Component-based engineering

Formulate problem in terms of software components with well-defined cohesive properties.
Each software component is an independent program.
Connect components via message-passing: send requests and receive responses.
To the extent possible, construct a situation in which correctness of components implies correctness of the total application.
Example: mashup

Different components (in different languages)

- Google Maps
- JSON
- User interface
- API rental service
- Message
Advantages of Component-Based Engineering

Minimal coupling between components. Components can be written in different languages.
Testing scaffoldings for one component are relatively easy to write.
If interfaces between components are simple, system correctness is implied by component correctness; If A, B, C are correct, then a computation that utilizes communication between A, B, C is also correct.
Components can run on different physical machines.
Components can be created from legacy applications in virtually any language.
Disadvantages of component-based engineering

Components operate in an **open world**: any other process (on the internet(!)) can request services. (Subroutines operate in a **closed world**; the source code of the current program completely defines how they are used) Thus, security (and integrity) become significant problems.
Two component-based strategies

Service-Oriented Architecture (SOA):
create "services" that perform parts of a computation.
Designed for distributed network deployment... to optimize performance...
No rule against running all services on the same machine.

Windows .NET: create "components" that interoperate to create an entire computation.
Designed to run on an individual Windows machine... To make writing an application easier...
No rule against contacting components on other machines.

Similar strengths and weaknesses.
Similar architectures(!)
Service-Oriented Architecture

Goal: construct a computation from "services"
Challenge: services run on the internet, must invoke some kind of access control.
Solution: SOAP
SOAP: Simple Object Access Protocol

Service request and response are XML documents.

Clients convert subroutine calls to XML text.

Basic security/integrity model: guard clauses based upon XPATH and Xschemas
### How is the picture changing?

<table>
<thead>
<tr>
<th>Simple development</th>
<th>SOA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrote guard clauses into the code</td>
<td>guard clauses are external, in a different language.</td>
</tr>
<tr>
<td>The code functions in a closed world</td>
<td>We have to guard against previous input errors.</td>
</tr>
</tbody>
</table>
Guard Clauses in SOAP

Monday, November 19, 2012
1:52 PM

A compliant request is filtered at the service level, while bad requests and responses are repelled. A firewall pointer points toward "the bad guys."
Constructing SOAP Guard Clauses

XPATCH: a language for specifying parts of XML documents.

Xschemes: a language for specifying the appropriate form for an XML document.
How SOAP guard clauses work

Monday, November 19, 2012
1:57 PM
Attributes of SOAP guard clauses

External to the program.
Written in a pattern matching language.
Fully express the syntax limits for inputs.
Do not express the semantics of inputs.
The basics of XML

There is only one top level element.
Element opens (<cat>) and closes (<cat>) must balance. This is a context-free grammar.
Convention: <cat/>=<cat></cat>
The idea of XPATH

- Allow specification of parts of an XML document.
- Can specify three kinds of things:
  - A node: one XML element.
  - A nodeset: a sequence of nodes.
  - A condition: an empty nodeset is FALSE; a nodeset with at least one element is TRUE.
- An XPATH pattern that matches nothing is thought of as selecting the "empty nodeset".
Basic XPATH syntax

- **location**: where something is within the source document.
  - foo/bar matches an element named bar inside the content of an element named foo, without any intervening containing elements.
  - /foo/bar matches an element named bar inside the content of an element named foo, starting at top level of the document.
  - foo//bar matches an element named bar inside the content of an element named foo, at any level of nesting.
  - foo/@bar matches the attributes bar of tags named foo.
  - * matches any tag inside the current context.
  - ..: matches one level up ... foo/bar/.. matches any foo containing a bar.
  - .: matches current level: foo/bar/. is the same as foo/bar.

- Examples: consider the document:

```
<?xml version="1.0"?>
<candies>
  <candy><name>taffy</name><price>$5</price>
    <length>1 ft</length></candy>
  <candy><name>chocolate</name><price>$3</price>
    <weight>1 lb</weight></candy>
</candies>
...end of xpath/data.xml
```

- What nodesets do the following match?
  - candies/candy/name
  - candies/candy/*
  - candy/*
  - candy/../*
Advanced XPATH syntax

- **conditions**: select locations that satisfy tests.
  - foo/bar[@title='cat'] matches a tag <bar> inside a tag <foo>, that happens to have an attribute 'title' whose value is 'cat'
  - bar[@title and @publisher] matches a tag <bar> that has both 'title' and 'publisher' attributes.
  - bar[what] matches a tag <bar> that contains a tag <what>.

- **Examples**: 
  - candy[name='chocolate']
  - candy[name]
  - candy[2]

- **axes**: qualify how to interpret a name.
  - default is *child*: interpret names as children of current node.
  - self:: - current node.
  - parent:: - parent of current node.
  - preceding-sibling:: - siblings before current node.
  - following-sibling:: - siblings after current node.
  - descendant-or-self:: - current node or descendants.

- **Examples**: 
  - name.='chocolate']/following-sibling::*
  - candy[name.='chocolate']/following-sibling::candy
XPath functions

- **functions**: allow one to create complex conditions.
- **position()**: relative position of current document item.
- **last()**: number of total document items in context.
- **node()**: the node we're at in the tree.
- **count(nodeset)**: how many things match a pattern.

Examples:

- `candy[position()=last()]`
- `candy[position()=1]`
**XPATH shorthands:**

- `.` means `self::node()`
- `..` means `parent::node()`
- `//` means `/descendant-or-self::node()/`
- `foo/bar` is the same as `foo/child::bar`
- `foo//bar` is the same as `foo/descendant-or-self::bar`.
- `@thing` is the same as `attribute::thing`.
- `foo[<stuff>]` is a foo that contains `<stuff>`. 
Functions in conditions

- [...] - apply a condition to a pattern.
- child::*[@name and @address] - children with name and address attributes.
- Example: preceding-sibling::*[position()=1] (or preceding-sibling::*[1]) matches the node before the current one. Preceding indices count up as one moves backward.
- Example: following-sibling::*[position()=1] (or following-sibling::*[1]) matches the node following the current one at the same level.
- Example: child::*[count(@*)=2]: matches all children with exactly two attributes!
What this all means

XPATH is a way of specifying pieces of a document.
We will use this -- together with a glue language -- to move document parts around and change format.
This is a key concept in binding of services.
Xschemas

Define a pattern that a request must match.
Inputs not matching the pattern are not serviced.
Unit of matching: an XML document that represents the request.
Functionally, an Xschema -- together with a schema interpretation engine -- form a **guard clause** for a service.
XSchemas

- Lecture source: XML in a Nutshell, O'Reilly Inc (chapter on XML Schemas)
- Second source: documentation for Xschemas: http://www.w3.org/TR/xmlschema-0/

Goal of schemas

- Create a language that describes the **grammar with which programs communicate**.
- Define very expressive notion of what it means to be a complying document.
- Make it possible to utilize "self-defining" formats: if one sends a schema with a document, one *knows* whether the document conforms or not.
- Develop an algebra of "enveloping types" and "type restriction".
Type restriction
• This is a subtle point, but also a major theme of the course.
• Suppose we have two schemas. There is a well-defined notion of what it means for one to be "more restrictive" than another.
• Definition: a schema X is more restrictive than the schema Y whenever the set of documents that match X is a subset of the set of documents that match Y.
• A key idea: if one has a program that works given input conforming to a schema Y, and X is more restrictive than Y, and we have a document D that validates according to X, then the program will work on D.

Example:
• Simple XML document
  • <?xml version="1.0"?><fullName>Alva Couch</fullName>
• Schema for that document (call it "address.xsd")

  <?xml version="1.0"?><xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="fullName" type="xs:string"/>
  </xs:schema>
  ○ xmlns:xs: a namespace: all things in schema start with xs:
  ○ xs:element: an element in the schema in namespace xs:
  ○ xs:string: a data type.
• Embedding schema references: you can embed them in the document itself:

  <?xml version="1.0"?><fullName xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="document02.xsd">Alva Couch</fullName>

Then, put the schema elsewhere in document02.xsd:
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="fullName" type="xs:string"/>
</xs:schema>
Data types

- tell you what can be in the content of a tag.
- xs:anyURI: A Uniform Resource Identifier
- xs:base64Binary: Base64-encoded binary data
- xs:boolean: true or false, 0 or 1
- xs:byte: integer \( >= -128 \) and \( <= 127 \)
- xs:dateTime: date and time
- xs:duration: length of time in some units.
- xs:integer: positive or negative integer
- xs:decimal: signed number
- xs:language: same values as xml:lang
- xs:Name: an XML name
- xs:string
Attributes

• quite complex and expressive system for defining attribute types.

```xml
<xs:element name="fullName">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="language" type="xs:language"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>
```

• attributes of a thing are related to the thing semantically, not syntactically.
  ○ complexType: not expressible in terms of simple types above
  ○ simpleContent: the basic building block of a type
  ○ extension: define the type of the simple content.
  ○ attribute: define an attribute related to the type of the simple content.

• This is a kind of inheritance
  ○ base type is `xs:string`
  ○ derived type has an extra attribute, `language`.
  ○ the attribute goes in the master element, but refers to a sub-element!
  ○ we can tell which attribute refers to which subcontent!
**Namespaces:**

- A *namespace* is created by utilizing a common prefix for XML elements.
- There are several of these:
  - xsl: extensible stylesheet
  - xs: xml schema
  - xsi: xml schema instance
- Namespaces must be defined for both schema and document having that schema.
  - For the document:

  ```xml
  <foo:fullName xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:foo="http://www.eecs.tufts.edu/g/150WEB/notes/xschema/document03.xsd"
    xsi:schemaLocation="http://www.eecs.tufts.edu/g/150WEB/notes/xschema/document03.xsd"
    http://www.eecs.tufts.edu/g/150WEB/notes/xschema/document03.xsd"
    language="en">Alva Couch</foo:fullName>
  ```
  - xmlns: where to get definitions for the namespace
  - The schema is again:

  ```xml
  <?xml version="1.0"?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="fullName" type="xs:string"/>
  </xs:schema>
  ```
Qualification

- a schema controls whether a document must contain qualifiers for the namespace, or whether they are implied.
  - case one: unqualified; no addr: in front of each element or value name; "think" that they're there.

- Consider this:
- `<?xml version="1.0"?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="qualified"
    elementFormDefault="qualified">
    <xs:element name="fullName" type="xs:string"/>
  </xs:schema>

  - attributeFormDefault: controls attributes.
  - elementFormDefault: controls elements. This means that we must put addr: in front of each element, like this:

    `<addr:fullName xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:addr="http://www.eecs.tufts.edu/g/150WEB/notes/xschema/document04.xsd" language="en">Alva Couch</addr:fullName>`
Complex types

- Any element type containing other elements (rather than text) is a "complex type".
- Suppose we want to define a document like this:
  
  ```xml
  <address>
    <fullName>
      <first>Alva</first>
      <last>Couch</last>
    </fullName>
  </address>
  
  Then address has a "complex type".
- We start by defining the namespace elements for the type and qualifying names if necessary:
- Then we have to define complex types in the schema. This is the document:
  
  ```xml
  <?xml version="1.0"?>
  <addr:address xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:addr="document05.xsd"
    addr:language="en">
    <addr:fullName>
      <addr:first>Alva</addr:first>
      <addr:last>Couch</addr:last>
    </addr:fullName>
  </addr:address>
  ```

- Here's our schema: *document05.xsd*

  ```xml
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
    <xs:element name="address">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="fullName">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="first" type="nameComponent"/>
                <xs:element name="last" type="nameComponent"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
  ```
• xs:complexType: prepares us for a thing that will contain elements
• xs:sequence: an element container, specifies sequencing.
• type=addr:nameComponent: refers to a subtype declaration!
• this particular subtype is just a string, but we could be more demanding about it.
• even one element requires a sequence.
Sequences and occurrences

- minOccurs and maxOccurs: define limits on number of occurrences.
  
  ```xml
  <xs:complexType>
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="middle" type="xs:string" minOccurs="0"/>
      <xs:element name="last" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  ```

- A first name, followed by an optional middle, followed by a last name.
Repeated and empty elements

- consider this schema
  
  contents of xschema/document06.xsd...

```xml
<?xml version='1.0'?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="phonelist" type="myListType"/>
  <xs:complexType name="myListType">
    <xs:sequence>
      <xs:element name="phone" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="number" type="xs:string"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

- This accepts

```xml
<?xml version='1.0'?>
<addr:phonelist xmlns:addr="document06.xsd">
  <addr:phone number='555-1212'/>
  <addr:phone number='628-5000'/>
</addr:phonelist>
```

- Parts of this declaration:
  - phonelist is defined via listType.
  - listType is a sequence with perhaps unbounded instances.
○ instances are named phone.
○ each one of these has an attribute number that is a string.
○ each one is otherwise empty.
May I beg your indulgence?

- really understanding this nonsense requires a simpler notation.
- may I utilize unqualified documents for now?
- this would make things simpler
- my examples are "bad practice",
- but they'll be readable
- and the simple thing of adding namespace qualification will make them acceptable.
Complex content

- mechanism for inheriting type characteristics
- note that attributes can be inherited!

```
<xs:element name="phone" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:complexContent>
      <xs:restriction base="xs:anyType">
        <xs:attribute name="number" type="xs:string"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- `xs:complexContent`: something that has a restricted type
- `xs:anyType`: start with anything.
- `xs:attribute`: put in an attribute.
- `disallow anything else.`
Creating new simple types

- We can make up a new simple type
  
  `<xs:simpleType name="locationType">`
  `<xs:restriction base="xs:string"/>`
  `</xs:simpleType>`

- Not useful yet, but wait, there is a way to make it useful.
Facets:

- way of defining complex content by property inheritance and restriction.
- length, minLength, and maxLength
- pattern
- enumeration
- whiteSpace
- maxInclusive and maxExclusive
- minInclusive and minExclusive
- totalDigits
- fractionDigits

### length

- must often know how long one's input can be

```xml
<xs:simpleType name="shortString">
  <xs:restriction base="xs:string">
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
```

- xs:restriction: defines base type to restrict.
- xs:maxLength: only 10 characters.

### whitespace facet

- determines how to handle whitespace in validating a document
  - collapse: replace multiple white space with one space
  - preserve: treat all whitespace literally
  - replace: every whitespace character becomes a space.
• This is done *before* validation!

```xml
<xs:simpleType name="shortString">
  <xs:restriction base="xs:string">
    <xs:maxLength value="10"/>
    <xs:whitespace value="collapse"/>
  </xs:restriction>
</xs:simpleType>
```

matches any string that, after multiple spaces are replaced by one space, has no more than 10 characters in it!

**enumeration facet**

• restricts a string to a set of values

```xml
<xs:simpleType name="locationType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="work"/>
    <xs:enumeration value="home"/>
  </xs:restriction>
</xs:simpleType>
```

• This limits a location to be either "work" or "home"

**numeric facets**

• control what kinds of numbers will be accepted in a document.

• Example:

```xml
<xs:simpleType name="onetoten">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="10"/>
  </xs:restriction>
</xs:simpleType>
```

matches the numbers 1 to 10, inclusive.
• minExclusive and maxExclusive have the obvious meanings (but only mean something useful for decimal numbers)
  
  ```
  <xs:simpleType name="positive">
    <xs:restriction base="xs:decimal">
      <xs:minEnclusive value="0"/>
    </xs:restriction>
  </xs:simpleType>
  ```

  defines a decimal number > 0

  **floating point**
  
  • format is ASCII!

  • Example:
    
    ```
    <xsd:simpleType name="myNumber">
      <xsd:restriction base="xs:decimal">
        <xsd:totalDigits value="5"/>
        <xsd:fractionDigits value="2"/>
      </xsd:restriction>
    </xsd:simpleType>
    ```

  • Oops! This defines total and fractional digits, *not* sign and decimal point. 234.34 matches, as well as -112.40

  **patterns**

  • Can also match general data patterns.
    
    ```
    <xs:simpleType name="ssn">
      <xs:restriction base="xs:string">
        <xs:pattern value="d\d\d-\d\d-\d\d\d\d"/>
      </xs:restriction>
    </xs:simpleType>
    ```

    matches all social security numbers in the pattern 123-45-6789

  **lists**
• can parse content into space-separated words, put conditions on the words
  <xs:simpleType name='Words'>
    <xs:restriction base="xs:string">
      <xs:list itemType="ssn"/>
    </xs:restriction>
  </xs:simpleType>

  declares a list of ssn's!

unions
• can allow alternatives:
  <xs:attribute name="identifier">
    <xs:simpleType>
      <xs:union memberTypes="ssn phoneNumber"/>
    </xs:simpleType>
  </xs:attribute>

  says that an identifier is either a ssn or a phoneNumber.

complex types
• Complex types can be mixed, meaning that character data can be mixed with elements.
  <xs:element name="letter">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="greeting" type='greetingType'/>
        <xs:element name="body" type='bodyType'/>
        <xs:element name="closing" type='closingType'/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
choice

- xs:choice allows one to select between alternatives for complex content:
  
  `<xs:element name="greeting">
    `<xs:complexType mixed="true">
      `<xs:choice>
        `<xs:element name="hello"/>
        `<xs:element name="hi"/>
        `<xs:element name="dear"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>``

all

- xs:all controls whether elements appear in any order:

  `<xs:element name="body">
    `<xs:complexType mixed="true">
      `<xs:all>
        `<xs:element name="perpetrator"/>
        `<xs:element name="victim"/>
        `<xs:element name="cashAmount"/>
      </xs:all>
    </xs:complexType>
  </xs:element>``

  says that somewhere in mixed content, there should be three elements, perpetrator, victim, and cashAmount.

- E.g.

  `<body>
    The idiotic <perpetrator>burglar</perpetrator> stole <cashAmount>$1</cashAmount> from <victim>Mr. Stevens</victim>.  
  </body>`
any

- the xs:any label allows any content at all.

```xml
<xs:element name="notes" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:any
        namespace="http://www.w3.org/1999/xhtml"
        minOccurs="0" maxOccurs="unbounded"
        processContents="skip"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

says that the notes element can contain arbitrary XHTML, and that we should not validate it.
OUCH! This is complicated.

- Must define, for each object, the qualifications it has.
- Must have these qualifications agree between object producer and consumer.
- Result: everything works!
Some very subtle notes

- Can only apply one restriction or extension at a time.
- Must nest types to both extend and restrict.
- Reason: need to know order in which to extend and restrict. Order does matter!
- Example: create an element 'bar' with an attribute foo having string content, and content that is an ssn.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
<xs:element name='bar' type='secondtype'/>
<xs:complexType name='firsttype'>
  <xs:simpleContent>
    <xs:restriction base="xs:string">
      <xs:pattern value="\d\d\d-\d\d-\d\d\d\d"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name='secondtype'>
  <xs:simpleContent>
    <xs:extension base="firsttype">
      <xs:attribute name="language" type="xs:language"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
</xs:schema>
```

Caveats:

- nameSpace concept defined in documentation, not allowed in schemas.
  - Basic idea: create an abstract string that defines the scope of your own names.
  - Usual value: the url of your xsd file.
  - In specification, not recognized in schema for XSD!