src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
/*******************************************************************************
 * Copyright (c) 2000, 2006 IBM Corporation and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 * Contributors:
 *     IBM Corporation - initial API and implementation
 *******************************************************************************/
package org.eclipse.ui.internal.ide;

import java.util.Vector;

/**
 * A string pattern matcher suppporting &amp;#39;*&amp;#39; and &amp;#39;?&amp;#39; wildcards.
 */
public class StringMatcher {
    protected String fPattern;

    protected int fLength; // pattern length

    protected boolean fIgnoreWildCards;

    protected boolean fIgnoreCase;

    protected boolean fHasLeadingStar;

    protected boolean fHasTrailingStar;

    protected String fSegments[]; //the given pattern is split into * separated segments

    /* boundary value beyond which we don't need to search in the text */
    protected int fBound = 0;

    protected static final char fSingleWildCard = '¥u0000';

    public static class Position {
        int start; //inclusive

        int end; //exclusive

        public Position(int start, int end) {
            this.start = start;
            this.end = end;
        }

        public int getStart() {
            return start;
        }

        public int getEnd() {
            return end;
        }
    }
```

```
/*******************************************************************************
 * Copyright (c) 2000, 2006 IBM Corporation and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 * Contributors:
 *     IBM Corporation - initial API and implementation
 *******************************************************************************/
package org.eclipse.ui.views.navigator;

import java.util.Vector;

/**
 * A string pattern matcher, suppporting ?*? and ??? wildcards.
 */
/* package */class StringMatcher {
    protected String fPattern;

    protected int fLength; // pattern length

    protected boolean fIgnoreWildCards;

    protected boolean fIgnoreCase;

    protected boolean fHasLeadingStar;

    protected boolean fHasTrailingStar;

    protected String fSegments[]; //the given pattern is split into * separated segments

    /* boundary value beyond which we don't need to search in the text */
    protected int fBound = 0;

    protected static final char fSingleWildCard = '¥u0000';

    public static class Position {
        int start; //inclusive

        int end; //exclusive

        public Position(int start, int end) {
            this.start = start;
            this.end = end;
        }

        public int getStart() {
            return start;
        }

        public int getEnd() {
            return end;
        }
    }
```

src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
56
57      /**
58       * StringMatcher constructor takes in a String object that is a simple
59       * pattern which may contain &#39;*&#39; for 0 and many characters and
60       * &#39;&#63;&#39; for exactly one character.
61       *
62       * Literal &#39;*&#39; and &#39;*&#39; characters must be escaped in the pattern
63       * e.g. &quot;&#92;*&quot; means literal &quot;*&quot;, etc.
64       *
65       * Escaping any other character (including the escape character itself),
66       * just results in that character in the pattern.
67       * e.g. &quot;&#92;a&quot; means &quot;a&quot; and &quot;&#92;&#92;&quot; means &quot;&#92;&quot;
68       *
69       * If invoking the StringMatcher with string literals in Java, don't forget
70       * escape characters are represented by &quot;&#92;&#92;&quot;.
71       *
72       * @param pattern the pattern to match text against
73       * @param ignoreCase if true, case is ignored
74       * @param ignoreWildCards if true, wild cards and their escape sequences are ignored
75       *        (everything is taken literally).
76       */
77      public StringMatcher(String pattern, boolean ignoreCase,
78              boolean ignoreWildCards) {
79          if (pattern == null) {
80              throw new IllegalArgumentException();
81          }
82          fIgnoreCase = ignoreCase;
83          fIgnoreWildCards = ignoreWildCards;
84          fPattern = pattern;
85          fLength = pattern.length();
86
87          if (fIgnoreWildCards) {
88              parseNoWildCards();
89          } else {
90              parseWildCards();
91          }
92      }
93
94      /**
95       * Find the first occurrence of the pattern between <code>start</code>(inclusive)
96       * and <code>end</code>(exclusive).
97       * @param text  the String object to search in
98       * @param start  the starting index of the search range, inclusive
99       * @param end  the ending index of the search range, exclusive
100      * @return an <code>StringMatcher.Position</code> object that keeps the starting
101      * (inclusive) and ending positions (exclusive) of the first occurrence of the
102      * pattern in the specified range of the text; return null if not found or subtext
103      * is empty (start==end). A pair of zeros is returned if pattern is empty string
104      * Note that for pattern like &quot;*abc*&quot; with leading and trailing
```

```
56
57      /**
58       * StringMatcher constructor takes in a String object that is a simple
59       * pattern which may contain '*' for 0 and many characters and
60       * '?' for exactly one character.
61       *
62       * Literal '*' and '?' characters must be escaped in the pattern
63       * e.g., "¥*" means literal "*", etc.
64       *
65       * Escaping any other character (including the escape character itself),
66       * just results in that character in the pattern.
67       * e.g., "¥a" means "a" and "¥¥" means "¥"
68       *
69       * If invoking the StringMatcher with string literals in Java, don't forget
70       * escape characters are represented by "¥¥".
71       *
72       * @param pattern the pattern to match text against
73       * @param ignoreCase if true, case is ignored
74       * @param ignoreWildCards if true, wild cards and their escape sequences are ignored
75       *        (everything is taken literally).
76       */
77      public StringMatcher(String pattern, boolean ignoreCase,
78              boolean ignoreWildCards) {
79          if (pattern == null) {
80              throw new IllegalArgumentException();
81          }
82          fIgnoreCase = ignoreCase;
83          fIgnoreWildCards = ignoreWildCards;
84          fPattern = pattern;
85          fLength = pattern.length();
86
87          if (fIgnoreWildCards) {
88              parseNoWildCards();
89          } else {
90              parseWildCards();
91          }
92      }
93
94      /**
95       * Find the first occurrence of the pattern between <code>start</code>(inclusive)
96       * and <code>end</code>(exclusive).
97       * @param <code>text</code>, the String object to search in
98       * @param <code>start</code>, the starting index of the search range, inclusive
99       * @param <code>end</code>, the ending index of the search range, exclusive
100      * @return an <code>StringMatcher.Position</code> object that keeps the starting
101      * (inclusive) and ending positions (exclusive) of the first occurrence of the
102      * pattern in the specified range of the text; return null if not found or subtext
103      * is empty (start==end). A pair of zeros is returned if pattern is empty string
104      * Note that for pattern like "*abc*" with leading and trailing stars,
```

src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
104stars, position of &quot;abc&quot;
105     * is returned. For a pattern like&quot;*&#63;&#63;*&quot; in text
  &quot;abcdf&quot;, (1,3) is returned
106     */
107    public StringMatcher.Position find(String text, int start, int end) {
108        if (text == null) {
109            throw new IllegalArgumentException();
110        }
111
112        int tlen = text.length();
113        if (start < 0) {
114            start = 0;
115        }
116        if (end > tlen) {
117            end = tlen;
118        }
119        if (end < 0 || start >= end) {
120            return null;
121        }
122        if (fLength == 0) {
123            return new Position(start, start);
124        }
125        if (fIgnoreWildCards) {
126            int x = posIn(text, start, end);
127            if (x < 0) {
128                return null;
129            }
130            return new Position(x, x + fLength);
131        }
132
133        int segCount = fSegments.length;
134        if (segCount == 0) {
135            return new Position(start, end);
136        }
137
138        int curPos = start;
139        int matchStart = -1;
140        int i;
141        for (i = 0; i < segCount && curPos < end; ++i) {
142            String current = fSegments[i];
143            int nextMatch = regExpPosIn(text, curPos, end, current);
144            if (nextMatch < 0) {
145                return null;
146            }
147            if (i == 0) {
148                matchStart = nextMatch;
149            }
150            curPos = nextMatch + current.length();
151        }
152        if (i < segCount) {
153            return null;
154        }
155        return new Position(matchStart, curPos);
156    }
157
158    /**
```

```
104position of "abc"
105     * is returned. For a pattern like"*??*" in text "abcdf", (1,3) is returned
106     */
107    public StringMatcher.Position find(String text, int start, int end) {
108        if (text == null) {
109            throw new IllegalArgumentException();
110        }
111
112        int tlen = text.length();
113        if (start < 0) {
114            start = 0;
115        }
116        if (end > tlen) {
117            end = tlen;
118        }
119        if (end < 0 || start >= end) {
120            return null;
121        }
122        if (fLength == 0) {
123            return new Position(start, start);
124        }
125        if (fIgnoreWildCards) {
126            int x = posIn(text, start, end);
127            if (x < 0) {
128                return null;
129            }
130            return new Position(x, x + fLength);
131        }
132
133        int segCount = fSegments.length;
134        if (segCount == 0) {
135            return new Position(start, end);
136        }
137
138        int curPos = start;
139        int matchStart = -1;
140        int i;
141        for (i = 0; i < segCount && curPos < end; ++i) {
142            String current = fSegments[i];
143            int nextMatch = regExpPosIn(text, curPos, end, current);
144            if (nextMatch < 0) {
145                return null;
146            }
147            if (i == 0) {
148                matchStart = nextMatch;
149            }
150            curPos = nextMatch + current.length();
151        }
152        if (i < segCount) {
153            return null;
154        }
155        return new Position(matchStart, curPos);
156    }
157
158    /**
```

src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
159     * match the given <code>text</code> with the pattern
160     * @return true if matched eitherwise false
161     * @param text  a String object
162     */
163    public boolean match(String text) {
164        return match(text, 0, text.length());
165    }
166
167    /**
168     * Given the starting (inclusive) and the ending (exclusive) positions in the
169     * <code>text</code>, determine if the given substring matches with aPattern
170     * @return true if the specified portion of the text matches the pattern
171     * @param text  a <code>String</code> object that contains the substring to match
172     * @param start  marks the starting position (inclusive) of the substring
173     * @param end  marks the ending index (exclusive) of the substring
174     */
175    public boolean match(String text, int start, int end) {
176        if (null == text) {
177            throw new IllegalArgumentException();
178        }
179
180        if (start > end) {
181            return false;
182        }
183
184        if (fIgnoreWildCards) {
185            return (end - start == fLength)
186                    && fPattern.regionMatches(fIgnoreCase, 0, text, start,
187                            fLength);
188        }
189        int segCount = fSegments.length;
190        if (segCount == 0 && (fHasLeadingStar || fHasTrailingStar)) {
191            return true;
192        }
193        if (start == end) {
194            return fLength == 0;
195        }
196        if (fLength == 0) {
197            return start == end;
198        }
199
200        int tlen = text.length();
201        if (start < 0) {
202            start = 0;
203        }
204        if (end > tlen) {
205            end = tlen;
206        }
207
208        int tCurPos = start;
209        int bound = end - fBound;
210        if (bound < 0) {
```

```
159     * match the given <code>text</code> with the pattern
160     * @return true if matched eitherwise false
161     * @param <code>text</code>, a String object
162     */
163    public boolean match(String text) {
164        return match(text, 0, text.length());
165    }
166
167    /**
168     * Given the starting (inclusive) and the ending (exclusive) poisitions in the
169     * <code>text</code>, determine if the given substring matches with aPattern
170     * @return true if the specified portion of the text matches the pattern
171     * @param String <code>text</code>, a String object that contains the substring to match
172     * @param int <code>start<code> marks the starting position (inclusive) of the substring
173     * @param int <code>end<code> marks the ending index (exclusive) of the substring
174     */
175    public boolean match(String text, int start, int end) {
176        if (null == text) {
177            throw new IllegalArgumentException();
178        }
179
180        if (start > end) {
181            return false;
182        }
183
184        if (fIgnoreWildCards) {
185            return (end - start == fLength)
186                    && fPattern.regionMatches(fIgnoreCase, 0, text, start,
187                            fLength);
188        }
189        int segCount = fSegments.length;
190        if (segCount == 0 && (fHasLeadingStar || fHasTrailingStar)) {
191            return true;
192        }
193        if (start == end) {
194            return fLength == 0;
195        }
196        if (fLength == 0) {
197            return start == end;
198        }
199
200        int tlen = text.length();
201        if (start < 0) {
202            start = 0;
203        }
204        if (end > tlen) {
205            end = tlen;
206        }
207
208        int tCurPos = start;
209        int bound = end - fBound;
210        if (bound < 0) {
```

src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
211             return false;
212         }
213         int i = 0;
214         String current = fSegments[i];
215         int segLength = current.length();
216 
217         /* process first segment */
218         if (!fHasLeadingStar) {
219             if (!regExpRegionMatches(text, start, current, 0, segLength)) {
220                 return false;
221             } else {
222                 ++i;
223                 tCurPos = tCurPos + segLength;
224             }
225         }
226         if ((fSegments.length == 1) && (!fHasLeadingStar)
227                 && (!fHasTrailingStar)) {
228             // only one segment to match, no wildcards specified
229             return tCurPos == end;
230         }
231         /* process middle segments */
232         while (i < segCount) {
233             current = fSegments[i];
234             int currentMatch;
235             int k = current.indexOf(fSingleWildCard);
236             if (k < 0) {
237                 currentMatch = textPosIn(text, tCurPos, end, current);
238                 if (currentMatch < 0) {
239                     return false;
240                 }
241             } else {
242                 currentMatch = regExpPosIn(text, tCurPos, end, current);
243                 if (currentMatch < 0) {
244                     return false;
245                 }
246             }
247             tCurPos = currentMatch + current.length();
248             i++;
249         }
250 
251         /* process final segment */
252         if (!fHasTrailingStar && tCurPos != end) {
253             int clen = current.length();
254             return regExpRegionMatches(text, end - clen, current, 0, clen);
255         }
256         return i == segCount;
257     }
258 
259     /**
260      * This method parses the given pattern into segments seperated by wildcard '*' characters.
261      * Since wildcards are not being used in this case, the pattern consists of a single segment.
262      */
263     private void parseNoWildCards() {
264         fSegments = new String[1];
```

```
211             return false;
212         }
213         int i = 0;
214         String current = fSegments[i];
215         int segLength = current.length();
216 
217         /* process first segment */
218         if (!fHasLeadingStar) {
219             if (!regExpRegionMatches(text, start, current, 0, segLength)) {
220                 return false;
221             } else {
222                 ++i;
223                 tCurPos = tCurPos + segLength;
224             }
225         }
226         if ((fSegments.length == 1) && (!fHasLeadingStar)
227                 && (!fHasTrailingStar)) {
228             // only one segment to match, no wildcards specified
229             return tCurPos == end;
230         }
231         /* process middle segments */
232         for (; i < segCount && tCurPos <= bound; ++i) {
233             current = fSegments[i];
234             int currentMatch;
235             int k = current.indexOf(fSingleWildCard);
236             if (k < 0) {
237                 currentMatch = textPosIn(text, tCurPos, end, current);
238                 if (currentMatch < 0) {
239                     return false;
240                 }
241             } else {
242                 currentMatch = regExpPosIn(text, tCurPos, end, current);
243                 if (currentMatch < 0) {
244                     return false;
245                 }
246             }
247             tCurPos = currentMatch + current.length();
248         }
249 
250         /* process final segment */
251         if (!fHasTrailingStar && tCurPos != end) {
252             int clen = current.length();
253             return regExpRegionMatches(text, end - clen, current, 0, clen);
254         }
255         return i == segCount;
256     }
257 
258     /**
259      * This method parses the given pattern into segments seperated by wildcard '*' characters.
260      * Since wildcards are not being used in this case, the pattern consists of a single segment.
261      */
262     private void parseNoWildCards() {
263         fSegments = new String[1];
```

src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
265         fSegments[0] = fPattern;
266         fBound = fLength;
267     }
268
269     /**
270      * Parses the given pattern into segments seperated by wildcard &#39;*&#39; characters.
271      * @param p, a String object that is a simple regular expression with ?*? and/or &#39;&#63;&#39;
272      */
273     private void parseWildCards() {
274         if (fPattern.startsWith("*")) { //$NON-NLS-1$
275             fHasLeadingStar = true;
276         }
277         if (fPattern.endsWith("*")) {//$NON-NLS-1$
278             /* make sure it's not an escaped wildcard */
279             if (fLength > 1 && fPattern.charAt(fLength - 2) != '¥¥') {
280                 fHasTrailingStar = true;
281             }
282         }
283
284         Vector temp = new Vector();
285
286         int pos = 0;
287         StringBuffer buf = new StringBuffer();
288         while (pos < fLength) {
289             char c = fPattern.charAt(pos++);
290             switch (c) {
291             case '¥¥':
292                 if (pos >= fLength) {
293                     buf.append(c);
294                 } else {
295                     char next = fPattern.charAt(pos++);
296                     /* if it's an escape sequence */
297                     if (next == '*' || next == '?' || next == '¥¥') {
298                         buf.append(next);
299                     } else {
300                         /* not an escape sequence, just insert literally */
301                         buf.append(c);
302                         buf.append(next);
303                     }
304                 }
305                 break;
306             case '*':
307                 if (buf.length() > 0) {
308                     /* new segment */
309                     temp.addElement(buf.toString());
310                     fBound += buf.length();
311                     buf.setLength(0);
312                 }
313                 break;
314             case '?':
315                 /* append special character representing single match wildcard */
316                 buf.append(fSingleWildCard);
317                 break;
318             default:
```

```
264         fSegments[0] = fPattern;
265         fBound = fLength;
266     }
267
268     /**
269      * Parses the given pattern into segments seperated by wildcard '*' characters.
270      * @param p, a String object that is a simple regular expression with '*' and/or '?'
271      */
272     private void parseWildCards() {
273         if (fPattern.startsWith("*")) { //$NON-NLS-1$
274             fHasLeadingStar = true;
275         }
276         if (fPattern.endsWith("*")) {//$NON-NLS-1$
277             /* make sure it's not an escaped wildcard */
278             if (fLength > 1 && fPattern.charAt(fLength - 2) != '¥¥') {
279                 fHasTrailingStar = true;
280             }
281         }
282
283         Vector temp = new Vector();
284
285         int pos = 0;
286         StringBuffer buf = new StringBuffer();
287         while (pos < fLength) {
288             char c = fPattern.charAt(pos++);
289             switch (c) {
290             case '¥¥':
291                 if (pos >= fLength) {
292                     buf.append(c);
293                 } else {
294                     char next = fPattern.charAt(pos++);
295                     /* if it's an escape sequence */
296                     if (next == '*' || next == '?' || next == '¥¥') {
297                         buf.append(next);
298                     } else {
299                         /* not an escape sequence, just insert literally */
300                         buf.append(c);
301                         buf.append(next);
302                     }
303                 }
304                 break;
305             case '*':
306                 if (buf.length() > 0) {
307                     /* new segment */
308                     temp.addElement(buf.toString());
309                     fBound += buf.length();
310                     buf.setLength(0);
311                 }
312                 break;
313             case '?':
314                 /* append special character representing single match wildcard */
315                 buf.append(fSingleWildCard);
316                 break;
317             default:
```

src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
319                 buf.append(c);
320             }
321         }
322
323         /* add last buffer to segment list */
324         if (buf.length() > 0) {
325             temp.addElement(buf.toString());
326             fBound += buf.length();
327         }
328
329         fSegments = new String[temp.size()];
330         temp.copyInto(fSegments);
331     }
332
333     /**
334      * @param text  a string which contains no wildcard
335      * @param start  the starting index in the text for search, inclusive
336      * @param end  the stopping point of search, exclusive
337      * @return the starting index in the text of the pattern , or -1 if not found
338      */
339     protected int posIn(String text, int start, int end) {//no wild card in pattern
340         int max = end - fLength;
341
342         if (!fIgnoreCase) {
343             int i = text.indexOf(fPattern, start);
344             if (i == -1 || i > max) {
345                 return -1;
346             }
347             return i;
348         }
349
350         for (int i = start; i <= max; ++i) {
351             if (text.regionMatches(true, i, fPattern, 0, fLength)) {
352                 return i;
353             }
354         }
355
356         return -1;
357     }
358
359     /**
360      * @param text  a simple regular expression that may only contain '&#63;'(s)
361      * @param start  the starting index in the text for search, inclusive
362      * @param end  the stopping point of search, exclusive
363      * @param p  a simple regular expression that may contains '&#63;'
364      * @return the starting index in the text of the pattern , or -1 if not found
365      */
366     protected int regExpPosIn(String text, int start, int end, String p) {
367         int plen = p.length();
368
369         int max = end - plen;
370         for (int i = start; i <= max; ++i) {
```

```
318                 buf.append(c);
319             }
320         }
321
322         /* add last buffer to segment list */
323         if (buf.length() > 0) {
324             temp.addElement(buf.toString());
325             fBound += buf.length();
326         }
327
328         fSegments = new String[temp.size()];
329         temp.copyInto(fSegments);
330     }
331
332     /**
333      * @param <code>text</code>, a string which contains no wildcard
334      * @param <code>start</code>, the starting index in the text for search,
    inclusive
335      * @param <code>end</code>, the stopping point of search, exclusive
336      * @return the starting index in the text of the pattern , or -1 if not found
337      */
338     protected int posIn(String text, int start, int end) {//no wild card in pattern
339         int max = end - fLength;
340
341         if (!fIgnoreCase) {
342             int i = text.indexOf(fPattern, start);
343             if (i == -1 || i > max) {
344                 return -1;
345             }
346             return i;
347         }
348
349         for (int i = start; i <= max; ++i) {
350             if (text.regionMatches(true, i, fPattern, 0, fLength)) {
351                 return i;
352             }
353         }
354
355         return -1;
356     }
357
358     /**
359      * @param <code>text</code>, a simple regular expression that may only contain
    '?'(s)
360      * @param <code>start</code>, the starting index in the text for search,
    inclusive
361      * @param <code>end</code>, the stopping point of search, exclusive
362      * @param <code>p</code>, a simple regular expression that may contains '?'
363      * @param <code>caseIgnored</code>, wether the pattern is not casesensitive
364      * @return the starting index in the text of the pattern , or -1 if not found
365      */
366     protected int regExpPosIn(String text, int start, int end, String p) {
367         int plen = p.length();
368
369         int max = end - plen;
370         for (int i = start; i <= max; ++i) {
```

src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
371             if (regExpRegionMatches(text, i, p, 0, plen)) {
372                 return i;
373             }
374         }
375         return -1;
376     }
377 
378     /**
379      *
380      * @return boolean
381      * @param  text  a String to match
382      * @param tStart  int that indicates the starting index of match, inclusive
383      * @param p  String,  String, a simple regular expression that may contain '&#63;'
384      * @param pStart
385      * @param plen
386      */
387     protected boolean regExpRegionMatches(String text, int tStart, String p,
388             int pStart, int plen) {
389         while (plen-- > 0) {
390             char tchar = text.charAt(tStart++);
391             char pchar = p.charAt(pStart++);
392 
393             /* process wild cards */
394             if (!fIgnoreWildCards) {
395                 /* skip single wild cards */
396                 if (pchar == fSingleWildCard) {
397                     continue;
398                 }
399             }
400             if (pchar == tchar) {
401                 continue;
402             }
403             if (fIgnoreCase) {
404                 if (Character.toUpperCase(tchar) == Character
405                         .toUpperCase(pchar)) {
406                     continue;
407                 }
408                 // comparing after converting to upper case doesn't handle all cases;
409                 // also compare after converting to lower case
410                 if (Character.toLowerCase(tchar) == Character
411                         .toLowerCase(pchar)) {
412                     continue;
413                 }
414             }
415             return false;
416         }
417         return true;
418     }
419 
420     /**
421      * @param text  the string to match
```

```
371             if (regExpRegionMatches(text, i, p, 0, plen)) {
372                 return i;
373             }
374         }
375         return -1;
376     }
377 
378     /**
379      *
380      * @return boolean
381      * @param <code>text</code>, a String to match
382      * @param <code>start</code>, int that indicates the starting index of match, inclusive
383      * @param <code>end</code> int that indicates the ending index of match, exclusive
384      * @param <code>p</code>, String,  String, a simple regular expression that may contain '?'
385      * @param <code>ignoreCase</code>, boolean indicating wether code>p</code> is case sensitive
386      */
387     protected boolean regExpRegionMatches(String text, int tStart, String p,
388             int pStart, int plen) {
389         while (plen-- > 0) {
390             char tchar = text.charAt(tStart++);
391             char pchar = p.charAt(pStart++);
392 
393             /* process wild cards */
394             if (!fIgnoreWildCards) {
395                 /* skip single wild cards */
396                 if (pchar == fSingleWildCard) {
397                     continue;
398                 }
399             }
400             if (pchar == tchar) {
401                 continue;
402             }
403             if (fIgnoreCase) {
404                 if (Character.toUpperCase(tchar) == Character
405                         .toUpperCase(pchar)) {
406                     continue;
407                 }
408                 // comparing after converting to upper case doesn't handle all cases;
409                 // also compare after converting to lower case
410                 if (Character.toLowerCase(tchar) == Character
411                         .toLowerCase(pchar)) {
412                     continue;
413                 }
414             }
415             return false;
416         }
417         return true;
418     }
419 
420     /**
421      * @param <code>text</code>, the string to match
```

src.org.eclipse.ui.internal.ide.StringMatcher.java - src.org.eclipse.ui.views.navigator.StringMatcher.java

```
422     * @param start  the starting index in the text for search, inclusive
423     * @param end  the stopping point of search, exclusive
424     * @param p  a string that has no wildcard
425     * @return the starting index in the text of the pattern , or -1 if not found
426     */
427    protected int textPosIn(String text, int start, int end, String p) {
428
429        int plen = p.length();
430        int max = end - plen;
431
432        if (!fIgnoreCase) {
433            int i = text.indexOf(p, start);
434            if (i == -1 || i > max) {
435                return -1;
436            }
437            return i;
438        }
439
440        for (int i = start; i <= max; ++i) {
441            if (text.regionMatches(true, i, p, 0, plen)) {
442                return i;
443            }
444        }
445
446        return -1;
447    }
448}
449
```

```
422     * @param <code>start</code>, the starting index in the text for search,
    inclusive
423     * @param <code>end</code>, the stopping point of search, exclusive
424     * @param code>p</code>, a string that has no wildcard
425     * @param <code>ignoreCase</code>, boolean indicating wether code>p</code> is
    case sensitive
426     * @return the starting index in the text of the pattern , or -1 if not found
427     */
428    protected int textPosIn(String text, int start, int end, String p) {
429
430        int plen = p.length();
431        int max = end - plen;
432
433        if (!fIgnoreCase) {
434            int i = text.indexOf(p, start);
435            if (i == -1 || i > max) {
436                return -1;
437            }
438            return i;
439        }
440
441        for (int i = 0; i <= max; ++i) {
442            if (text.regionMatches(true, i, p, 0, plen)) {
443                return i;
444            }
445        }
446
447        return -1;
448    }
449}
450
```