RMIT School of Computer Science & IT

COSC2104/2106 Document Markup Languages

Patterns in XML Schemas

The Russian doll pattern

This pattern is coined after the famous Matryoshka Russian dolls—wooden dolls of decreasing size placed one inside another. The Russian doll pattern defines all subelements locally; thus, each element and its type are encapsulated by their parent, much like the Russian dolls.
The example in Listing 8—a simplified representation of a help document for a home appliance—demonstrates this pattern:

```
<xs:schema>
<xs:element name="HelpDoc">
<xs:complexType>
 <xs:sequence>
  <xs:element name="Section">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="Title" type="xs:string"/>
     <xs:element name="Body" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
 <xs:/complexType>
</xs:element>
</xs:schema>
```

NB only one global element

An associated instance might look like:

```
<HelpDoc>
 <Section name="operation_instructions">
  <Title>Operating your appliance.</Title>
  <Body>First, open the packaging and check to see...</Body>
 </Section>
</HelpDoc>
```

You can see that every sub element, attribute, and type in the listing is defined **locally**. The only global element is the root, HelpDoc. This syntax is compact and some may consider it easily readable. Russian doll style schemas don't expose their components to other types, elements, or schemas, so they are also considered highly decoupled (that is, elements are not globally dependent on other elements) and cohesive (related elements are grouped within a single self-contained parent).

Salami slice pattern

With the Salami slice pattern, you take the next step toward exposing content models. In this pattern, you move all your **locally defined elements** into **global** definitions.

The Salami slice pattern exposes **all elements** so you can reference and reuse them in other parts of the schema, and it makes them transparent to other schemas. A major advantage in this approach is that elements are highly reusable. However, this also means that all namespaces are globally exposed, and coupling between elements increases. In the example, the Section element is globally coupled to the Title and Body elements. Any modification to the Title and Body elements would subsequently affect the Section definition.

Note than when we refer to these global element definitions, we use "**ref**"

```
<xs:schema>
 <xs:element name="Body" type="xs:string"/>
 <xs:element name="Title" type="xs:string"/>

<xs:element name="Section">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="Title"/>
   <xs:element ref="Body"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string"/>
 </xs:complexType>
 </xs:element>

 <xs:element name="HelpDocs">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="Section"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Venetian blinds pattern

In the Venetian blinds pattern, instead of defining all elements globally, you start by **defining all types globally**, as in the example.

The Venetian blinds style **uses global type definitions** to increase reuse capabilities. Because all sub elements are localized, it comes with the added benefit of being able to hide namespaces. This approach allows you to expose your structure definitions for reuse while using the elementFormDefault attribute as a switch to hide or expose namespaces. You get the best of both worlds

When we refer to global type, we refer to the type name. Note that inbuilt types do not need to be redefined, as they are already available.

```
<xs:schema>

    <xs:complexType name="section.type">
      <xs:sequence>
        <xs:element name="Title" type="xs:string"/>
        <xs:element name="Body" type="xs:string"/>
      </xs:sequence>

      <xs:attribute name="name" type="xs:string"/>
    </xs:complexType>

    <xs:complexType name="helpdocs.type">
      <xs:sequence>
        <xs:element name="Section" type="section.type"/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name="HelpDocs" type="helpdocs.type"/>
</xs:schema>
```

Garden of Eden pattern

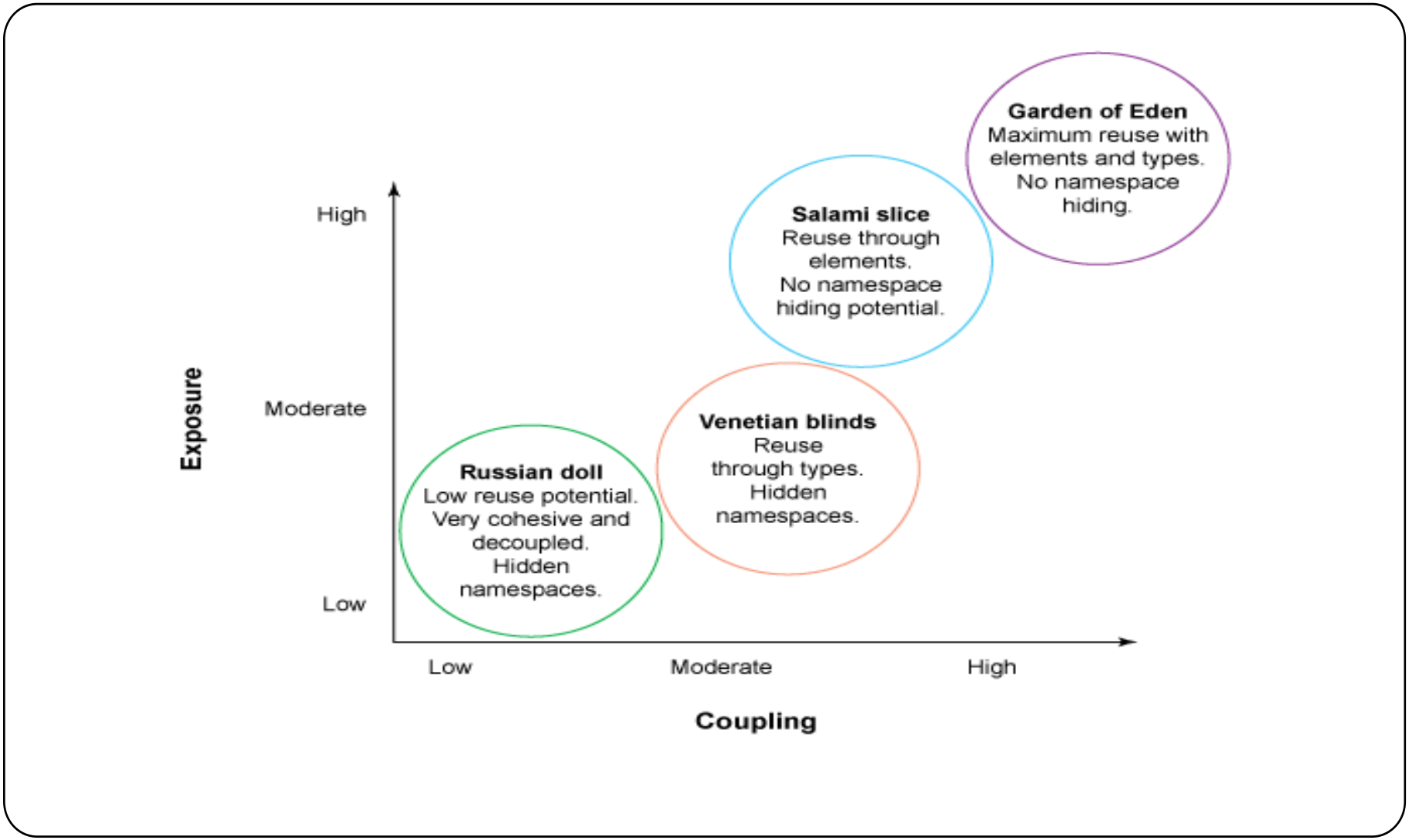
In the Garden of Eden design pattern, you make **both element declarations and type declarations global,** taking globalization to the extreme

By making every possible element, attribute, and type global, you create a scenario that maximizes reuse, both internally and between schemas—albeit by forcing namespaces to be exposed. By completely exposing your structure, you make the schema highly coupled but agile. Because elements are interdependent, sweeping changes to the schema can be applied quickly.

```
<xs:schema>
    <xs:attribute name="name" type="xs:string"/>
    <xs:element name="Title" type="xs:string"/>
    <xs:element name="Body" type="xs:string"/>
    <xs:element name="Section" type="section.type"/>
    <xs:element name="HelpDocs" type="helpdocs.type"/>

    <xs:complexType name="section.type">
      <xs:sequence>
        <xs:element ref="Title"/>
        <xs:element ref="Body"/>
      </xs:sequence>
      <xs:attribute ref="name"/>
    </xs:complexType>

    <xs:complexType name="helpdocs.type">
      <xs:sequence>
        <xs:element ref="Section"/>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```


Garden of Eden
Maximum reuse with elements and types. No namespace hiding.
Salami slice
Reuse through elements. No namespace hiding potential.
Venetian blinds
Reuse through types. Hidden namespaces.
Russian doll
Low reuse potential. Very cohesive and decoupled. Hidden namespaces.
Exposure
High
Moderate
Low
Low
Moderate
High
Coupling

Best practices: Schema management and evolution

When designing schemas, you often perform a balancing act among exposing reusable components, hiding namespaces and limiting namespace exposure, and decreasing coupling (or the interdependence of multiple global elements/types). The previous figure summarizes the reuse potential of each of the four schema patterns, indicating their relative rank in both coupling and exposure:

Providing high potential for reusing schema components can reduce future development time and make sweeping changes easy. However, it can also create scenarios in which multiple elements and types are unnecessarily coupled. When schemas become highly coupled, elements and types become interdependent, making it difficult to manage future changes and additions. Coupling run amok prevents schema evolution, because other systems depend on your interfaces remaining consistent. It's important to be careful with what and how much you expose. Once you make a choice, it can be difficult to undo.