String tokenizing wrap-up and DUML translator design

Read Chapter 6
StringTokenizers are great for splitting strings

```java
StringTokenizer tokenizer =
    new StringTokenizer( "This is a sentence." );

while ( tokenizer.hasMoreTokens() )
{
    System.out.println( tokenizer.nextToken() );
}

Output (4 "words"):
This
is
a
Sentence.
```
Let’s add to the SpellChecker class

- Using a `StringTokenizer`, we can break lines up into words and spell-check each word ...
You can control the delimiters that define words

- By default, a `StringTokenizer` uses any whitespace character as a delimiter (spaces, tabs, newlines, etc.)
- You can provide your own set of delimiter characters that mark word boundaries instead:

```java
StringTokenizer tokenizer = new StringTokenizer("Hey--too! Much punctuation? For ‘you’?", " ,.;:-+=[]{}|?<>'\"!");

while ( tokenizer.hasMoreTokens() )
{
    System.out.println( tokenizer.nextToken() );
}
```

- Skips over sequences of delimiter characters
The DUML rules are simple

- A **word** shall contain at least two characters in order for its first and/or last character to be interpreted as a DUML tag.
- The **first** character shall be a starting tag if the word is not nested within another tagged phrase.
- The **last** shall be an ending tag if it occurs within a phrase marked with the matching starting tag.
- Tags are *, ^, =, and ().
A DUML Translator operates in one of two modes

Start

Looking for a start tag

Tag start char

Matching tag end char

Looking for an end tag
On closer inspection, there are actually four modes.
We will discuss three translator strategies

- A basic `if/else if` implementation in one method (bad)
- A switch statement with separate cases in support methods (a little better)
- An OO design using separate support classes—using polymorphism instead of if statements
The “simplest” idea is just to use a big if statement

```java
int thisChar = in.read();
while ( thisChar != -1 )
{
    if ( thisChar == '*' )
    { // ...
    }
    else if ( thisChar == '^' )
    { // ...
    }
    else if ( thisChar == '=' )
    { // ...
    }
    else if ( thisChar == '(' )
    { // ...
    }
    else
    { // ...
    }
} // Major problems!
```
Alternatively, we can break it into pieces

- The most obvious way to break such a solution up into pieces is to place each possibility (each case) in a separate support method.

- Java provides an alternative to the if statement called a **switch statement** (also called a case statement)

```java
switch ( value )
{
    case '*':
        // do one thing
        break;
    case '^':
        // do another thing
        break;
    default:
        // do something else
        break;
}
```
What if we replace the switch with polymorphism?

- The previous approach still isn’t very OO—instead, it is a good “procedural” solution (you could do it in Pascal or C)

- Instead of using a separate `method` for each state, and then using a `switch statement` to control which one we execute …

- We could use a separate `object` to represent each state, each with a different implementation of a `translateCharacter()` method …

- … And then use `polymorphism` to take the correct action based on the “current state”