University of Huddersfield

MEng Group Project

Cryptic Crossword Solver

Development Document

Authors:
Mohammad Rahman
Leanne Butcher
Stuart Leader
Luke Hackett

Supervisor:
Dr. Gary Allen

Examiner:
Dr. Sotirios Batsakis

Friday, 9^{th} May 2014

Contents

List of Figures

Chapter 1

Development

The development phase was split across three 'iterations'. Each iteration would add and improve upon existing functionality. In order to keep the documentation as compact as possible, this chapter will only focus upon the final development phase. This will ensure that the contents of this chapter is in line with the latest development phase and source code.

This chapter will focus upon the user interface development and how it interacts with the Java Servlets. There will also be significant focus upon the 'back end' system, including how a clue is solved and how the solvers interact with various resources such as dictionaries and thesauri.

Finally the overall system architecture will be discussed, providing an insight into its development and how it has added to the overall product.

1.1 User Interface

In order for users to use the system a simple and powerful user interface was required. The design reasonings behind the user interface were described in the design section.

The user interface has been designed utilising a fall-back system, which is a standard web development approach. The majority of the user interface is delivered by the server utilising JavaServer Pages (JSP) language.

The JSP language essentially extends from XML, and allows for html-like code to be written so that when compiled with Java, a full server-side page is rendered. Within this project an additional JavaServer Pages Standard Tag Library was used. The library – xml – provided additional functionality so that JSP was able to directly utilise XML within its rendering technique.

Figure 1.1 illustrates an example XML parsing snippet from the `solver.jsp` file.

```
<x:choose>
  <x:when select="$solution/trace">
    <p>Solution Trace:</p>
    <ol>
      <x:forEach select="$solution/trace" var="trace">
        <li><x:out select="$trace"/></li>
      </x:forEach>
    </ol>
  </x:when>
  <x:otherwise>
    <p>Solution Trace Unavailable.</p>
  </x:otherwise>
</x:choose>
```

Listing 1.1: isPatternValid deduces if a given solution pattern is valid

The code snippet above shows use of the XML library, as denoted by the '`x`' name space to certain elements. The code is utilising XPATH to find all possible traces that are listed within the trace element (the trace element is an array).

For each of the traces found they will be printed out into the standard HTML output. However if a solution has not been found a simple predefined message will be presented.

The server side rendering that has been described above is known as the fall-back option. This option will work on all browsers and operating systems. The reason for this is that the rendering is controlled by the server, and hence can be fully tested.

Many modern browsers will support JavaScript in some form of fashion. This allows for

various additional functionality to be provided to enhance the user's web browsing experience.

The cryptic crossword website features a JavaScript override that will override the default server-side rendering and provide its own rendering. For example when a user clicks upon the 'solve' button, the website will display a message informing the user that the clue is currently being solved as shown in figure 1.1. Under a non-JavaScript supporting browser this would simply look like a normal page request that is taking its time.

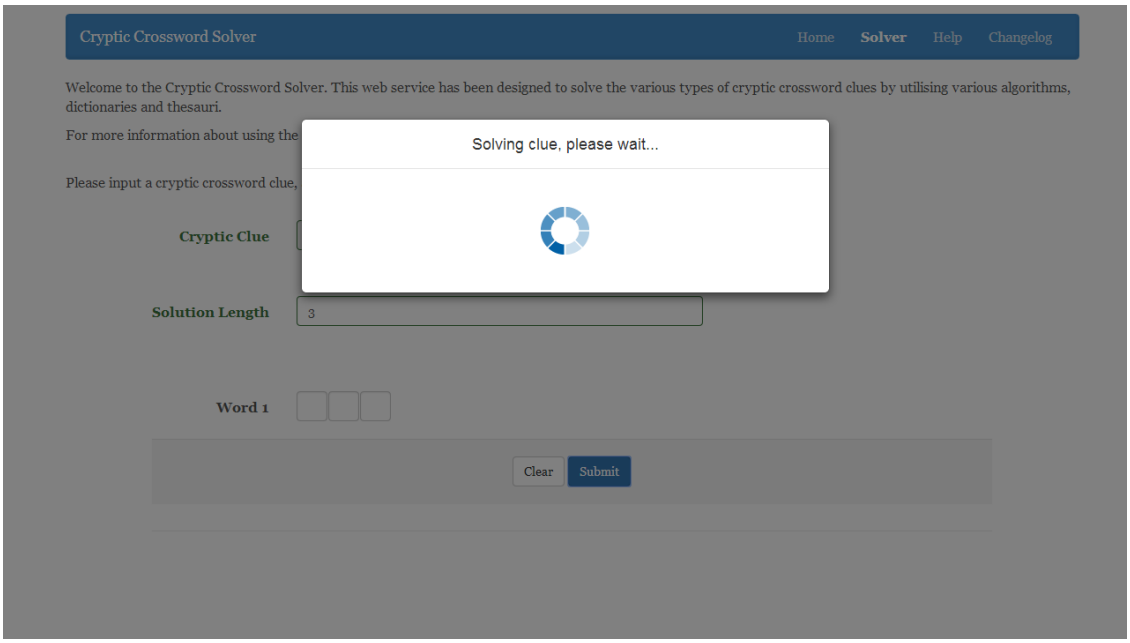

Figure 1.1: Message informing the user that a clue is currently being solved

The JavaScript engine will also provide some basic forms of validation, to prevent the user from waiting a long time simply to find out that there was a validation error. The validation occurs upon a key press, and hence the feedback is instant as shown in figure 1.2.

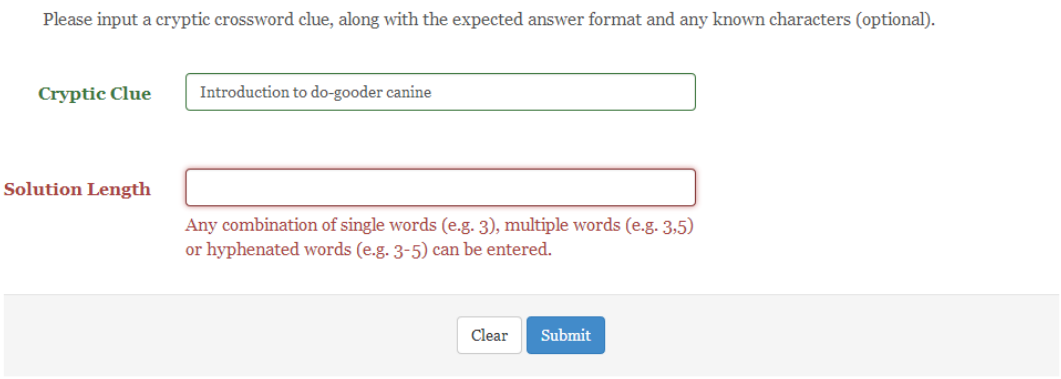

Figure 1.2: Live validation feedback ensures users are entering correct data

The JavaScript engine will also make POST requests to the server. This reduces the total amount of work the server will be required to do, as it only has to return the requests (rather than render HTML).

The reason for the fall-back system that was previously described is that if for some reason the JavaScript engine fails to load, or is incompatible with the device, then the JavaScript will not load. However, the functionality of the site will still continue to work, as the browser will 'fall-back' to the standard server-side validation and rendering.

1.2 Web Service & Servlets

The core system is wrapped around a RESTful web service, that allows users from various devices to submit clues to be solved. Within this section both the web service and the servlet implementations will be discussed and presented.

1.2.1 Web Service

During the project analysis phase the decision was taken that the system's functionality will be delivered via a web service. The web service was developed using Java Enterprise Edition (Java EE) and Tomcat 7.

The main reason for using Java is that the web service could easily make use of the various packages that are provided by the chosen natural language processing library — Apache OpenNLP written in Java.

The web service has solely been produced using the Java EE platform and does not use any additional frameworks or libraries such as Apache Axis. The reason being is that the Java EE platform will run on any machine that is capable of running the Java virtual machine without any additional configuration.

Although Apache Axis (for example) provides additional functionality in configuring the web service, it was decided that the project was to focus upon the solving of a clue. Therefore a 'standard web service' setup would easily meet the requirements of the project.

Another design decision was taken to ensure that the web service followed a RESTful style of communication. The main reason for this is that some of the target devices (i.e. mobile and tablet platforms) do not support SOAP based communication without additional plug-ins. RESTful web services also provide a number of advantages over their SOAP-based counter parts, as was highlighted within the research section.

1.2.2 Servlets

The servlet design has been split across two classes — `Servlet` and `Solver`.

The `Servlet` class extends the standard `HttpServlet` class and provides common functionality. The `Servlet` class provides a base for all system Servlets to use the common functionality.

The `Servlet` class is able to if a given request is from a JSON, XML or Ajax background. For example, if a client was to make a request to the web service through a web browser utilising an Ajax request, then the `isAjaxRequest` method would return `true`.

For illustrative purposes the `isAjaxRequest` method is shown below in listing 1.3.

```
protected boolean isAjaxRequest(HttpServletRequest request) {
  String ajax = request.getHeader("x-requested-with");
  return ajax != null && ajax.toLowerCase().contains("
     xmlhttprequest");
}
```

Listing 1.2: isAjaxMethod deduces if a request was made by AJAX

The servlet also contains two customised methods – one to handle errors, and the other to handle a good response – that are able to send a response back to the requesting client based upon a number of factors.

For example the methods are able to convert the return data into either XML or JSON depending upon what the client has asked for. The `sendError` method will also set the HTTP status code correctly, allowing the client to correctly authenticate the response.

Finally the `Servlet` class overrides the `init` method, which is automatically called as part of the object construction. This method will initialise resources at servlet creation (i.e. first run-time within tomcat) rather than during the first call to the servlet.

In doing this, Tomcat will take more time initially starting up, however the user will notice that their queries are dealt with much quicker. In this project the `init` method has been used to initialise the various in-memory dictionaries and thesauri.

The second class is the `Solver` servlet and handles all requests that are specifically for solving a given clue. The `Solver` servlet accepts both GET and POST requests, with each requiring the clue, the length of the solution and the solution pattern.

The `Solver` servlet upon receiving a request will validate the input parameters, based upon a number of criteria including presence checks and regular expressions. The code snippet below is an example validation rule that will validate the solution pattern against a regular expression.

```
private boolean isPatternValid(String pattern) {
   // Pattern string regular expression
   final String regex = "[0-9A-Za-z?]+((,|-)[0-9A-Za-z?]+)*";
   boolean match = Pattern.matches(regex, pattern);

   // Pattern String must be present and of a valid format
   return isPresent(pattern) && match;
}
```

Listing 1.3: isPatternValid deduces if a given solution pattern is valid

In order for validation to pass, the solution pattern must not be empty and must match to the regular expression stated in the method.

The `Solver` servlet class will initialise the solving of a clue if the three inputs are deemed to be valid. The `solveClue` method will utilise the Clue manager class, that will handle the distributing of the clue to the various solvers.

This has been designed so that the servlet and the solving processes are upon separate threads. This prevents Tomcat from freezing and allows it to handle requests from users.

Once all the solvers have finished executing, the `Solver` servlet will produce an XML document based upon the various elements. Once the XML document has been created, it will be sent back to the client as either XML or JSON.

1.3 Core

Within this section the core package will be presented. The core package is the package that forms the heart of the system, and provides various base classes that when instantiated will represent Clues, Solutions and Solution Patterns.

1.3.1 Clue

The clue class represents an individual cryptic crossword clue, and will maintain a list of possible solutions that have been computed. The Clue class is a basic class, that essentially provides references to other aspects of solving a clue, such as the solution pattern (see subsection 1.3.4).

A clue is able to have a number of solutions, and hence each clue will house a SolutionsCollection, which is described in more detail in subsection 1.3.3.

A clue will also have a solution pattern – described in subsection 1.3.4 – which allows for potential solutions to be matched against an expected pattern.

1.3.2 Solution

The solution class implements the Comparable interface, which is a standard Java interface that imposes ordering upon the object that implements it. It was decided that the solution class should implement the interface, so that solutions can be compared.

The comparing of solutions is perhaps one of the most common pieces of functionality that the system will be required to do. An example use case would be comparing any number of solutions to deduce which is more likely to be the correct answer.

Listing 1.4 shows the implemented compareTo() method found within the Solution class.

```
public int compareTo(Solution o) {
  int solutionCompare = solution.compareTo(o.getSolution());
  int confidenceCompare = -1
      * Double.compare(confidence, o.getConfidence());

  if (solutionCompare == 0) {
    // compare the actual solution text.
    return 0;
  } else if (confidenceCompare == 0) {
    // return a comparison of the solution text.
    return solutionCompare;
  } else {
```

```
      // compare them based on their confidence.
      return confidenceCompare;
    }
  }
```

Listing 1.4: compareTo() compares two solutions

The above code will try to deduce which of the two solutions are closest to being correct. It does this by comparing their confidence ratings, to which every solution will have. If the solution is the same the 0 is returned, whist -1 or +1 returned depending upon which solution is closest to being correct.

As well as housing the confidence rating, the solution class also houses the solution trace. The solution trace is a list of steps that were taken in order to compute the solution. It is intended that the solution traces will help to teach the user how to complete similar clues in the future.

1.3.3 SolutionCollection

The SolutionCollection class extends the standard Java HashSet class utilising the Solution class as the element type. Although a HashSet can not guarantee the order of the set, it can guarantee that no duplicates will be added to the set. This decision was taken to ensure that the system is not dealing with large amounts of repetitive datasets that will only harm the performance of the system as whole.

In order to get around the fact that the ordering of the set has no guarantee, the system makes use of the fact that the element type – Solution – implements the Comparable interface. This means that a copy of the current collection can be retuned in a sorted order if possible. This is illustrated in listing 1.5.

```
  public Set<Solution> sortSolutions() {
    return new TreeSet<>(this);
  }
```

Listing 1.5: Method returns a new sorted collection

As both the TreeSet class and the HashSet both share the same parent class Set, it is possible to cast the result of this method back to a SolutionCollection.

The SolutionCollection class overrides basic methods found within the HashSet class, such as `contains`, `add`, `addAll`, whilst providing additional functionalities, such as returning all solutions whose confidences are greater than (or less than) a given value.

1.3.4 Solution Pattern

The SolutionPattern class models the solution to a corresponding clue. For example if the clue is nine letters long, then the SolutionPattern class would provide information about the pattern of that solution using any given known characters.

The solution pattern is able to split a well-formed pattern and represent it as an in-memory object allowing for a faster clue matching process, in comparison to constantly trying to match a string.

A good example of the level of functionality available in this class is shown in listing 1.6.

```
public void filterSolutions(Set<Solution> solutions) {
  Collection<Solution> toRemove = new ArrayList<>();
  // For each proposed solution
  for (Solution solution : solutions) {
    // If it doesn't match the pattern, throw it out
    if (!match(solution.getSolution())) {
      toRemove.add(solution);
    }
  }
  solutions.removeAll(toRemove);
}
```

Listing 1.6: Method returns collection of matched solutions

The code snippet will match the given set of solutions — often a SolutionCollection — to the current object. A match can simply be described as matching a solution pattern to a possible solution, for example 'd??k' could match to 'duck', 'deck' or even 'dork'.

This method is often used within the filtering down of potential solutions, and thus ensures that the application is not dealing with too much data. In effect this helps the application become more efficient when solving solutions.

Obviously the efficiency of the matching process is directly linked to the number of known characters. If a large number of known characters is given by the user, then the total 'search space' is dramatically reduced.

As an example there are about 308 million different letter combinations within a six letter word (26^6) – this refers to the arrangement of letters, and not 'actual' words. If just one of those letters is known, this would be reduced down to 11 million different letter combinations (26^5), and knowing two letters would reduce it down to around half a million (26^4).

1.3.5 Manager

The Manager class manages the process of solving the given clue. The manager class heavily utilises the standard Java future interface, which is designed to represent the result of asynchronous computation.

Listing 1.7 illustrates the distribution functionality that distributes a copy of all the necessary resources — such as the clue and its solution pattern – and starts the computation upon various new threads (if available upon the system).

Each of the solvers are obtained dynamically via the plug and play system to which more information can be found within section 1.4 on page 14. Once the solvers are obtained they are initialised. The initialisation process simply creates a new object instance of the solvers, and starts them solving the clue.

Once all solvers have finished, their computed solutions are added to an overall solutions collection, which will ignore duplicate solutions as explained in section 1.3.3.

Each of the solutions will have their confidences adjusted based upon how 'correct' the solution is.

```
public SolutionCollection distributeAndSolveClue(Clue clue) {
    // This will hold the solvers to be run at runtime
    Collection<Solver> solvers = getSolversFromClasses(clue);
    // This will hold the returned data from the solvers
    Collection<Future<SolutionCollection>> solutions =
       initiateSolvers(solvers);
    // This will hold all solutions that have been returned
    SolutionCollection allSolutions = new SolutionCollection();
    // Now we need to 'unpack' the SolutionCollections
    for (Future<SolutionCollection> future : solutions) {
      try {
        allSolutions.addAll(future.get());
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
      }
    }
    // Adjust confidence scores based on category matches
    Categoriser.getInstance().confidenceAdjust(clue,
       allSolutions);

    return allSolutions;
  }
```

Listing 1.7: Method to distrbute the clue out to all solvers

1.4 Plug and Play Architecture

One of the major aspects of the design and development of the system was the underlying architecture. A well designed architecture would not only improve the overall efficiency of the system, it would also aid in reducing development time and improve the maintainability of the system.

It was decided that a 'plug and play' architecture would be adopted. A plug and play architecture allows for external components to be attached to a main system, and thus allows for that system to be able to detect that component and use it (plu, 2006).

The plug and play architecture is a common model found within computing, an example usage would be a network. A computing device is able to connect (either via a wire or wireless) to a network and it will automatically be able to access the network resources without having to reconfigure the computer, the network or other devices connected to that network.

The plug and play architecture would aim to solve the problem of having an unknown number of solvers working within the overall system. By implementing the architecture, any number of solvers could be added and removed to the system at any time (compile time or run time). Thus allowing the system to be as flexible as possible when it comes to handling the solvers. As a bi-product of this, it would also enable any future developers to be able to add their own solver without having to re-factor any core application code.

The architecture uses an 'includes' file which lists all solvers that are intended to be included within the system. The file also supports the ability to ignore files which enables solvers to not be included in the system at run time. Listing 1.8 illustrates an example snippet. Lines that start with '#' are files that are to be ignored — i.e. the class will not be imported at run time. This is illustrated in listing 1.8.

```
uk.ac.hud.cryptic.solver.Acrostic
uk.ac.hud.cryptic.solver.Anagram
#uk.ac.hud.cryptic.solver.Homophone
#uk.ac.hud.cryptic.solver.DoubleDefinition
uk.ac.hud.cryptic.solver.Pattern
```

Listing 1.8: plug and play solver import file

Listing 1.8 illustrates that the Acrostic, Anagram and Pattern solvers will be 'loaded' in to the system's list of available solvers. Whilst the Homophone and DoubleDefinition solvers will not be loaded. It must be stated that currently the system only supports the loading of solvers at the initial start up of the application. Solvers can not be added or removed during the use of the application, but due to the flexibility of the architecture this could be easily added in a future revision.

The loading and parsing of the classes are handled within the manager class found within the core package, which was described in more in detail in section 1.3.5 on page 13.

Each of the required classes added to the properties file are imported by utilising Reflection. Reflection allows for the solver classes to be 'loaded' at run time, by importing the Java class and then instantiating it at run time. Listing 1.9 illustrates the snippet of code that 'imports' and initialises the various solvers.

```
try {
  // Name of the solver read from the propertes file
  String solverName;
  // Load the class
  Class<?> cls = Class.forName(solverName);
  Constructor<?> c = cls.getDeclaredConstructor(Clue.class);
  // Instantiate the given solver class
  Solver solver = (Solver) c.newInstance(clue);
  // Add to the list of solvers
  solvers.add(solver);
} catch (Exception e) {
  // Error handling omitted
}
```

Listing 1.9: run time class import using reflection

Although reflection may provide a number of issues such as security implications and performance issues in this instance it was deemed to be a benefit, as it allows for extensible features (solvers) to be defined at run time, rather than at compile time.

As long as all additional solvers extend the main solver class, then the above reflection practise will work for all future solvers.

1.5 Solvers

A Feasibility Study was created to attempt to predict the difficulty of specific clue types and their regularity in cryptic crosswords. All seventeen clues types were analysed in order to plan which would be implemented in each iteration of the implementation process. The first iteration involved Hidden, Anagram, Acrostic and Pattern as they all had a low difficulty. The next iteration involved Homophones, Palindromes, Double Definition and Spoonerisms to step up the difficulty of the algorithms. Finally, the last solvers to be implemented were Charades, Deletions, Containers and Reversals as they were seen as the most beneficial to implement in the time left for the project.

Therefore, with the additional reason of a time limit, Purely Cryptic and & lit clue types were not implemented due to their high difficulty and Substitutions, Shifting and Exchange clue types were not implemented due to their rarity.

1.5.1 Hidden

When investigating the Hidden clue type, it was found that the answer to the clue would be within the clue itself. For example:

Creamy cheese used in apricot tart.

Answer: RICOTTA

In the clue above the answer 'ricotta' is hidden within the two words 'apricot tart'. The algorithm takes the clue as a whole, without spaces, and then uses the substring method within Java to find all possible hidden words (as the Hidden clue type can also hide words in reverse, the clue is also passed in reverse). It does this by taking the index of the for loop and length of the solution as boundaries.

Listing 1.10 illustrates how all possible solutions are found (without additional solution trace functionality):

```
int index;
for (index = 0; index <= limit; index++) {
    Solution s = new Solution(clue.substring(index, index +
        totalLength), NAME);
    solutions.add(s);
}
```

Listing 1.10: Retrieving all possible solutions using the substring method in Java

For the above clue, where limit is equal to seven because 'ricotta' is seven letters long, the first five iterations will add the following solutions to the list; 'creamyc', 'reamych', 'eamyche',

'amychee', 'mychees'. Eventually, the loop will pick up 'ricotta' and add it to the solution list. From the output gained in the first five iterations it is apparent that a lot of invalid solutions are added to the list, therefore a number of steps are also implemented to eliminate invalid solutions.

Once all possible solutions have been found, they are all checked to make sure they are not words from the original clue. For example, 'apricot' is seven letters long and will have been picked up through the algorithm, however it is not a hidden word and therefore not a valid solution.

Next, the solutions are checked against the pattern provided. This means if the user has input known letters or there are spaces in the end solution but the solution being checked does not match these requirements, these solutions are removed.

Finally, all solutions are checked against the dictionary to determine whether they are valid words. The solutions that are left are then returned to the user.

1.5.2 Anagram

Within a clue in the form of an 'Anagram' the end solution is placed within the clue itself. For example:

A sportsman spinning the tale

Answer: ATHLETE

The answer comes from rearranging the letters in 'the tale' to get athlete.

The algorithm first checks whether it is possible to be an algorithm clue type. This is done by checking the length of the clue is greater than the solution length. The solution should be composed of all the characters from one (or more) words contained within the clue. In other words, the anagram solution cannot be formed from characters that have been picked and chosen across all the words of the clue. The next step creates a list of possible substrings of the entire clue, which match up with the length of the solution.

Once all the substrings have been collected from the clue, for each substring a new thread is created to speed up the algorithm as finding all possible combinations of letters within a clue on one thread is slow. Within each thread, the combination is passed to a recursive method which attempts to build up more combinations which could potentially be the solution. Dictionary filters and pattern filters are handled within the recursive method meaning once all possible solutions have been found, they are returned to the user.

1.5.3 Acrostic

The Acrostic clue type is another of the types which have the end solution placed within the clue itself which can be seen in the following clue:

What's seen at start of any road running one way?

Answer: ARROW

The answer to the above clue can be found by taking the first letter from each word in the phrase 'any road running one way'. The algorithm splits the clue into an array of words to allow each word to be looked at individually. For each word, the first letter is taken using the substring method in Java. The first letter of each word is stored in a string to use the substring method once again to search through possible solutions within the string.

To find the possible solutions from the string, it is done within a loop with the same number of iterations as the length of the end solution. This means the loop will pick up the following possible solutions for the above clue; 'wsaso', 'sasoa', 'asoar', 'soarr', 'oarro', 'arrow'.

As with previous algorithms (e.g. Hidden), this method adds various invalid potential solutions into the list. To eliminate invalid solutions, the potential solutions are run through the dictionary to determine whether they are valid English words. Finally, the potential solutions will be run through the pattern given by the user and the solution itself to ensure the requirements for the end solution match up. Once all this has been completed, the results are returned to the user.

1.5.4 Pattern

The Pattern clue type is one which holds the end solution within the clue itself. For example:

Beasts in tree sinned, we hear - nothing odd there

Answer: REINDEER

From the phrase 'tree sinned, we hear', each odd letter spells 'reindeer'.

The algorithm firstly checks that the pattern can potentially be of type Pattern. This is done by taking the total length of the clue without any punctuation and dividing it by two. By doing this, the maximum length of any solution found by this algorithm is determined. This means if the end solution is longer, it cannot be a clue of this type and therefore the algorithm ends.

The next step is to look for words in even positions. To do this a string of letters to find potential solutions within it needs to be generated. Listing 1.11 illustrates how every other character is found whether it is for even or odd letters:

```
private String getEveryOtherChar(Clue c, boolean even) {
    final String text = c.getClueNoPunctuation(true);
    String newString = "";
    int i;
    for (i = even ? 0 : 1; i < text.length(); i += 2) {
        newString += text.charAt(i);
    }
    return newString;
}
```

Listing 1.11: Retrieving every other character from a string

The method above gets the clue as a string with no punctuation or spaces. It then takes the letter at an even index or an odd index depending on the value of the boolean passed in. For finding letters in even positions, the value will be true and for the example clue above the following string will be generated; 'batitesnewhantigdtee'.

With the string generated, substrings are found within a loop using the length of the end solution as a boundary. This means with the string generated above, the first five iterations within the loop will find the following possible solutions; 'batitesn', atitesne', 'titesnew', 'itesnewh', 'tesnewha'.

When all possible even solutions have been found, the algorithm then repeats the same steps to find all possible odd solutions. The final steps are to filter the solutions on their length and whether they match the known letters, if any, that have been input by the user as well as checking whether the words are valid dictionary words. Once all these steps have been completed, the potential solutions are passed back to the user.

1.5.5 Homophone

A 'Homophone' clue solution is a word which sounds the same as a word in the clue or a synonym of a word in the clue. For example:

Animal said to have connections

Answer: LINKS

The answer comes from an animal called 'lynx' which has the same pronunciation as the solution 'links'.

The algorithm uses a homonym dictionary to retrieve pronunciations for clue types and synonyms. Each word of the clue is taken and firstly run through a method which gets all homonyms of that word. Then, the algorithm retrieves all the synonyms for each clue word and in turn finds all the homonyms of each of the synonyms.

Once all the homonyms have been found the algorithm checks to make sure none of the clue words themselves have been added as potential solutions and removes them if so. They are then checked to determine that they are valid dictionary words and match the pattern given by the user. Finally, all potential solutions are passed back to the user.

1.5.6 Palindrome

The Palindrome clue type gets it's end solution by taking a word from the clue itself and taking a synonym from it. This synonym must be spelt the same whether the word is reversed or in it's normal state. An example of a palindrome is 'noon' because when the word is reversed it still retains it's original spelling. Below is an example Palindrome clue:

Look both ways

Answer: PEEP

As with 'noon', 'peep' can be reversed and still retains it's original spelling.

The algorithm for the Palindrome clue type first takes each word from the clue and places them into a list. Each word is then run through the thesaurus to find their synonyms. Once this has been completed a filtering mechanism is needed to find all synonyms which follow the rules of palindromes.

Listing 1.12 illustrates how all the synonyms that have been found are filtered by the algorithm:

```
private void filterNonePalindromes(Collection<String>
   solutions) {
   for (Iterator<String> it = solutions.iterator(); it.hasNext
      ();) {
       String normal = WordUtils.removeSpacesAndHyphens(it.
          next());
       String reverse = new StringBuilder(normal).reverse().
          toString();

       // If the word isn't "symmetrical" from both sides,
          remove it
       if (!normal.equals(reverse)) {
           it.remove();
       }
   }
}
```

Listing 1.12: Checking to see if a word is a palindrome

The method to remove synonyms that are not palindromes uses an iterator to loop through each one. For each one, spaces and hyphens are removed to make the comparison possible. This step is essential for palindromes such as 'put up' because with the space present the algorithm would not see 'pu tup' as equal. Once all punctuation and spaces have been removed from the synonym, the alogorithm then uses the reverse method within Java to retrieve the reversed synonym. For example, if the word 'cryptic' was passed to the reverse method it would return 'citpyrc'. This means when the normal word 'cryptic' and the reverse word 'citpyrc' are passed to the if statement to check whether they were equal, it would be false and therefore the synonym 'cryptic' would be removed as a potential solution.

Once the synonyms have been filtered, the algorithm then does further filtering to make sure the potential solutions fit the pattern and known letters of the end solution. When the filtering has taken place, the potential solutions are passed to the user.

1.5.7 Double Defintion

The solution for a Double Defintion clue comes from taking two words within the clue that could be the definition of the solution. For example:

Car plant

Answer: LOTUS

A 'lotus' can be either a type of car or a type of plant, hence why the clue is of type Double Definition.

To explain the algorithm for this clue type it is necessary to also explain the functionality that comes with using the thesaurus. The thesaurus used for the project is a text file where each line starts with a word. Each of the following words on that line are synonyms of the first word. From implementing the solvers different ways to access synonyms were found, these different methods provide a range of results which vary from vague to limited synoynms being found. The Double Definition algorithm uses two of these methods, one which takes a clue word and finds it in the thesaurus where it is the first word and adds all the synonyms to a list, this method can be known as getting 'first level synonyms'. The second method does not only look for the clue word as the first word of a row, it looks for the clue word in any position in any row in the thesaurus, therefore returning a wider range of results. This second method can be known as getting 'second level synoynms'.

The algorithm itself first splits the clue into an array so each word can be looked at individually. For each word in the clue, both the methods explained above are used. This means two maps are created, one which contains the clue word along with it's first level synonyms and a second which also contains the clue word but along with it's second level synoynms.

Now the algorithm compares each of these maps to find different words in the clue with the same synonym. This is firstly done by comparing each word and it's synonyms for the first

level synonyms. Next, it is done by comparing first level synonyms with second level synonyms. If it occurs, the synonym that has been found for two words in the clue, the potential solution is added to a list. As the thesaurus methods check for invalid solutions (potential solutions which do not match the length of the end solution etc.), once all synonyms have been matched up and checked, the results can be returned to the user.

Second level synonyms are not compared with second level synonyms. This is because second level synonyms can be extremely vague so the chances of finding a synonym which links to two words in the clue is more likely and less likely to be the correct answer. Another reason is that the likelihood of this algorithm returning results for a clue that is not in fact of Double Definition type is highly more likely if second synonyms for one word are compared with second synoyms for another. Although, the chances of finding the answer for all double defintion clues by comparing two sets of second level synonyms is highly beneficial, the downfalls outweigh the benefits and therefore that is why this design decision was made.

1.5.8 Spoonerism

A 'Spoonerism' clue type involves swapping the first one or two letters from two words to retrieve a new phrase. For example:

In which to immerse the beasts of Spooner's ocean liner

Answer: SHEEP DIP

A synonym for 'ocean' is deep and a synonym for 'liner' is ship. When the first letter from one of these synonyms is swapped for the first two letters from the other synonym, the phrase 'sheep dip' is created.

The first task the algorithm completes involves removing all words within the clue that have less than or a length of two letters. This is because the swapping of two letters would not be possible on a word of two letters meaning it should potentially also speed up the algorithm by removing extra words to complete functionality on. The next step checks to see if the clue contains a word which starts with 'spoon' to determine whether the clue is of this type. This step is necessary as all 'Spoonerism' clues have an indicator like 'Spooner' or 'Spooner's' meaning the algorithm can terminate if it is not found.

It is also common for the words that have their synonyms manipulated are close to the indicator either to the left or the right so the algorithm takes two words to the left of the indicator and two words to the right. Then, the minimum and maximum length of synonyms to find are then calculated if the solution is one word or multiple words.

With the words found around the indicator and the minimum and maximum lengths of synonyms retrieved, synonyms are found for each of the words and put into a list. Next, each synonym is paired up with every other synonym and they are passed in as a pair

to another method which swaps their first letter or letters around. This method does the following possible combinations to find the potential solutions; for ‘deep’ and ‘ship’, the first two letters (‘seep’, ‘dhip’), two letters with one letter (‘sheep’,‘dip’), one letter with two letters (‘sep’,‘dehip’), two letters with two letters (‘shep’,‘deip’).

Each combination is checked against the pattern and the dictionary. If they are valid they are added to a list of possible solutions which are returned to the user.

1.5.9 Charade

The ’Charade’ clue type can take a range of different aspects of a clue to get to the end solution. The example below shows a clue which uses all possible aspects which can be used:

Quiet bird has a sign on a strange occurrence

Answer: PHENOMENON

In the clue above, abbreviations, synonyms and clue words themselves are used (it is also possible for substrings of clue words to be used). To get to the answer; ‘P’ is an abbreviation of ‘quiet’, ‘HEN’ is a synonym of ‘bird’, ‘OMEN’ is a synonym of ‘sign’ and ‘ON’ is a full word taken from the clue itself. All these appended together make up the end solution ‘phenomenon’.

The first step for the algorithm is to take each of the clue words and get synonyms, construct substrings and retrieve abbreviations for each word. It then attempts to reduce all the data retrieved by comparing to the pattern input by the user.

With all the possible abbreviations, substrings and synonyms possible solutions are generated. To do this a set is created first. Charades can be constructed from any sequential components from the clue. For example, if the clue words are ”one two three”, then a solution may be constructed by using components for clue words ”one, two, three”, ”one, three”, ”one, two” or ”two, three”. A method in the algorithm generates these combinations of clue words which will be processed one by one in an attempt to find the solution.

Once these combinations have been found, they are processed one by one to attempt to find solutions. This is done using recursion with a base case of the potential solution being longer or of equal length to the end solution or running out of data to append to the solution. The recursive method takes an abbreviation or a substring or a synonym for a word in the combination passed in and appends another abbreviation, substring or synonym from another word featured in the combination passed in and carries on until it hits the base case. This generates a string which could be a potential solution. Once this has been achieved, possible solutions are checked against the dictionary and if they are valid they are returned to the user.

Retrieving all possible combinations makes the algorithm slow in speed. This means the solver has not been deployed on to the Cryptic Crossword Solver and will only be used for demonstration purposes.

1.5.10 Deletion

The solution for a 'Deletion' clue type comes from taking the first letter, last letter or both from a word. For example:

Dog beheaded bird

Answer: EAGLE

From taking the first letter from the word 'beagle' (a type of dog) the solution 'eagle' is found. 'Beheaded' is the indicator word for the example clue which tells the solver that the first letter must be taken. For the algorithm to determine whether the first letter, last letter or both must be taken from a word, an indicator file was created. The indicator file was split into three lists, one to determine the first letter being taken, another for the last and a final list for both letters to be taken.

The first task the algorithm completes is to read in the indicator file and compare it against the clue to determine which letter or letters need to be removed. Once this has been determined, the algorithm then finds synonym for each of these words using the thesaurus. All the synonyms are then filtered to remove the ones that will be longer than the end solution length when letters are removed.

When all the valid synonyms have been found, the algorithm then uses the substring method within Java to remove certain letters.

Listing 1.13 illustrates how the synonyms have letters removed depending on the position which has been determined by the indicator file:

```
// Remove head/head edge of a word
if (position == Position.HEAD || position == Position.EDGE) {
   solution = solution.substring(1);
}

// Remove tail/tail edge of a word
if (position == Position.TAIL || position == Position.EDGE) {
   solution = solution.substring(0, solution.length() - 1);
}
```

Listing 1.13: Removing letters from a word depending on the indicator

The code above compares the position found with the positions stored in the enum within the class then the letters are removed. If the synonym requires both the first and last letters to be removed, it will fall into both of the if statements and remove both.

After the correct letters have been removed, the algorithm checks if the word is still a valid dicationary word and matches the pattern provided by the user. If they are valid, they are added to a list and finally this list of potential solutions is returned to the user.

1.5.11 Container

The Container clue type involves putting a word inside another word. For example:

In appearance everyone is lacking in depth

Answer: SHALLOW

With the clue above, the synonym 'show' for appearance is found and the synonym for everyone, 'all', is found. Placing all inside show gives the solution 'shallow'.

A clue word itself can be put inside or around the outside of another clue word or synonym of clue word. Another possibility is a synonym can be placed inside or around the outside of another clue word or synonym of a clue word. These possibilities make the need for an indicator of which possibility it is, necessary.

For the Container clue type, an indicator file was created which determined which word went inside another. The indicators were seperated into two lists, one where the indicators mean the word in the left of the clue goes in the word in the right of the clue, and the other where the indicators mean the word in the right of the clue goes in the word to the left of the clue.

Firstly, the algorithm reads in the indicator file and finds the word in the clue which is an indicator. A variable is then set to determine whether the word to the left goes in the word to the right or vice versa.

The algorithm then finds synonyms for each of the words in the clue and adds them to a map. The synonyms are then filtered to make sure they are the correct length for the end solution. Once this has been achieved the synonyms are then matched up. This involves taking two synonyms for two words in the clue and putting one inside the other. This is done within a loop where the position of the word being contained is moved within the word acting as a container. For example, for the clue above, if the synonyms 'all' and 'show' are being matched, the loop will output the following potential solutions; 'sallhow', 'shallow', 'shoallw'. Once these potential solutions have been found, they are passed to the dictionary to make sure they are valid words.

When all the potential solutions have been found, they are then filtered to make sure any known letters input by the user match the potential solutions. Finally, they are all passed

back to the user.

1.5.12 Reversal

The reversal algorithm is the only algorithm to make some use of the natural language processing library (NLP) — Apache OpenNLP. Although the algorithm does not fully use the all capabilities of the NLP it does use it enough to ensure that the algorithms are not being over worked.

The algorithm will attempt to get a list of possible fodders that can be used throughout the solving process. From a initial review it was clear that the majority of the fodders within a reversal clue are a either a singular or plural noun.

Apache OpenNLP has a number of methods that can tokenise a string, meaning that each word in a sentence is aligned to it's type of word. For example if the algorithm was given the clue:

Secure weapons turned over (4)

Answer: SUNG (reversal of guns)

Then the only noun (or fodder) that would be returned is weapons. Although the algorithm can handle multiple fodders it can not handle the rare clues in which the answer can not be deduced from a noun.

Listing 1.14 illustrates the simply way in which the algorithm utilises Apache OpenNLP's tokenising system to be able to pick out all nouns.

```
for (int i = 0; i < tags.length; i++) {
  // Obtain words that are a singular or plural noun
  if (tags[i].equals("NN") || tags[i].equals("NNS")) {
    fodders.add(words[i]);
  }
}
```

Listing 1.14: Deducing all singular or plural nouns within the clue

Once a list of fodders have been computed each of the fodders will have their synonyms computed and reversed. If the newly generated 'word' can be found in the dictionary, and can be matched to the given patter then it is marked as a potential solution.

Listing 1.15 indicates the simplicity of the algorithm and allows for an efficient approach to solving the clue.

```
// Get all synonyms that match the reversed pattern
Set<String> synonyms = THESAURUS.getSecondSynonyms(fodder,
   pattern, true);
```

```
// Reverse all synonyms to try to create another word
for (String synonym : synonyms) {
  // Reverse the synonym
  String reversedWord = WordUtils.reverseWord(synonym);
  // Add as a solution if the reversed word is a real word
  if (DICTIONARY.isWord(reversedWord)) {
      // Add the solution to all possible solutions
      collection.add(new Solution(reversedWord, NAME));
  }
}
```

Listing 1.15: Core reversal algorthim deducing possbile solutions

However it must be stated that the algorithm (as with all other algorithms) does not have any understanding of the words it generates. For example it is unaware that Alpha and Beta are both types of characters. This can cause many issues for the reversal algorithm as it is depended upon additional/external 'knowledge'.

1.6 Resources

There are a number of resources which were needed to aid with solving the various clue types. As a lot of the solvers shared most of the resources a design decision was made to keep the resources in their own package for all solvers to access them as and when they need to.

1.6.1 Abbreviations

Some clue types, for example the 'Charade' clue type, require the knowledge of the different abbreviations that come with certain words. To provide this knowledge a file was found with a list of words and abbreviations in JSON format (JavaScript Object Notation).

Listing 1.16 illustrates the way in which the abbreviations for 'quiet' are displayed in the file:

```
"quiet": [
 "p",
 "pp",
 "sh",
 "mum"
],
```

Listing 1.16: A sample of the abbreviations file

The Abbreviations class reads in all the possible abbreivations from the file at run time and stores them within a map which includes the word to get the abbreviations for and it's abbreviations as a set.

There are two ways in which abbreviations could be needed to solve a clue, one way is to retrieve abbreivations for one word and the other is to retrieve abbreviations for a phrase or a whole clue.

The method to retrieve abbreviations for one single word simply returns the set of abbreviations for a given word (or key) in the map if it exists within the file.

The method to get abbreviations for a phrase or a whole clue gets the abbreviations for as many words as possible in the given clue. For example, "help the medic" will contain seven abbreviations for the word medic. "medal for the medic" will contain four abbreviations for medal and seven for medic. However, the clue "master of ceremonies" will return one abbreviation which matches the entire clue (i.e. "master of ceremonies"). The algorithm is greedy, and will attempt to match the biggest String possible in the given clue. This means it will match all of the String "master of ceremonies" before matching abbreviations for "master".

1.6.2 Dictionary

The dictionary is an essential resource for the solving of clues to programmatically determine whether a string of letters is a valid word. A text file was found with a list of words within it which is read into the Dictionary class when an instance is created.

Listing 1.17 illustrates simply how the words in the dictionary file are displayed:

```
abaci
aback
abacs
abactinal
abactinally
abactor
abactors
abacus
```

Listing 1.17: A sample of the dictionary file

Once the file has been read in, there are a custom list of words to exclude from the dictionary and a list of words that need to be added found from solving particular clues.

One of the methods used regularly is the filtering method which removes any words from a given collection that are not present in the dictionary. This is an effective way to remove words that have being constructed by a solver algorithm which are essentially just an assortment of letters which hold no identified meaning. A similar method is used to filter any prefixes passed in within a collection that are also not held in the dictionary.

When potential solutions have been found by a solver, it is necessary to ensure the algorithm has returned solutions that fit the end solutions pattern. Requirements such as the length input by the user and any known letters input by the user must be adhered to. In the dictionary class there is a method which gets all word matches within the dictionary for a given solution pattern. As with filtering solutions, there is also a method to match up all words in the dictionary that have the same prefix as a prefix passed in.

There are also simple methods solvers can used to identify whether a single word or a phrase is contained within the dictionary simply by checking whether the collection the dictionary file has been read into contains the word or not.

1.6.3 Thesaurus

For clue types which do not have the answer itself nested within it, it is usually necessary to take the clue words and find synonyms for them. This is where the necessity for a thesaurus applies to aid the algorithms in solving clues. A thesaurus file was found which holds a vast

number of entries where each word after the first word in an entry is a synonym of the first word.

Listing 1.18 illustrates how the words in the thesaurus file are displayed with the example word 'dank':

```
dank,boggy,damp,dampish,dewy,fenny,humid,marshy,moist,muggy,
     rainy,roric,roriferous,sticky,swampy,tacky,undried,wet,
     wettish
```

Listing 1.18: A sample of the thesaurus file for the word 'dank'

As with the other resources the file is read in and stored within a collection, this time in the form of a map where the first word is stored within an entry along with it's synonyms.

There are a wide range of different methods that can be used to retrieve a different array of synonyms from vague to specific. Below is a list of functionality that has been written to retrieve synoynms:

- Obtain a list of "synonyms of a word's synonyms" to increase the chances of finding the correct solution. These must match against a supplied pattern.
- Obtain a list of "synonyms of a word's synonyms" to increase the chances of finding the correct solution. These must match a minimum and maximum length passed in for the synonym.
- Retrieve all single word synonyms of a given word with a maximum and minimum length (because some synonyms can be more than one word long)
- Retrieve all synonyms of a given word with no pattern or minimum or maximum length
- Get the synonyms for as many words as possible in the given clue. For example, in the clue "help the medic", look for synonyms of "help the medic", "help the", "the medic", "help", "the", "medic".
- Retrieve all synonyms in the same entry in the thesaurus as a given word. This means to not only look at the first word of an entry for synonyms, look through all entries and return every entry which contains a given word.
- Retrieve all synonyms in the same entry in the thesaurus as a given word which match against the given pattern.
- Check if a given potential solution found by a solver matches as a synonym against any of the words present in the clue.

1.6.4 Homonym Dictionary

The 'Homophone' clue type requires a resource for retrieving homonyms for words. A file was found which lists words with a string representation of the words pronunciation and as with the other resources, the file is read in and stored within a collection holding the word and an list of the pronunciation split into chunks.

An additional collection is formed which is essentially a reverse of the first collection created. This allows pronunciations to be looked up faster. For example, looking up the pronunciation "HH AH0 L OW1" will return "hello". This saves having to iterate the entire homophone map to search for words with the same pronunciation.

Listing 1.19 illustrates how the words in the homonym dictionary file are displayed with the example word 'hello':

```
HELLO  HH AH0 L OW1
```

Listing 1.19: A sample of the homonym dictionary file for the word 'hello'

The class allows the algorithm to retrieve the pronunciation of a given word as well as get words which share the same pronunciation as the supplied word. This only works for words which share the exact same pronunciation.

1.6.5 Categoriser

Some algorithms for certain clue types require indicators to determine how to generate the end solution. For example, the 'Container' clue type requires an indicator to identify which word is the container word and which word is to be contained. For the purpose of providing a confidence score for an end solution, indicators can also be used. As the clue is passed to all solvers, it is possible that two solvers may return the same answer back to the user. By using the indicator file associated with the clue type it is possible to assign a higher rating to a solution from a solver if the clue has an indicator within it from the associated file.

To provide faster access to the indicators for each clue type, the files listing the indicators are read in and stored within a map in the Categoriser class with the name of the clue type and a collection of the indicators.

One use of the Categoriser class is to retrieve the indicators simply by passing in a clue type. Another, is the functionality to remove the indicator from a clue to potentially speed up a solver algorithm by removing extra unnecessary checking on an indicator word.

Listing 1.20 illustrates how the indicator is removed from the clue:

```
public String removeIndicatorWords(String c, String type) {
    // Remove any punctuation
    String clue = WordUtils.normaliseInput(c, false);
```

```
    // The indicator words for the given type have to be
       present
    if (indicators.containsKey(type)) {
        for (String i : indicators.get(type)) {
            if (clue.contains(i)) {
                // Only remove the first match
                clue = clue.replace(i, "");
                break;
            }
        }
    }
    // Remove any double spaces, etc...
    return WordUtils.normaliseInput(clue, false);
}
```

Listing 1.20: Removing the indicator from the clue

The code above uses another class to remove any punctuation from the clue itself, then it checks to see if the clue type has an entry within the map storing all the indicators. If a word in the clue matches an indicator in the indicator file, it is removed and the modified clue is returned.

As mentioned, the Categoriser class can also boost the confidence of a solution if it contains an indicator within the clue that is featured in the indicator file.

Listing 1.21 illustrates how the confidence is boosted using indicators:

```
public void confidenceAdjust(Clue c, SolutionCollection
   solutions) {
    Collection<String> matchingTypes = getMatchingClueTypes(c);
    for (String clueType : matchingTypes) {
        for (Solution s : solutions) {
            if (clueType.equals(s.getSolverType())) {
                double confidence = Confidence.multiply(s.
                   getConfidence(),
                        Confidence.CATEGORY_MULTIPLIER);
                s.setConfidence(confidence);
                s.addToTrace("Confidence rating increased as
                   the clue contains indicator
                       word(s) suggesting the solution is of
                          type \"" +
                       clueType + "\".");
            }
        }
    }
```

```
}
```

Listing 1.21: Boosting the confidence of a solution

The first line of code in the method above calls another method to retrieve a list of clue types that have an indicator in the associated indicator file which matches a word within the clue passed in. Then, all the solutions found from all the solvers for the clue passed in are matched against the clue types that have been found and if the solver the solution has come from matches a clue type found, the confidence is boosted using a method in the Confidence class.

1.7 Utilities

The utilities package (named utils) is a package that contains various classes that cannot be grouped together in their own package, and provide generic helper methods in the aiding the system as a whole.

Within this section two commonly used classes — Confidence and WordUtils — will be discussed in more detail.

1.7.1 Confidence

The confidence class will manage the various functions that are associated with calculating and assigning confidence scores. The calculating of any given confidence score can be easily achieved across multiple solvers by focusing upon the 'common elements' that are found between all solvers. Once the 'common elements' are calculated then specifics ratings can be applied on top.

Each generated solution will start with an initial confidence of 42. The confidence rating will then be increased or deceased based upon certain features that are found within the clue, solution and the solver that managed to generate the solution.

As previously mentioned there are a number of features, and to describe all of them would be laborious, however three example features will be presented below:

- if a solution is a synonym of the clue's definition word then the confidence is increased by 50%
- if a solution has a synonym of the clue's definition word then the confidence is increased by 10%
- if a double-definition with two first-level synonyms is found then the confidence rating is increased by 40%

It is possible for more than one confidence increase to be applied to one solution, however it is not possible for a confidence to be less than 0 or greater than 100.

The confidence class is a 'bear bones' class that simply stores the value of each type of confidence increase, and by how much that increase should be. The confidences adjustments are often made by the various resources as shown in the Resources section.

1.7.2 WordUtils

The WordUtils class is a collection of helper methods relating to the manipulation of words and sentences. The class is a static class, and has been set as static due to the fact that the

WordUtil functionality is used extensively throughout the system.

An example method is shown in listing 1.22. This method will deduce if a word can be 'created' using the supplied string of characters. This method is often used in conjunction with the dictionary and thesaurus classes when trying to find hidden words.

```
public static boolean hasCharacters(String targetWord, String
   characters) {
  // This list will be reduced as characters are consumed
  Collection<Character> remaining = new ArrayList<>();
  for (char c : characters.toCharArray()) {
    remaining.add(c);
  }
  // Loop over each of the target word's characters
  for (char c : targetWord.toCharArray()) {
    // Continue if the char is available in the character pool
    if (!remaining.remove(c)) {
      return false;
    }
  }
  // the target word can be built
  return true;
}
```

Listing 1.22: deduces if a word can be made with a given set of unilearncharacters

1.8 Plug-ins

Section 1.4 on page 14 described the overall system's plug and play architecture and how it has helped the overall project development phase. Within this section an in depth description and small example will be presented to help future developers create a new solver that can be plugged into the system.

All solvers must be placed within the `uk.ac.hud.cryptic.solver` package. This package will be the base package in which the run time class importer will attempt to import the various solver classes.

Once developed, the new solver class must be added (with it's full package name) to the properties file which was described in section 1.4.

Each solver must extend the `Solver` base abstract class. The newly extended class can have various methods but must implement the `solve` and `toString` methods.

The `toString` method should return a String representation of the type of solver that the solver is, for example "container" or "anagram". The `solve` method will be automatically called by the system, and the core algorithm must be placed within this method (or utilise class specific methods).

Listing 1.23 illustrates a basic acrostic solver template.

```
public class MyAcrosticSolver extends Solver {

  public MyAcrosticSolver() {
    super();
  }

  public MyAcrosticSolver(Clue clue) {
    super(clue);
  }

  @Override
  public SolutionCollection solve(Clue c) {
    // Add the algorithm here
  }

  @Override
  public String toString() {
    return "acrostic";
  }

}
```

Listing 1.23: An example user-defined solver

As shown in listing 1.23 the solve method returns a SolutionCollection which is added to an overall collection to deduce the likely correct solution to the clue. This was described within sub-section 1.3.5 on page 13.

Glossary of Terms

The following section contains a glossary with the meanings of all names, acronyms, and abbreviations used by the stakeholders.

Term/Acronym	**Definition**
The Guardian	A newspaper with a website featuring cryptic crosswords
Blackberry	A mobile phone platform by Blackberry
iOS	A mobile phone platform by Apple
Android	A mobile phone platform by Google
NLP	Natural Language Processing
SRS	Software Requirements Specification
App	Short for application

Bibliography

(2006). *High definition : an A to Z guide to personal technology*. Houghton Mifflin, Boston.