Knowledge Engineering Final Assessment

# Question 1a

I developed an XSD to describe a virtual media library. Currently it only contains games like a “Steam Library” but this can easily be extended to include more things like Books, Music, Podcasts ect. Below I will list out the features of my XSD and explain how they get used

## Named Elements

I implemented both simple and complex named elements. A simple type can be understood as an element which only contains one datatype literal eg. xsd:string or xsd:decimal.



Shown above is a simple type I created called monetaryValueType which will hold the price of the game. I have also placed a simple restriction on it saying that it can only be of type xsd:decimal with 2 fractional values for the cents.

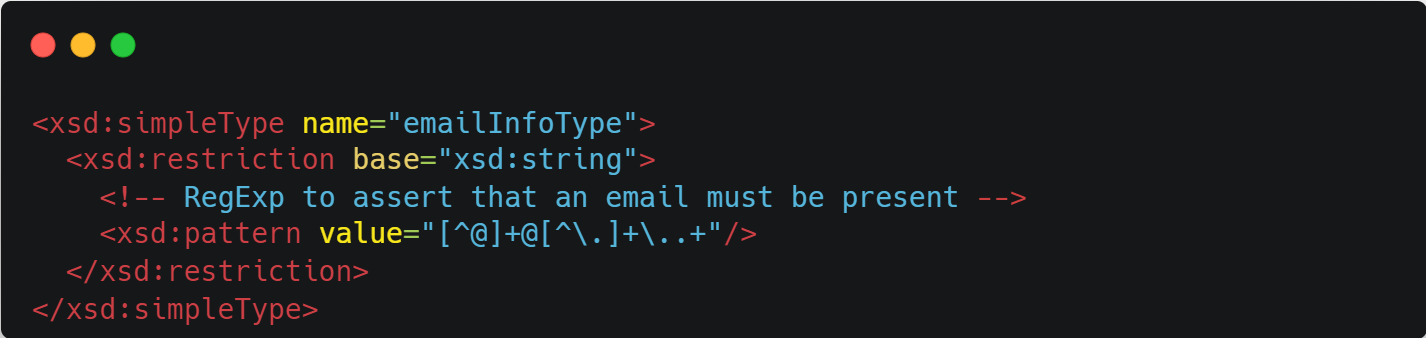
We can also make complex elements in XSD. This is when a type is made up by nesting other simple types:



Here I have create a fullname complex type that nests the firstname and lastname simple types. This was used to create the names for the owner of the library also for the names of the developers of the games.

## Restrictions

Restrictions allow us to define limits on the things that can be contained in this element. In the simple named element, I showed how the monetaryValueType can be restricted such that it can only contain valid price values, decimal numbers that only have 2 fractional digits. Alternatively, one can also place a regex restriction on the type.



This uses the regex pattern “[^@][+@[^\.]+\](mailto:+@[%5e\.%5d+\)..+” to check whether the input provided conforms to an email pattern, namely two alphanumeric strings separated by an @ symbol and a third string being separated by a “.” E.g “johnnyphone@gmail.com” . Any other string format used in the XML would throw an error.

## Enumeration

Enumeration is used when we know the XML element can only have a fixed discrete set of options. For example, if we had to describe the weeks of the day, we would put an enumeration. This way the user only has the option to select from the options we provide rather than giving us anything. Using this restriction where possible allows us to simplify XML processing as we have a guarantee that this option will have a limited number of possibilities. I used this in the Game ontology to describe the ESRB rating of the game, which is a type of age rating system to show which age ranges this game is suitable for.



## Cardinalities

Cardinalities allows to define how many times an element can occur. By default the minimum occurrence is set to 1. But this and the max occurrences can be changed to whatever we want. I used this in the Series element. A Series, as I have defined it, is a collection for multiple games. As such it needs to have at least one game, but there should be no limitation as to how many games a Series can have.



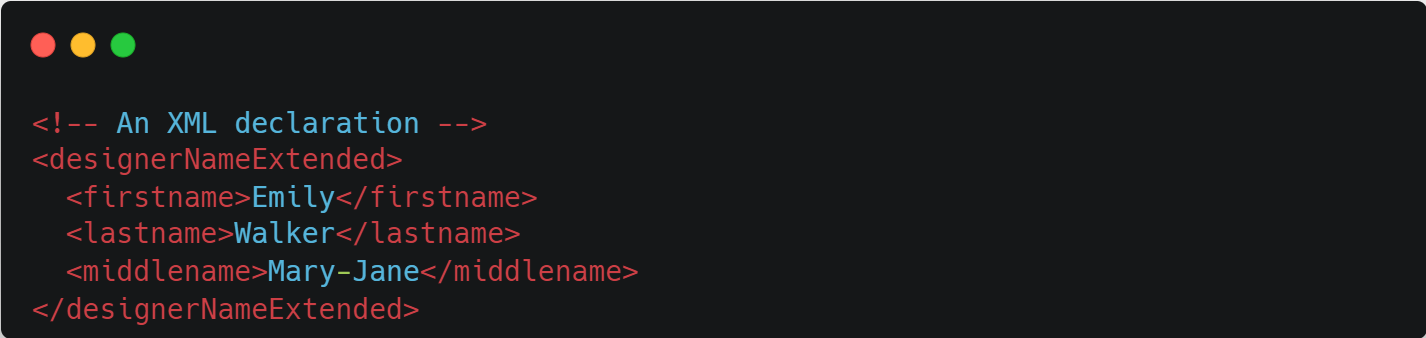
By setting the maxOccurs to unbounded, I have specified that any element with the series type can have as many games in it as it wants. I could not find an element in my ontology that could make use of a minOccurs greater than 1.

## Derived By Extension

In XSD this refers to the practice of creating a new type by extending another type. I used this in the fullname to allow users to define a middle name as well as a first and last name.



This means that the fullnameExtendedType will inherit all the fields form fullnameType and will add the middlename field after the inherited field. This was used to create the designerNameExtended element which was used to describe designers of a game who want to have their middle name documented.



It is important to note that the added field, in this case the middlename field, has to go after the fields from the original type.

## Sequence

The sequence tag allows us to say that the type must have a series of elements. I made use of this in the game type to specify that it must have:

* Title
* Designer Name (normal or extended version)
* If it is Installed
* Price
* Payment method (either by card or by paypal)
* ESRB Rating
* Popularity/Game Rating
* Genre



## Choice

This allows us to specify that users can choose between which tag to instantiate. For example, my ontology supports 2 types of payments, either card payments or PayPal payments. When creating a game element, they have to specify whether they used a card to buy the game and the first and last 4 digits associated with the card or if they used PayPal and the email associated with that account.



The choice was also used in defining the game type so that users can either use the fullname type when referring to the developers or they can make use of the fullname extended type.

## Attributes

Attributes allow us to embed some data in the opening of an XML tag rather than that data being nested inside the XML tag. This could be used for information which may be important to have, but not important enough to warrant its own element. I made use of this in the Series type. The attribute allows users to define the Series title.

A screen shot of a computer code

Description automatically generated

# Question 1b

Nothing demonstrated the power of my XSD more than the series type. This is contained in a Media library and can contains games. An example of a series called Kingdoom is shown below:

A screen shot of a computer code

Description automatically generated A screen shot of a computer code

Description automatically generated

Here I show how the unbounded maxOccurs cardinality allow us to nest as many games in the Series element as we want. The choice between payment options is also demonstrated here with “Kingdoom II: Longsword” and “Kingdoom IV: Excalibur” being bought with a card who’s first and last digits separated by a period is 3060 and 7653 respectively. “Kingdoom I” and “Kingdoom III: Sabre” were brought using PayPal and hence they have the associated email rather than a card number. The regex restriction of the email type can also be seen to work here. If we inlcude an email that isn’t valid like “johnnyphones@gmail@yahoocom” the XML parser will throw an error:

A screenshot of a computer

Description automatically generated

We can also see the enumeration at play in the esrbRating element. If we specify an input that is not in the enumeration specified, the XML parser will throw an error:

A screenshot of a computer program

Description automatically generated

The Choice feature in XSD can be seen in the <paymentMethod> tag in the XML. We can specify either a card payment or a PayPal payment but not both.

A screenshot of a computer

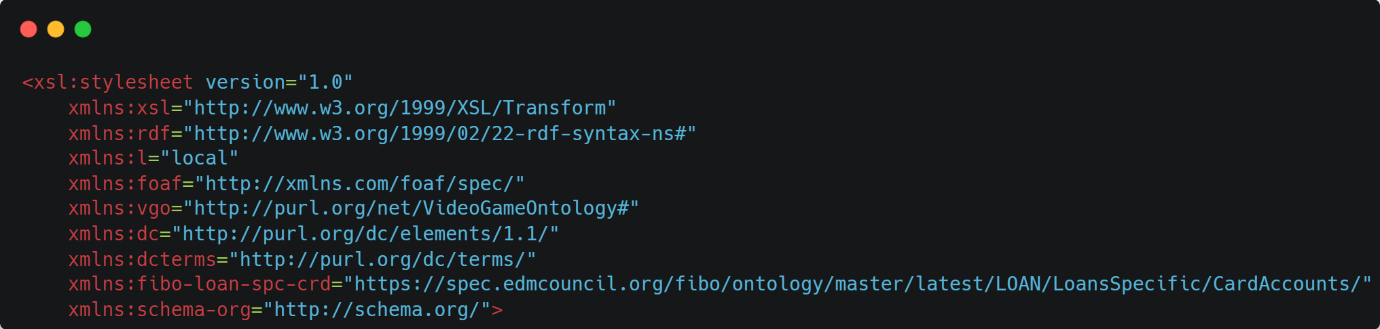
Description automatically generated

If one tries to specify both types, the first one in order takes precedence and the XML parser will throw an “Invalid Content” error as shown above.

# Question 1c

Here I had to develop an XSLT that would convert the generated XML into RDFS encoded in XML. The XSLT was coded in Altova XMLSpy as provided error checking and a feature to apply the XSLT to an XML document and view the output.

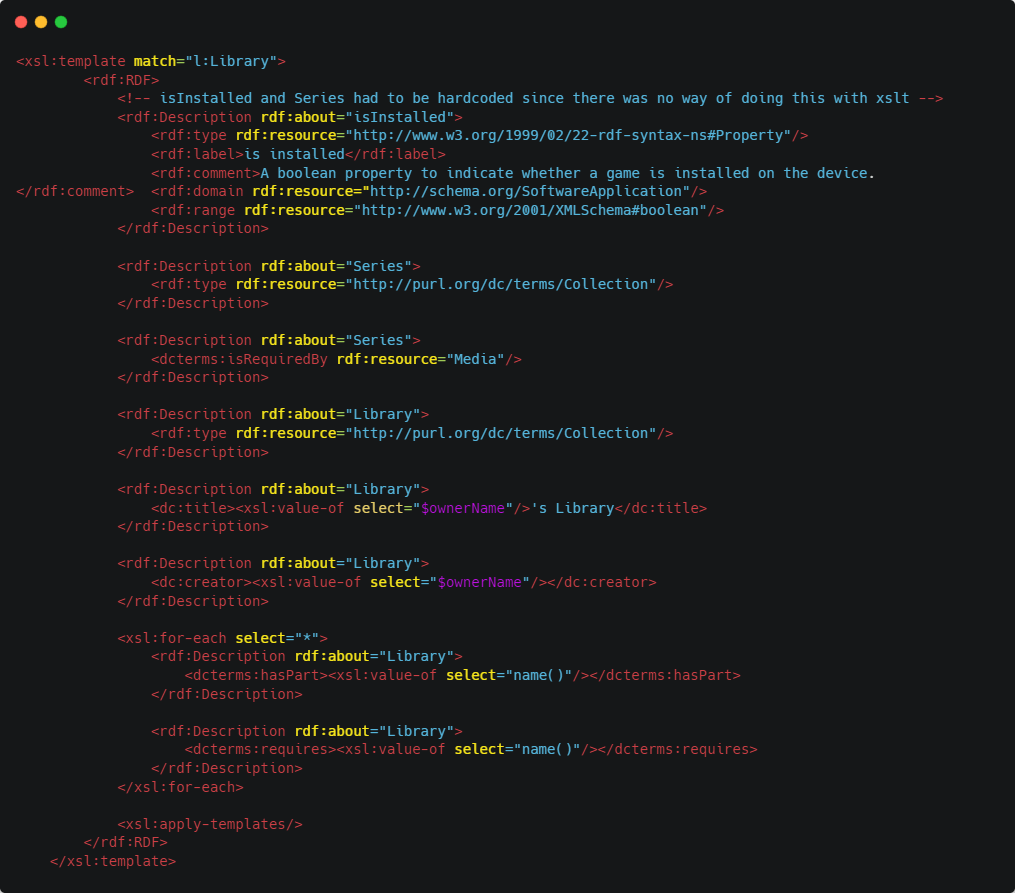
The XSLT began with the definitions of some namespaces that I would use throughout this transformation stylesheet. There are some common namespaces such as foaf which is an ontology to describe people, rdf which provides access to the Resource Description Framework tags and dcterms which provides access to the Dublin Core terms. Some other namespaces that helped with the description of this video game ontology includes VideoGameOntology (vgo), FIBO (fibo-loan-spc-crd) and Schema.org (schema-org).



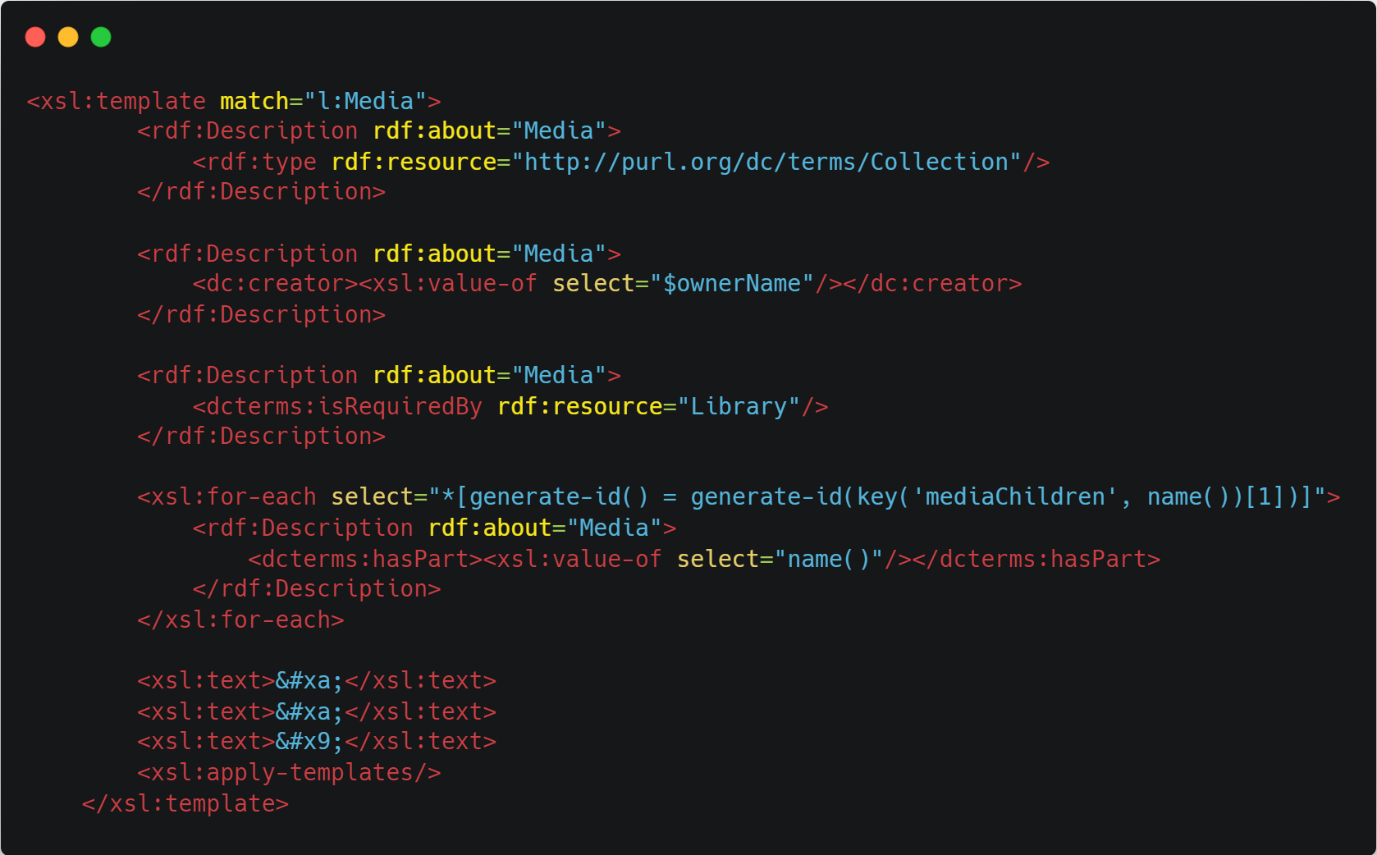
I then proceeded to created a variable for the name of the Owner, and a key for all the children of Media. The reason for this key will be discussed later on. The XSLT then begins by matching the “/” which means match the root and then perform everything contained in <xsl:template> which in this case is to apply all templates which have a match for the children of the root.



The match at this level is the Library element. The code for the <xsl:template> matching the library is shown below



I begin by opening the RDF document, indicated by the <rdf:RDF> tag. Then I create a new description about the “isInstalled” element since I couldn’t find any publicly available ontologies that defined this element. In order to provide a description about an element, one has to use the <rdf:Description> tag, this is how things are defined in XML based RDF. In order to inject values into the resulting document from the provided XML, I made use of the <xsl:value-of select=“XXX”/> operation. This allows the XSLT parser to select a global variable indicated by the “$”, or a property of the current element like its name via name() or the text it contains via text(). XSLT also provides us with for-loops, in this case I used it to loop over all the children of Library, indicated by <xsl:for-each select=“\*”>, and inset the text contained within. Finally, at the end I match any templates that apply to the children at this layer.



Shown above is the XSLT template used for the Media element. Much of it is the same as the template used for the Library element, however there is a small difference in the for-loop. I wanted to programmatically say that Media contains Series and Game as its children using the <dcterms:hasPart> tag. However, if I said “ select=“\*” ” then it would loop over every instance of Series and Game, meaning there would be many duplicates in the output. Instead, the expression in the “select” is used to select all nodes (\*) where the ID of the node matches the ID of the first node returned by the “ key(‘mediaChildren’, name()) ” function. Essentially if there are multiple instances of Series, Game or whatever else Media may hold in the future, the expression guarantees that they will only appear once in the output. The text wrapped in the <xsl:text> elements are tab and newline characters to properly format the output.



Shown above is the XSLT used to match the Series Element. The unique operation in this template is the use of the translate function. This allows the Series titles to be transformed into a valid URI. Since Titles can contain spaces, they automatically are non-valid URIs, so the translate command replaces all the space with underscores, which seems to be the standard for dealing with spaces when creating URIs. This is also applied to the names of all the Games contained in Series.

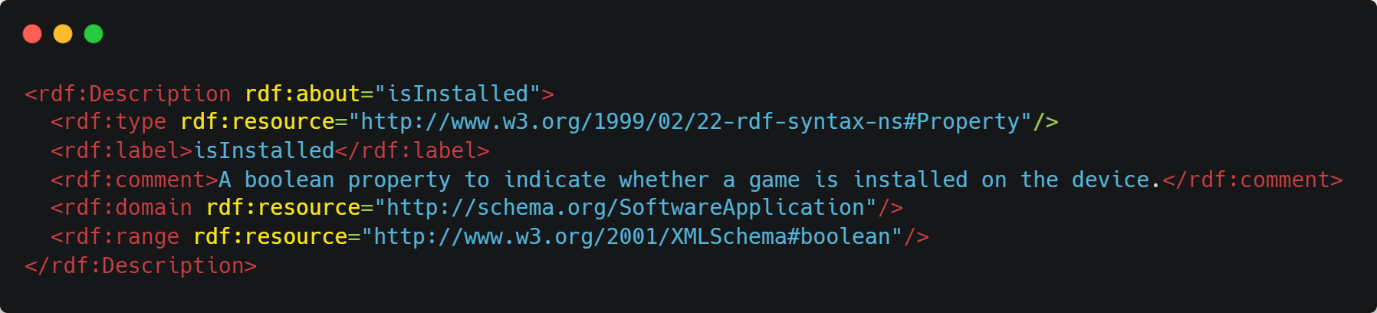


Shown above is the XSLT used when matching a Game element. The majority of it is similar to the previous templates shown, however one difference is in the use of the <xsl:if> element. As the name suggests, this allows us to conditionally execute statements. Since the Game element can have either PayPal or Card payment and can contain either designerName, which contains the first and last name of the designer, or designerNameExtended, which contains the first, last and middle name of the designer, I needed a way to test if these elements were present and then process them accordingly. I also make use of the “concat()” function provided by XSLT, to concatenate the first, middle and last names of the designer together.

# Question 1d

Resource Description Framework (RDF) is vocabulary created by W3C that allows for the encoding, the exchange and the understanding of information in a way that can be understood by machines. It was developed as part of the Semantic Web Imitative and has its first draft published in October of 1997 [1]. From this initial introduction of RDF, derivatives such as RDF Schema (RDFS) and Web Ontology Language (OWL) were created to help further the goal of intelligent structured processing of information. Before the semantic web, textual data was available on the web, but they could not be easily understood by machines. For example, if I had a webpage describing my family history, and I wanted to query whether a certain person was my mother, a machine could probably answer the question by making a guess based on what was written in the document, but it can’t be sure. RDF, RDFS and OWL, allow us to structure data in such as way so that a machine can firstly understand what a mother is, and what it means for someone to be a mother. I can also explicitly define relationships using the “subject predicate object” idea, e.g. Mary isTheMotherOf John. Provided we define what the “isTheMotherOf” predicate is, when we ask a query about the mother of John, the computer can use this relationship to say that Mary is John’s mother. OWL extends on this basic idea of relationships in RDF by implementing description logic which allows computers to not only reason about certain fragments of knowledge but also to provide proofs as to how it generated an answer in response a question.

Illustrated in my RDF is the ability to define your own properties when a comparable one is not available online. For example, the “isInstalled” element which indicated whether a game is installed or not, could not be found in any publicly available ontology, so I defined it myself. The definition is shown in the screenshot below:



Also shown is the ability in RDFS to place restrictions on properties, so the isInstalled property can only be applied to something that is a software application and it can only contain data that is a Boolean (indicated by the domain and range tags).

Although not demonstrated, RDFS allows for the ability to logically organise classes in hierarchies by using the “subClassOf” term. So, one could say that “<Father, subClassOf, Person>” to imply that some of the properties and rules associated with Person could also be applied to Father since he is also a person.

Some limitations of RDFS include its limited expressivity. RDFS does not support cardinality restrictions, I cannot say that a game should only have one title, or one price. RDFS also does not have any localized range or domain constraints for properties. For example, I cannot say that the hasChild property range should be a person when applied to a person class and an animal when applied to an animal class. In RDFS it is entire valid to have an animal be a child of a person. In RDFS we can’t have inverse, symmetrical or transitive properties, which limit our ability to reason and infer knowledge. For example if we defined a “greaterThan” property in RDFS and we say that “A greaterThan B” and “B greaterThan C”, the computer cannot make any inferences about the relationship between A and C since there is no concept of transitivity. But in OWL, where we can define transitive properties, an OWL reasoner could infer that based on the knowledge provided, A must be greater than C. This lack of property types means that RDFS can only answer questions based on the knowledge defined, where as in OWL, it can infer relationships based on the knowledge provided, even if the relationship isn’t explicitly defined.

# Question 1e

For this question, I was asked to illustrate all the ways in which I used the XPATH specification to select different elements in my XSLT.

## Node Property Selection

XPath allows us to select different properties of a node like the name of the node, or the text contained within it through the use of the name() and text() functions. I used the name() function in my XSLT to select the names of the child nodes for the <dcterms:hasPart> tag as shown below:



The text() function was used to select the data contain within each node as shown below:

A black screen with blue and red text

Description automatically generated

## Path Traversal

Another feature of XPath is that it allows us to specify a path to walk down the hierarchy tree in the document. The path is structures as the names of the nodes separated by “/”. So, for example the path “/Library/Owner” would mean select the child node named Library and then from its children select the Owner tag. This can be seen in the code screenshot below



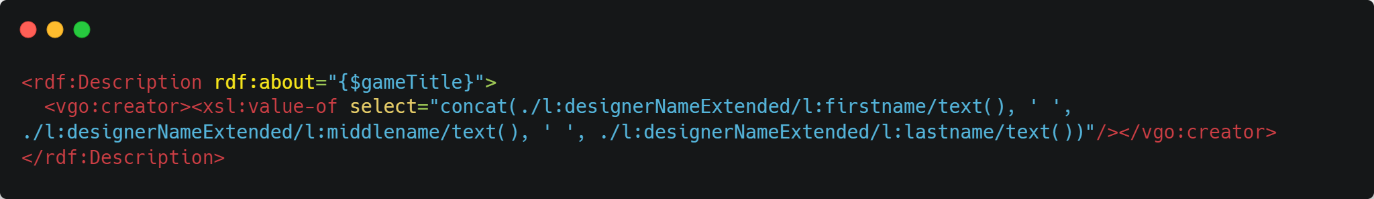
## Translation

Another feature of XPath that I demonstrated in my XSLT is the translate function. This was used to transform the titles of Games and Series into valid URIs. This was done by providing the translate function with the string that I wanted to transform and the character which I wanted to replace, in this case spaces with underscores. This is demonstrated in the code screenshot below:



## Concatenate

XPath also provides you with the ability to concatenate multiple strings together. This was used to concatenate the designer names together so they would be one string



## Boolean Testing

The paths in XPath can be used to test whether an element exists or not in conjunction with an if statement from XSLT. I used this in my XSLT to conditionally perform actions depending on the presence of a tag:



Here I provide the if statement with the path to a potential paypal tag from the current node. If it exists, the path will evaluate to an object passing the if test, otherwise it will evaluate to nothing.

## Wildcard Selection

XPath also allows us to use the “\*” wildcard identifier to match everything at a certain node level. So if one were to provide the “./\*” XPath, it would mean match all the children of the current node. I used this to specify the different parts that are contained in the Game element.



# References

[1] ‘Resource Description Framework (RDF) Model and Syntax’. [Online]. Available: https://www.w3.org/TR/WD-rdf-syntax-971002/