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, 9th May 2014

Contents

List of Figures

List of Tables

Chapter 1

Development

1.1 User Interface

In order for users to use the system a simple and powerful user interface was required. The design reasoning's 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 complied 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 it's 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

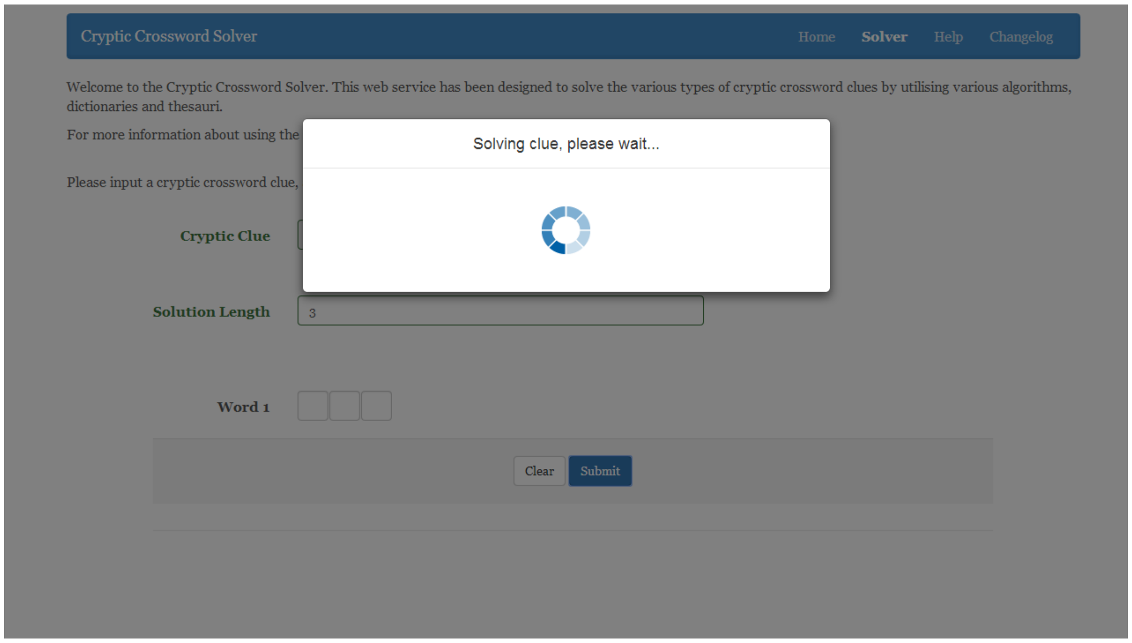
The code snippet above shows use of the XML library, as denoted by the '`x`' name space to certain elements. The code is utilising XPATH to find all possible traces that are listed within the trace element (the trace element is an array).

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 it's owner rendering. For example when a user clicks upon the 'solve' button, the web site 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 it's time.



Welcome to the Cryptic Crossword Solver. This web service has been designed to solve the various types of cryptic crossword clues by utilising various algorithms, dictionaries and thesauri.

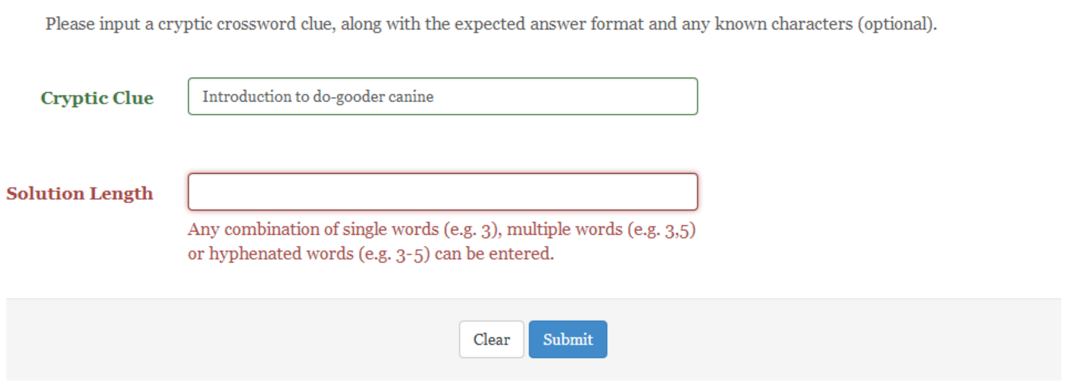
For more information about using the

Please input a cryptic crossword clue,

Solving clue, please wait...

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

The JavsScript 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.


Please input a cryptic crossword clue, along with the expected answer format and any known characters (optional).

Cryptic Clue: Introduction to do-gooder canine

Solution Length:

Any combination of single words (e.g. 3), multiple words (e.g. 3,5) or hyphenated words (e.g. 3-5) can be entered.

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 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 mains 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 proposes 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 request 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, 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 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 the 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 not guarantee, the system makes use the of 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.

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

```java
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 large number of known characters are 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 represents 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 it's 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 15. 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.

```java
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

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. The following piece of code 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);
}
```

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

1.5.3 Acrostic

1.5.4 Pattern

1.5.5 Homophone

1.5.6 Palindrome

1.5.7 Double Defintion

1.5.8 Spoonerism

1.5.9 Charade

1.5.10 Deletion

1.5.11 Container

1.5.12 Reversal

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