) rules to capture knowledge. The Schematron schemas and the OCL rules are vastly more important than the XML Schemas and the UML diagrams.