Regifying a Search String

Match Whole Word

If the GUI has a feature where the user can ask to only find results that land on word boundaries, the regular expression “**\b**” is added to the front and back of the search string.

```
        Original = “foo bar”
        Regified = “\bfoo bar\b”
```

Whitespace Flexibility

If the search string has a whitespace character in it, replace the whitespace character with the regular expression “**\s***” An expression that means “zero (or more) whitespace characters.” The inclusion of zero as a valid option is done because sometimes a string ***looks*** like it has spaces, but the actual text does not.

```
        Original = "foo bar"
        Regified = “foo\s*bar”
```

Between Two ~~Ferns~~ Alphabetic Characters

If the search string has two alphabetic characters in a row in it, insert the regular expression “**(\s*-\s*)?**” between them. The expression means “There may (or not) be a hyphen (dash) between these characters. Also, there may be (or not) whitespace around the hyphen.” This helps to find results that wrap from the end of one line to the beginning of the next. With a hyphen character interrupting the break.

```
        Original = "foo bar"
        Regified = “f(\s*-\s*)?o(\s*-\s*)?o b(\s*-\s*)?a(\s*-\s*)?r”
```

Regex Safety

If the search string has characters that mean something in regex, make them safe. This is done by looking for the regex-specific characters and adding an escape character “\” (backslash) in front of them.

```
if (character === "+" ||
    character === "\" ||
    character === "." ||
    character === "*" ||
    character === "?" ||
    character === "^" ||
    character === "$" ||
    character === "[" ||
    character === "]" ||
    character === "(" ||
    character === ")" ||
    character === "|" ||
    character === "{" ||
    character === "}" ||
    character === "/" ||
    character === "#") {
   // add "\" in front of it to make it literal
}
```

```
       Original = "foo\bar"
       Regified = “foo\\bar”
```

Consolidate Versions of Specific Punctuation Characters

If the search string contains one of various punctuation characters, replace it with a regular expression describing a defined set of common alternate forms of that punctuation character. This often happens with Microsoft Office documents.

The user types one thing, say **"** and the Microsoft editor inserts either **“** or **”** (which are different Unicode characters than the original ASCII quote character)

Here are the sets:

```
       Comma = “[,\u00A9]”
       Dash = “[-\u2015\u00ad\u0097\u0096]”
       Apostrophe = “['\u2018\u2019\u00b4\u0092]”
       Quote = “["\u201c\u201d\u0093\u0094]”
```

So, if any character in the set are encountered, replace it will the whole set>

```
       Original = "foo-bar"
       Regified = “foo[-\u2015\u00ad\u0097\u0096]bar”
```

In the previous section about hyphenation code between alphabetic characters, it should probably replace the dash character with “[-\u2015\u00ad\u0097\u0096]”

So that insertion should probably be **“(\s*[-\u2015\u00ad\u0097\u0096]\s*)?”**

C++ Code and Office vs CAD Considerations

In CDL, the Regify code makes a distinction between a document that is classified as an “Office” document versus a CAD drawing. I did not include those distinctions here since I don’t think that the other viewers make this distinction. Also, I am not sure that – in this case – the distinction is all that great.

In any case, here is the actual C++ code CDL uses. If you are able to parse it and have questions, feel free to email questions to me or Ken for clarity.

```
void CDLRedlineableController::RegifyString(bool bWordBoundaries,
                                             const DLString &PlainString,
                                             DLString &RegifiedString)
{
   bool officedoc = true;
   if(m_pView != NULL &&
      m_pView->GetIDoc() != NULL)
   {
      if(m_pView->GetIDoc()->IsCADDrawing(GetCurrentPage()))
      {
         officedoc = false;
      }
   }

   const DLChar *pBegin = PlainString.Str();
   const DLChar *pWork = pBegin;
   const DLChar *pEnd = pWork + PlainString.Length();
   RegifiedString = L"";

   if(bWordBoundaries)
   {
      RegifiedString += L"\\b";
   }

   while(pWork < pEnd)
   {
      if(iswspace(*pWork))
      {
         if(officedoc)
         {
            // replace spaces with "\s*"
            RegifiedString += L"\\s*";
         }
         else
         {
            RegifiedString += L"\\s";
         }
      }
      else
      {
         if(pWork > pBegin && officedoc)
         {
            if(!iswspace(*(pWork-1)))
            {
               if(iswlatinalpha(*(pWork-1)) &&
                  iswlatinalpha(*pWork) )
               {
                  // insert hyphen possibilities in between alpha chars
                  RegifiedString += L"(\\s*-\\s*)?";
               }
               else
               {
                  // insert whitespace possibilities in between digits
                  // or alpha and ditit chars
                  RegifiedString += L"\\s*";
               }
            }
```

```
}

// if the char is a regex command char, escape it.
if(*pWork == L'+' ||
   *pWork == L'\\' ||
   *pWork == L'.' ||
   *pWork == L'*' ||
   *pWork == L'?' ||
   *pWork == L'^' ||
   *pWork == L'$' ||
   *pWork == L'[' ||
   *pWork == L']' ||
   *pWork == L'(' ||
   *pWork == L')' ||
   *pWork == L'|' ||
   *pWork == L'{' ||
   *pWork == L'}' ||
   *pWork == L'/' ||
   *pWork == L'#'){RegifiedString += L"\\";}

bool charAdd = false;

switch(*pWork)
{
   case L',':
   case (DLChar)0x0082:
   {
      RegifiedString += L"[,";
      RegifiedString += (DLChar)0x0082;
      RegifiedString += L']';
      charAdd = true;
   }
   break;
   case L'-':
   case (DLChar)0x2015:
   case (DLChar)0x00ad:
   case (DLChar)0x0096:
   case (DLChar)0x0097:

   {
      RegifiedString += L"[-";
      RegifiedString += (DLChar)0x2015;
      RegifiedString += (DLChar)0x00ad;
      RegifiedString += (DLChar)0x0096;
      RegifiedString += (DLChar)0x0097;
      RegifiedString += L']';
      charAdd = true;
   }
   break;
   case L'\'':
   case (DLChar)0x2018:
   case (DLChar)0x2019:
   case (DLChar)0x00b4:
   case (DLChar)0x0092:
   {
      RegifiedString += L"[\'";
      RegifiedString += (DLChar)0x2018;   // left quote
      RegifiedString += (DLChar)0x2019;   // right quote
```

```
            RegifiedString += (DLChar)0x00b4;
            RegifiedString += (DLChar)0x0092;
            RegifiedString += L']';
            charAdd = true;
         }
         break;
         case L'\"':
         case (DLChar)0x201c:
         case (DLChar)0x201d:
         case (DLChar)0x0093:
         case (DLChar)0x0094:
         {
            RegifiedString += L"[\"";
            RegifiedString += (DLChar)0x201c;   // left double quote
            RegifiedString += (DLChar)0x201d;   // right double quote
            RegifiedString += (DLChar)0x0093;
            RegifiedString += (DLChar)0x0094;
            RegifiedString += L']';
            charAdd = true;
         }
         break;
         default:
            break;
      }
      if(!charAdd)
      {
         // assign the char
         RegifiedString += *pWork;
      }
   }

   pWork++;
}
```