University of Huddersfield

MEng Group Project

Cryptic Crossword Solver

Design, Development & Testing Document

Authors:
Mohammad Rahman
Leanne Butcher
Stuart Leader
Luke Hackett

Supervisor:
Dr. Gary Allen

Examiner:
Dr. Sotirios Batsakis

Friday, 18^{th} April 2014

Contents

List of Figures

List of Tables

Chapter 1

Design

1.1 Primary Path

Before being able to correctly design a large, complex system a good understanding of the main scenario through through the system will be required. The main scenario is "a sequence of event or actions" (Lunn, 2003) in which many smaller scenarios may occur.

The main scenario of the proposed cryptic crossword solver system is outlined below:

1. Input the cryptic clue to be solved
2. Input the pattern of the solution
3. Process the data based upon a number of algorithms
4. Output the most confident results

Based upon the previously given main scenario, the primary path of the entire system can be proposed. The primary path is defined as the most commonly used 'route' though a system with as few variances as possible (Lunn, 2003).

The proposed primary path for the cryptic crossword solver system is outlined below:

1. Input the cryptic clue to be solved
2. Input the pattern of the solution
3. Determine the type of clue that has been given (e.g. anagram)
4. Run the given clue type algorithm
5. Rank each of the results based upon a pre-defined criteria
6. Output the top ranking results

The primary path outlined above takes a relatively generic approach with regards to how a clue should be solved. Ideally the clue would be categorised into a certain type of clue, thus requiring that particular solver to be run.

However the primary path does not illustrate the steps that are required to be taken if the clue can not be categorised — does the system simply stop or would it try an alternative approach?

These additional steps may often be 'optional', and hence are described as alternative paths. The alternative path will contain small and subtle changes to the primary path (Lunn, 2003).

If the system is unable to determine the type of clue that has been then an alternative approach is to be used. In this instance, the primary path would house the following additions and changes:

3 Determine the type of clue that has been given (e.g. anagram)

3.1 Start a new instance of each solver

4 Run the each of the solvers separately

The above alternative path shows that if the clue can not be categorised, then the system will attempt to solve the clue utilising a brute force approach. The above alternative path, could also be regarded as a 'worst case scenario'.

The outlined primary and alternative paths are a simple way of illustrating the the general ideas behind the system. These paths will be actively referred to throughout the design process.

1.2 Activity Diagrams

Within the Primary Path section a primary path analysis was conducted, ensuring that common paths throughout the proposed system are identified. Within this section a number of activity diagrams will be presented that will break the primary path into a number of simple processes.

An activity is a task in a process that allows for small meaningful work to happen. A process may have a number of activities linked together to form a work flow (Lunn, 2003).

Figure 1.1 on page 9, illustrates the main activity diagram for the complete system. The activity diagram follows what was discussed within the Primary Path section, and expands upon some of the generalised points. Over the following subsections key parts of the main activity diagram will be discussed in more detail.

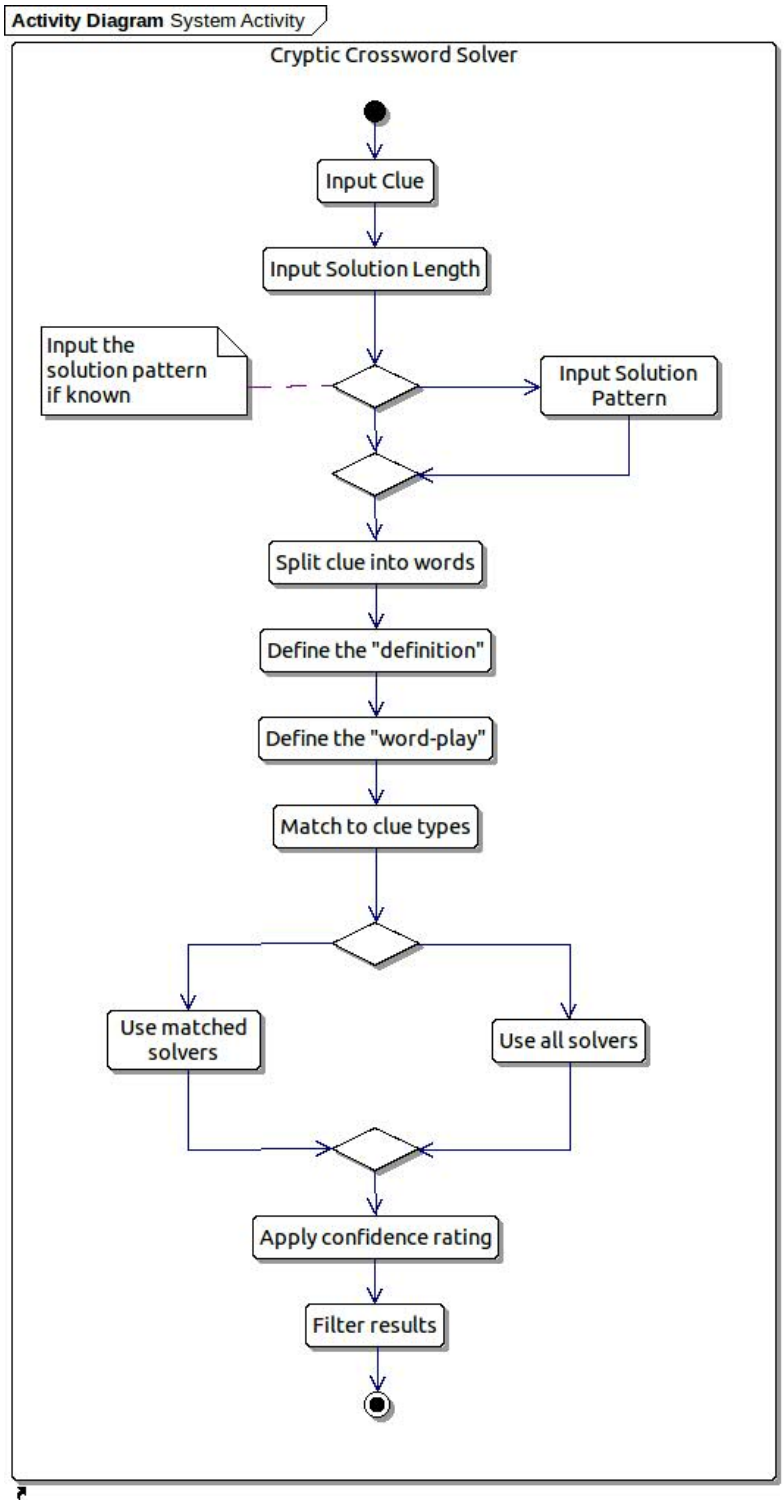

Figure 1.1: The complete Activity diagram

1.2.1 Clue Input

Figure 1.2 illustrates how the data should be input into the system. In order for the system to solve a given clue it requires the following three inputs:

- the clue
- the solution length
- the solution pattern

At first it may seem that the solution length is unnecessary, as it could be computed from the solution pattern. The reason for including the solution length is that it provides an additional validation element — much like having to type in a password twice when creating a new online account.

The solution pattern allows a user to predefine the required answer, based upon a number of characters. The solution pattern will accept all letters as well as three predefined 'special characters', as outlined below:

? ⇒ wild card character

- ⇒ hyphenated word

, ⇒ word separator

For example, the pattern "'????e?-in-???" could match to "mother-in-law" or "father-in-law", just as "??????? d?????" could match to "sitting ducks". As identified upon the activity path, a user may simply present the correct number of wild card characters to denote a unknown pattern.

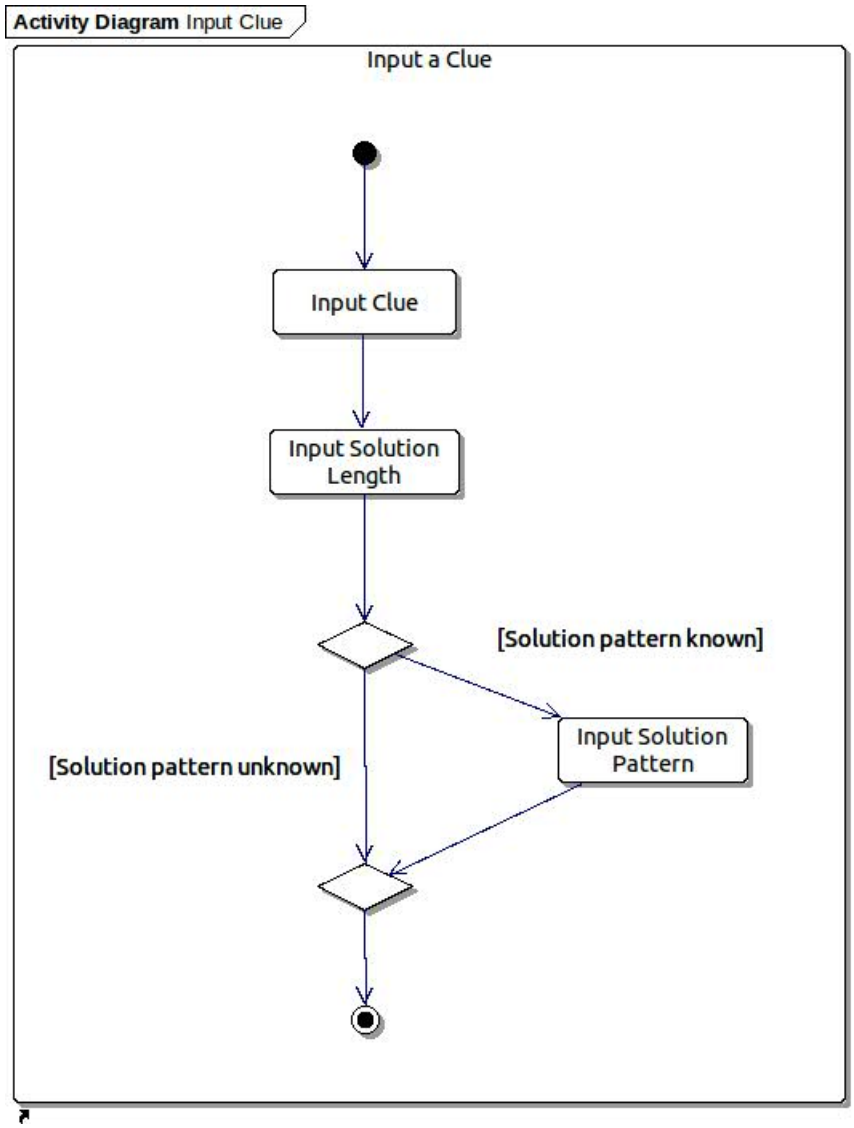

Figure 1.2: Activity diagram showing how a user would input a clue

1.2.2 Choosing a Solver

Once the user has successfully input the three variables, the system will then try to determine the type of clue. There are a number of actions that must be performed in order to deduce the type of clue, as shown in figure 1.3.

The system will try to match the given clue to a list of properties that each clue has. For example spoonerism clue types, will have the word "spooner" within the clue.

If the match is successful then the system will bump the ranking of that particular clue type. Once the system has finished applying the categorisation 'tests' to the clues, it will select

the top solver to try to solve the clue.

If the chosen solver is unable to solve the clue, then the next ranking solver will try to solve. This process will continue until a set of results have been returned, or if all solvers have been used.

If the categorisation fails for some reason, then the system will try to brute force it's way to answer, by using all solvers.

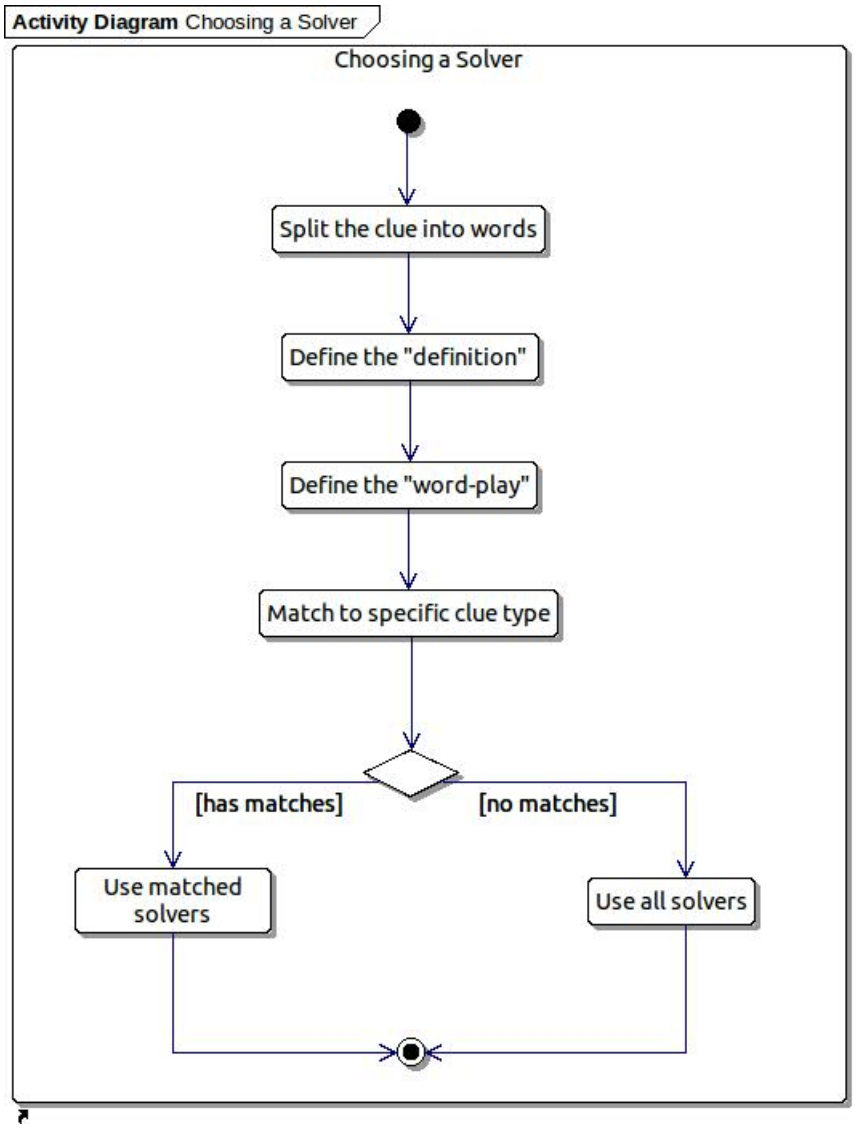

Figure 1.3: Activity diagram showing how a solver is to be chosen

1.2.3 Handling Results

Once the solving process has finished, the system will then attempt to apply a 'confidence rating'. The confidence rating of a clue is how close to the correct answer the system thinks it is.

For example, if a clue had a confidence rating of 100% it would deemed as the only answer, whilst 0% would indicate there is no chance of the answer being correct.

It is intended that if there are a large number of results, then the system may choose to filter the results down. For example if thousands of results are generated, then the system may choose to only return results that have a confidence rating higher than 70%.

Figure 1.4 shows the handling of results as an activity diagram.

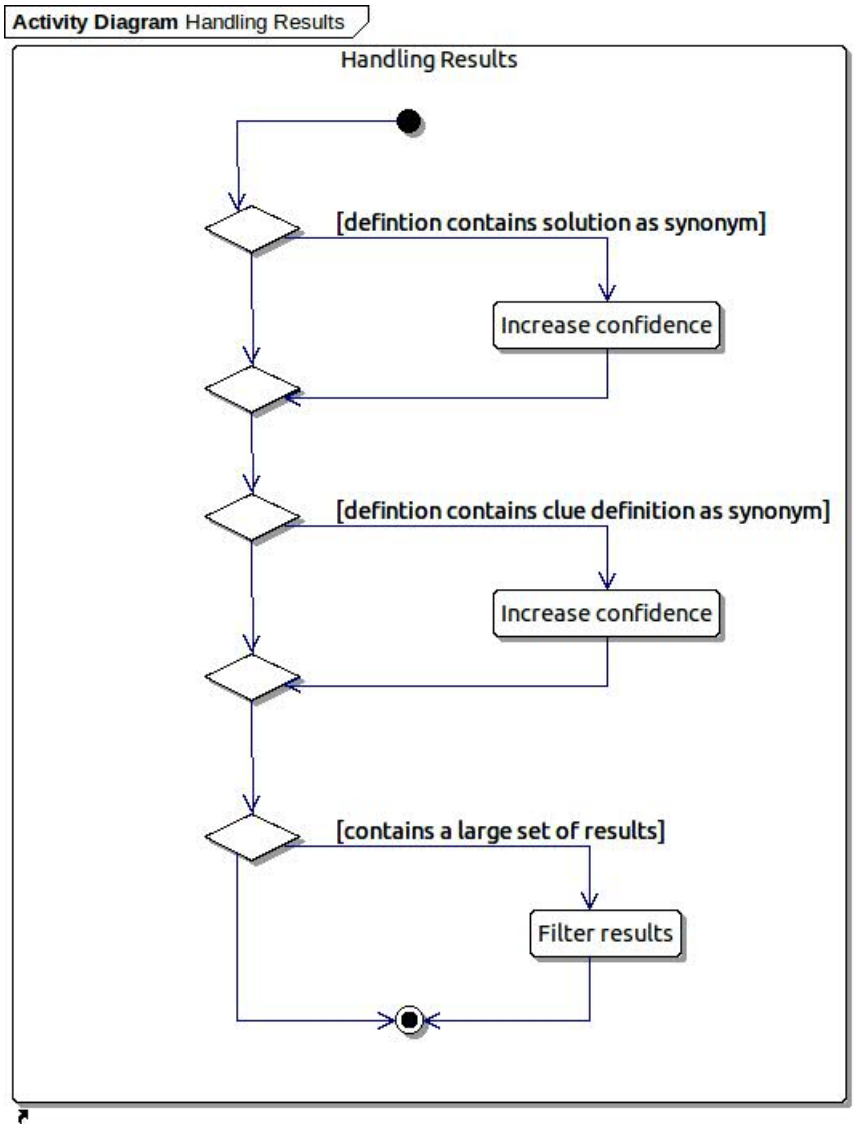

Figure 1.4: Activity diagram showing how results should be handled

1.3 Use Case

A Use Case diagram is "a definition of a meaningful interaction with a computer system", and provides various approaches to analysing system design requirements (Lunn, 2003). Within this section the use case of the entire proposed system will be presented and discussed. The use case has been split into two diagrams to increase the readability of the diagram.

The primary intended use of the system is to be able to submit a clue to be solved, which requires the clue, solution length and solution pattern. Figure 1.5 illustrates the three parameter requirements in order to submit a clue to be solved.

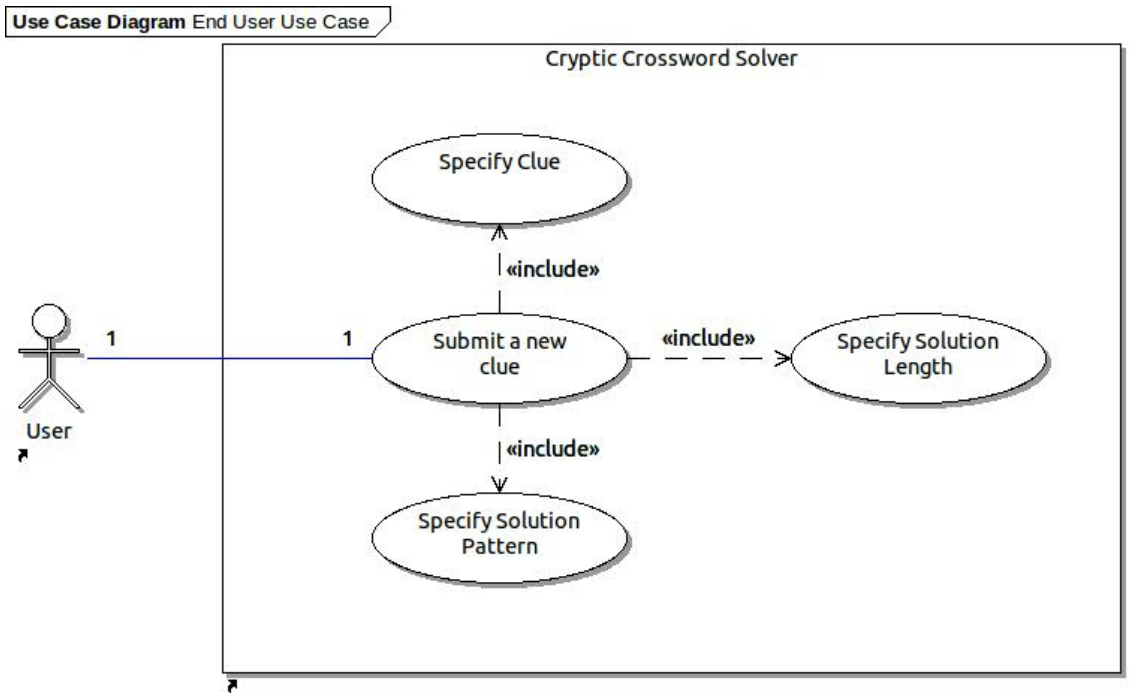

Figure 1.5: Use case illustrating the actor's range of possibilities

It must be stated that the solution pattern could be unknown. In this instance a set wild card characters could be given to the system. This was previously shown in figure 1.2 on page 11.

The remainder of the Use case is shown in figure 1.6. Once the use has input the various data elements the system will then attempt to solve the clue.

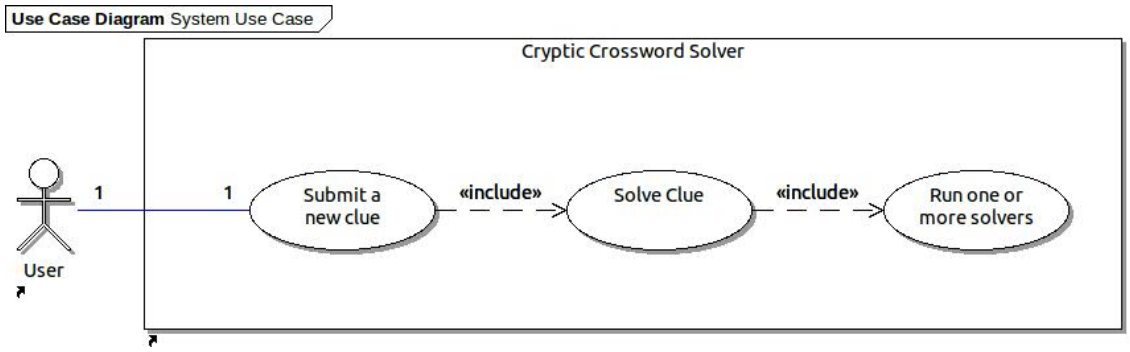

Figure 1.6: System Use Case

The use case demonstrates that there is very little user input required, and once the data parameters have been entered there will be no additional user interaction required.

A design decision has been taken so that there is minimal user interaction with the service. This will be particularly useful for those who are using a mobile device, as inputting data can be laborious in comparison to a desktop machine.

1.4 Sequence Diagrams

Within the previous Use Case section a number of key, generic functions were identified. In order for a sequence diagram to be developed, the generic functions will be converted into a set of objects and interactions. Each of the objects and interactions will eventually form the classes that will be used to build the system. Therefore a sequence diagram can be described as a "way of describing a journey through a system" (Lunn, 2003).

The two sequence diagrams that will be described within this section focus upon submitting a clue and solving a clue.

1.4.1 Submitting a Clue

Figure 1.7 illustrates a user sending data to the web service (servlet). Within this diagram a number of key elements such as how a clue will be solved have been removed. This is the ensure that the diagram is as simple as possible, as this diagram will only be focusing upon how a user sends and receives data.

The Solving a Clue subsection on page 18 describes in detail how a solver will solve the clue.

To simplify the diagram the user is submitting a 'block' of data, which in fact would be the three main parameters — the clue, the solution length and the solution pattern. This has been shown upon the diagram as a 'block' of data, because all three parameters will be sent to the system at the same time.

Once the data has arrived at the servlet, the data will be validated. If the data is deemed to be invalid, the servlet will send an error response back, finishing the current instance.

If the data is deemed to be valid, then the system will try to categorise the clue, and solve the clue using the required solver(s). This is will be discussed in more detail in the Solving a Clue subsection on page 18.

Once a set of solutions have been generated, the the in-memory set will be converted to either JSON or XML depending upon the client's request. Once converted the data is returned as part of the response, satisfying the client's request.

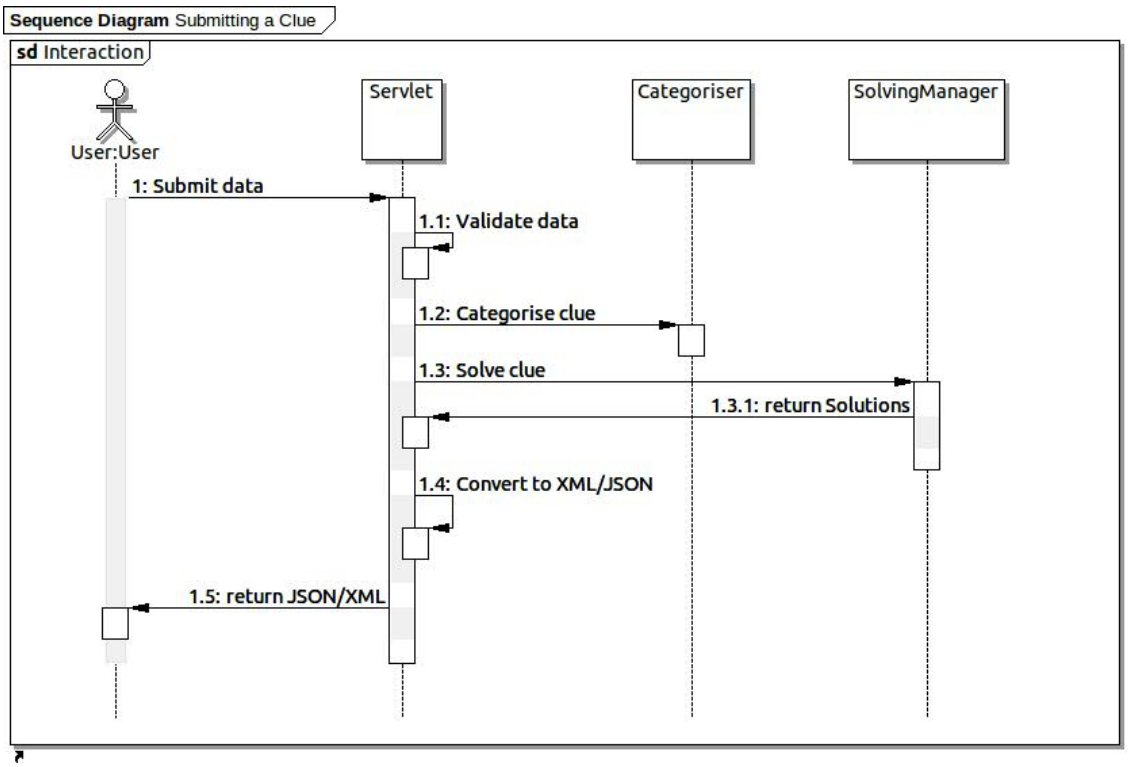

Figure 1.7: Sequence digram illustrating a user submitting a clue

1.4.2 Solving a Clue

Figure 1.8 illustrates how a given clue will be solved. This sequence diagram assumes that valid data has been passed to the 'SolvingManager'. This process was described in the previous subsection entitled 'Submitting a Clue' on page 17.

The 'SolvingManager' object will manage the process of solving a given clue. In order to do this it will have to house several processes, including distributing the clue out to one or more solvers, and merging all results.

Figure 1.8 illustrates the distributing and solving of the clue as a synchronous process, however this was used for illustrative purposes only. The system will in fact distribute and solve the clue upon an asynchronous basis. This will mean that each solver will be running at the same time, and thus reducing the total amount of time needed to solve the clue.

The system will wait however, for all solvers to finish, ensuring that all solutions can be returned to the user. The merging of the results will simply remove any duplicate solutions, upon a first come basis. So if a solution is already in the list, it will not be re-added because a different solver has managed to find the same solution.

Finally the confidence ratings will be adjusted. The adjustments are made upon a number

of factors, and were described in figure 1.4 on page 14.

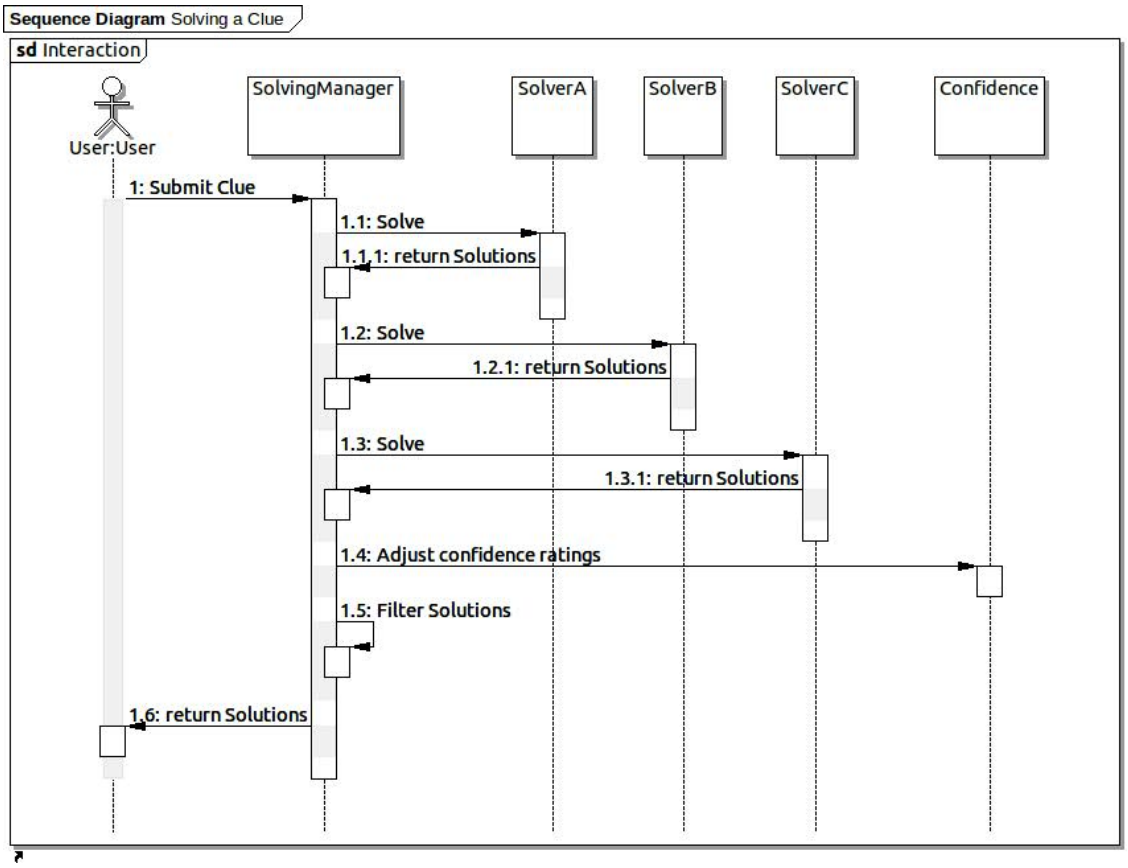

Figure 1.8: Sequence digram illustrating the system solving a clue

1.5 Class Diagrams

Within this section a range of class diagrams will be presented an discussed. “A class diagram describes the structure of a system by highlighting the system’s classes, attributes, methods, and relationships with other classes” (Lunn, 2003).

Figure 1.9 illustrates the seven packages that make up the cryptic crossword solver. Over the following subsections, each of the packages will be described in more detail.

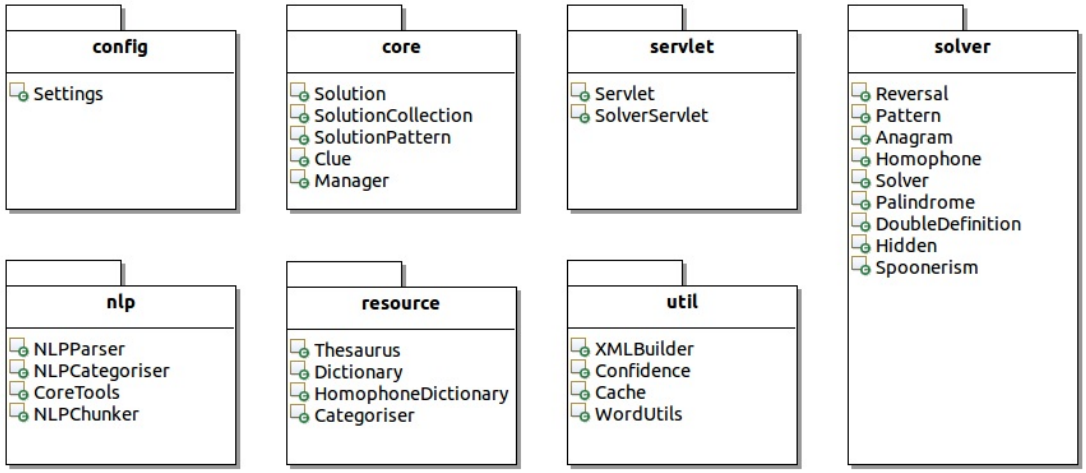

Figure 1.9: Package overview of the Cryptic Crossword Solver system

The architecture of the system will utilise a ‘plug and play’ approach. This approach, will allow for various numbers of solvers to be added at run time, rather than a compile time.

From a development point this will allow for the underlying system to be developed and tested utilising a limited number of solvers. Once the underlying system has been completed, solvers can then be added at any time, without having to alter any previously written code.

1.5.1 Config

The configuration class only contains one class, Settings. The Settings class is designed to hold all application specific settings, and each setting can be altered based upon the mode it is being run under — for example development, testing or production.

The Settings class follows the single design pattern which ensures that the class will have only one instance (Gamma, 1995). This prevents various numbers of the same Settings objects being used in memory, and thus wasting valuable space.

Figure 1.10 illustrates the config package, containing the Settings class.

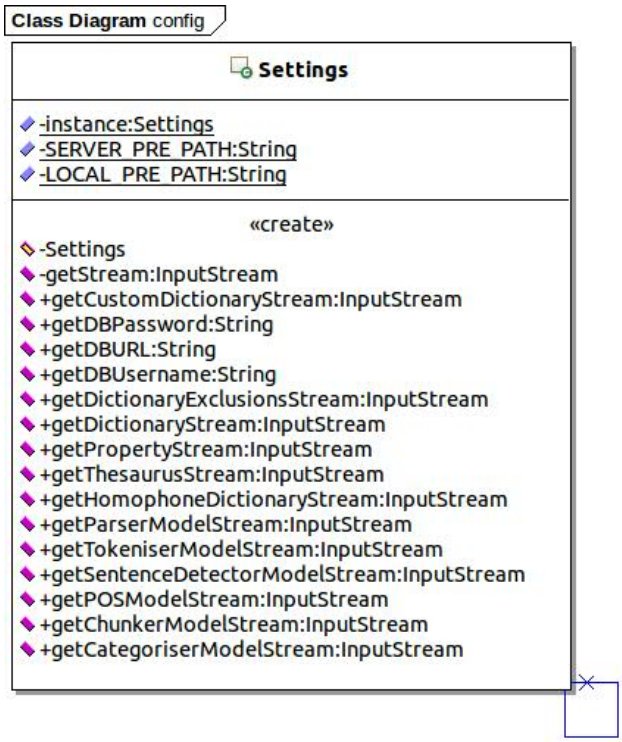

Figure 1.10: Config package class diagram

1.5.2 Core

The core package contains the various classes that are deemed to be at the heart of the system. A class has been devoted to the clue, a solution and a solution pattern class various specific functionality.

For example the solution pattern provides methods that allow for words to be computed, without having to work out the length of each word every time. The clue class is able to deduce the best solutions, based upon a collection of solutions.

The SolutionsCollection provides a simple interface to handle any number of solutions. The SolutionsCollection uses the HashSet as it's base class, which prevents duplicate solutions to be stored within the same set. It also provides a faster lookup, in comparison to List based structures.

Finally the manager class is the heart of the system and was originally illustrated in the subsection Solving a Clue on page 18. It will be able distribute a clue and it's solution pattern to the various solvers asynchronously. It also handles the merging of results within the `distributeAndSolve` method.as

Figure 1.11 illustrates the core package, containing the various core classes.

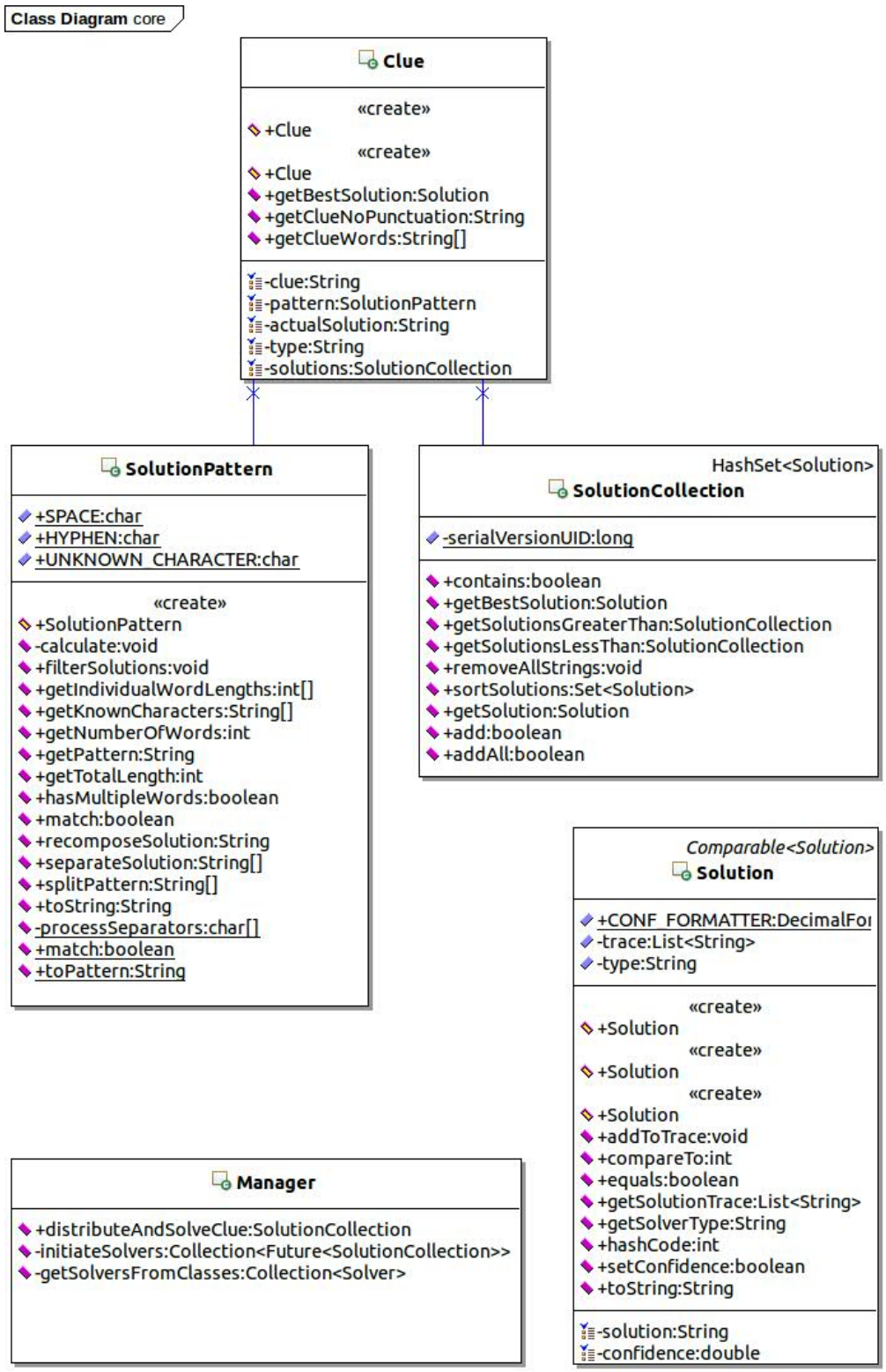

Figure 1.11: Core package class diagram

1.5.3 NLP

The NLP package contains the various classes that are focus on providing the system with an interface to the Apache OpenNLP library. This will allow the system to be able to use natural language processing techniques from within the application.

Figure 1.12 illustrates the NLP package, containing the various NLP related classes.

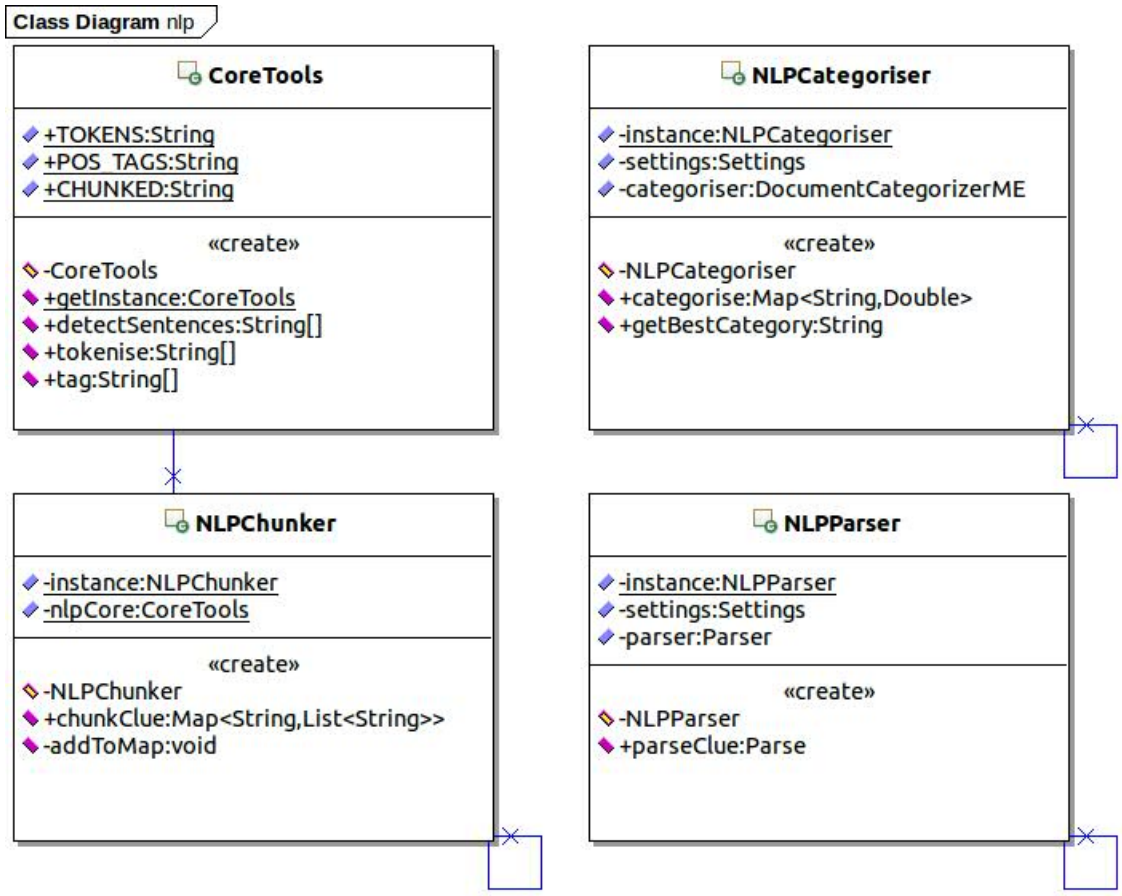

Figure 1.12: NLP package class diagram

1.5.4 Resource

The resource package contains various classes that provide interfaces to the array of resources that the system will use. As with the config package, all classes utilise the Singleton design pattern.

The reason for this is that it is to be expected that these classes will be used multiple times throughout the life of the system. To prevent multiple objects being created and being left

in memory, the singleton design pattern was selected. The pattern will "ensure that a class only has on instance, and provide a global point of access to it" (Gamma, 1995).

The Dictionary, Thesaurus and HomophoneDictionary classes provide the functionality that would be expected. For example the Dictionary has the ability to check to see if a given word is a 'real word'. Where as the Thesaurus class has the ability to find synonyms of a given word.

All of the classes are able to utilise the SolutionPattern class (as found in the core package), which allows for words to be matched upon various known and unknown character combinations.

The Categoriser class will map a given clue and solution combination to various lists of clue type indicators. If an indicator is found, then the confidence rating of the solution will be increased. This will form part of the overall ranking of a solution.

Figure 1.13 illustrates the resource package, containing the various resource classes.

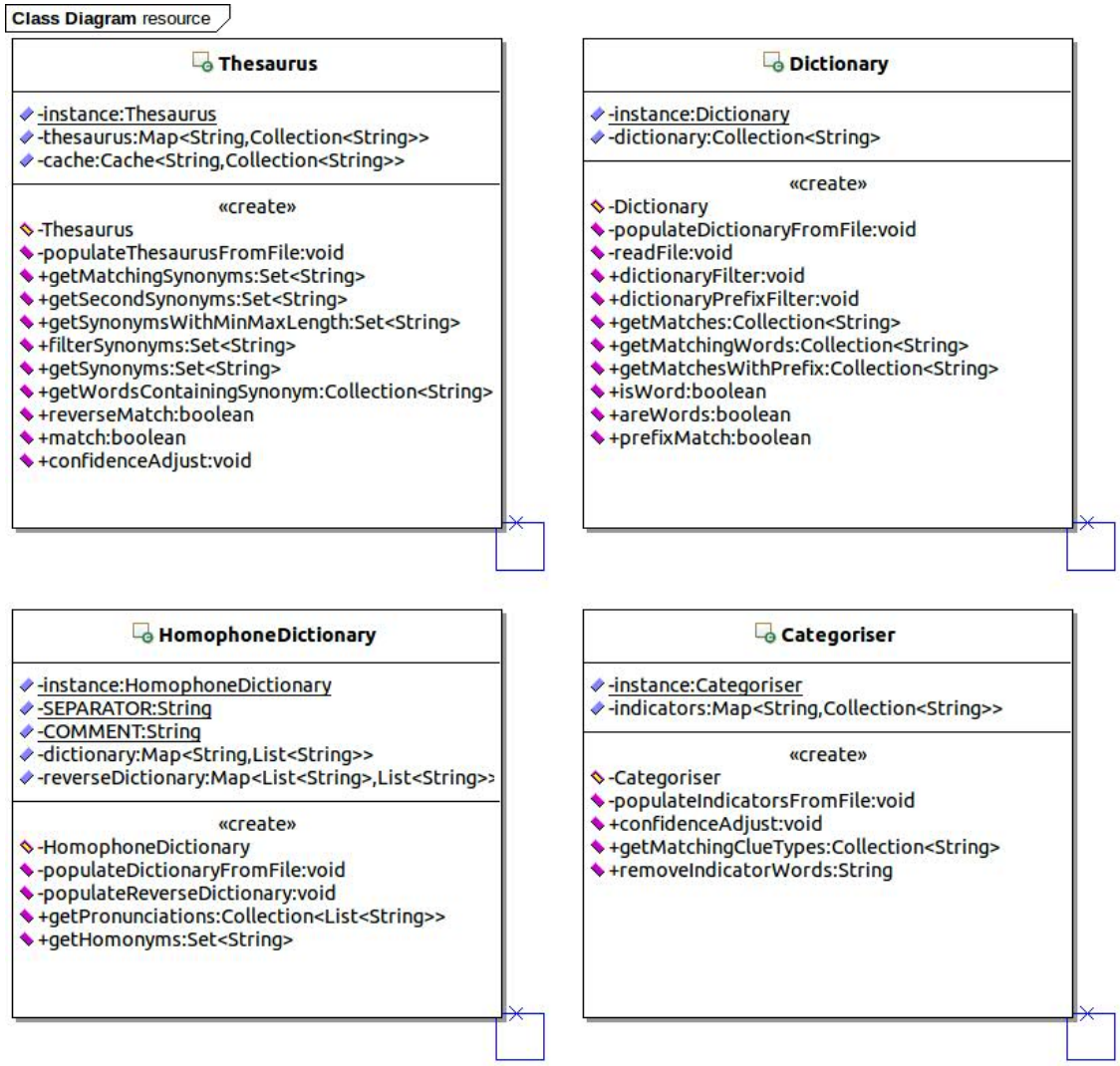

Figure 1.13: Resource package class diagram

1.5.5 Servlet

The servlet package contains two classes that provide the web service interface. Both classes tie into the Tomcat system, which allow provides a container for the website to sit in.

The Servlet class is a custom base class that various levels of common functionality, that other specific classes may which to utilise. This functionality includes converting XML into JSON, and determining if a reject was made via AJAX.

The Solver class extends the Servlet class, and provides the functionality required for handling solver related queries. All GET and POST requests to the Solver class made via AJAX will be treated as a request over the web.

However all GET and POST requests that do not pass the AJAX flag, will be treated as if the user intended on viewing a website. All non-AJAX posts will be redirected to the Solver's default HTML page, which will allow users to enter the various parameters via a web form.

It is intended that the solver class will listen upon all requests upon the /solver URL path — for example www.example.com/solver.

Figure 1.14 illustrates the servlet package, containing the public web service related classes.

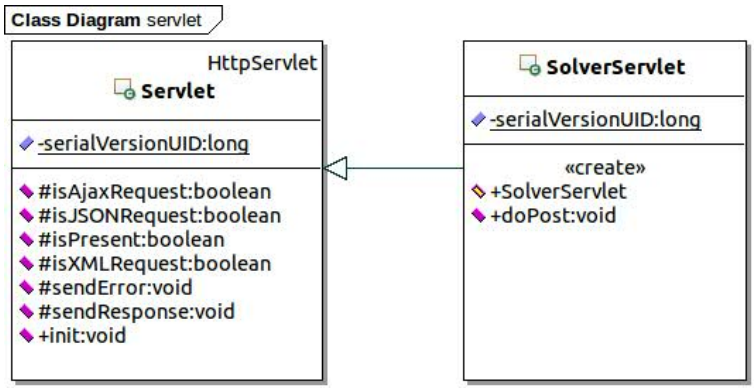

Figure 1.14: Servlet package class diagram

1.5.6 Solver

The solver package contains the all of the solver algorithm classes that are used as part of the over all solving process. Each solver extends a main Solver class which provides a number of base methods, as well as some additional methods that require each solver to override.

The Solver class implements the Callable interface, which provides functionality to run a class in it's own thread. A design decision was taken that the Callable interface was to be used, as once it has finished executing it is able to return an object unlike similar interfaces such as Runnable.

Each of solver classes implement the Abstract Factory design pattern via the base class Solver. This pattern "provides an interface for creating families of related objects without specifying their concrete class" (Gamma, 1995).

Each of the solvers is able to return a concrete product, which in this instance is a a SolutionCollection object. The SolutionCollection will contain all the solutions that the given solver has managed to compute.

Figure 1.15 illustrates the solver package, containing the various solving algorithms used by the system.

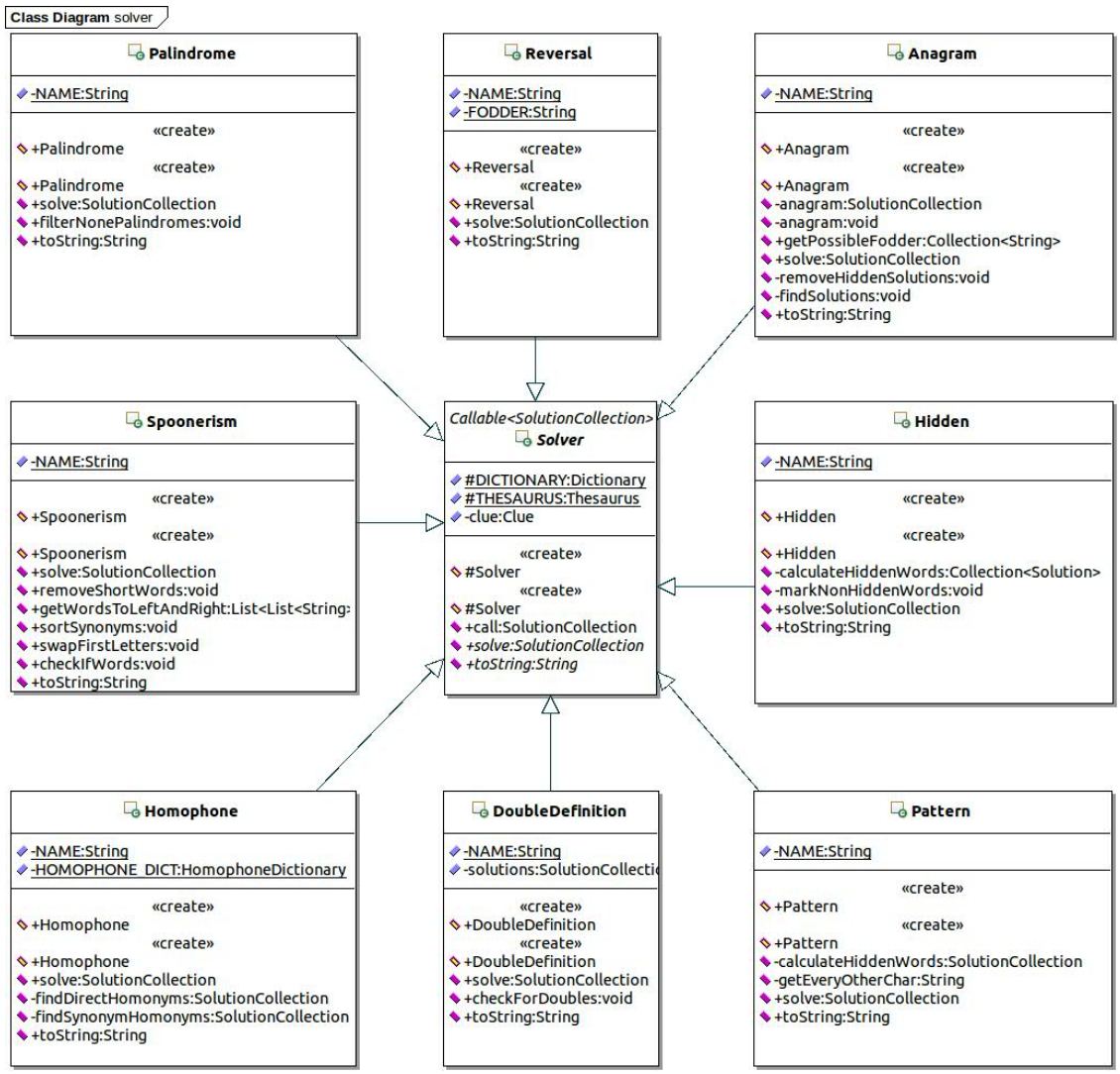

Figure 1.15: Solver package class diagram

1.5.7 Util

The util package contains a number of classes that are unrelated with each other, but provide functionality that is required throughout the system.

The XMLBuilder class provides a data encapsulation layer around an XML library. This class is able to build well-formed XML documents that are suitable for the system's out.

The WordUtils class provides a number of generic methods that are used to manipulate strings as words. This functionality is particular useful, as the system will be dealing with large amounts of natural language (English) sentences.

The confidence class provides a number of rules, that when applied to a solution is able to deduce it's overall 'correctness'. This value is directly reported back to the user as the confidence rating.

Figure 1.16 illustrates the util package, containing the a number of additional utility classes.

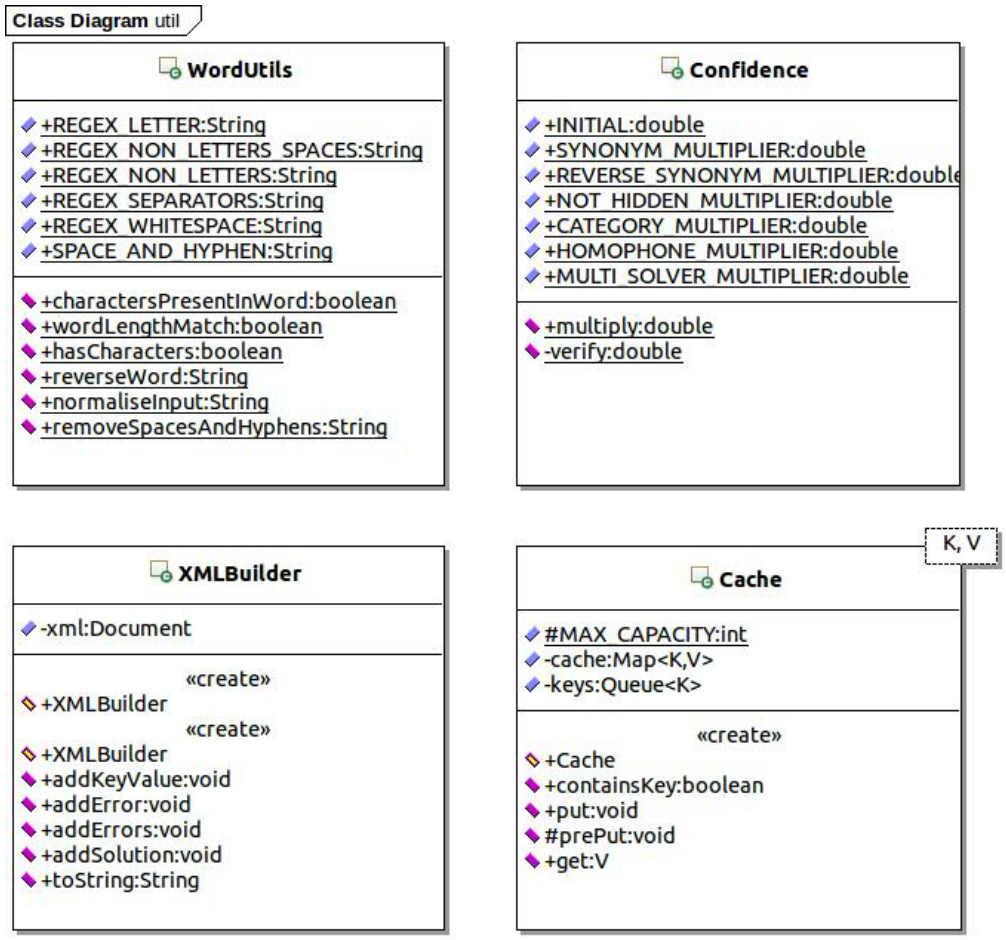

Figure 1.16: Util package class diagram

1.6 Database Design

As part of the testing strategy a list of clues and their solutions are required to be stored. In order to accurately store a large amount of training data, a database will be used.

The database design is of a minimalistic approach as it is not intended to hold a large amount of data, nor will it be used within a production environment. Figure 1.17 shows the proposed database design.

Each record will contain a clue, a solution, and the solutions length. The solutions length has been assigned a VARCHAR data type, because solutions can span across more than one word. For example 3,4 would relate to two words, of three and four character respectfully. Each of these attributes are required — i.e. not null.

The orientation attribute is an ENUM that contains either 'down' or 'up', indicating the direction that the solution is arranged. The orientation may be useful for certain clue types, such as reversal, where by the clue references the direction — e.g. north-to-south, or east-to-west.

The clue number attribute stores the number of clue, as some clues use the clue number in obtaining the solution answer. Although this is uncommon, it will be taken into consideration.

The clue type attribute is an ENUM that holds the various types of clues, such as 'purely cryptic' or 'anagram'. It is intended that these values will be 'mapped' to the system code, so that clues can be used automatically when training.

cryptic_clues	
♦ **id**	**INT(10)**
• clue	VARCHAR(255)
• solution	VARCHAR(32)
• length	VARCHAR(32)
◦ orientation	ENUM
◦ clue_number	TINYINT(3)
◦ type	ENUM

Figure 1.17: Testing database entity-relationship diagram

1.7 User Interface

The previous sections have presented a number of system designs from a programmatic point of view. Within this section, there will be a focus upon user interface designs, and thus how the end user will interact with the system.

Fundamentally there are two aspects to the user interface design of the system — inputting the clue and retrieving the results. Each of these aspects will be discussed in more detail in the following subsections.

1.7.1 Platform Support

One of the main objective of the project is to develop a system that can be used upon a number of mobile platforms, as well as trying not to neglect 'traditional' desktop users.

In order to complete both of these objectives, a responsive design is required. A responsive design is one that is able to dynamically change based upon the screen size.

This means that there will only be one code based for multiple screen sizes, and thus allowing for increases in maintainability. All of the designs within this section feature a responsive design. Figure 1.18 illustrates the power of responsive designs.

The left-hand side form shows a 'traditional' desktop experience, whilst the form upon the right-hand side, shows a mobile experience.

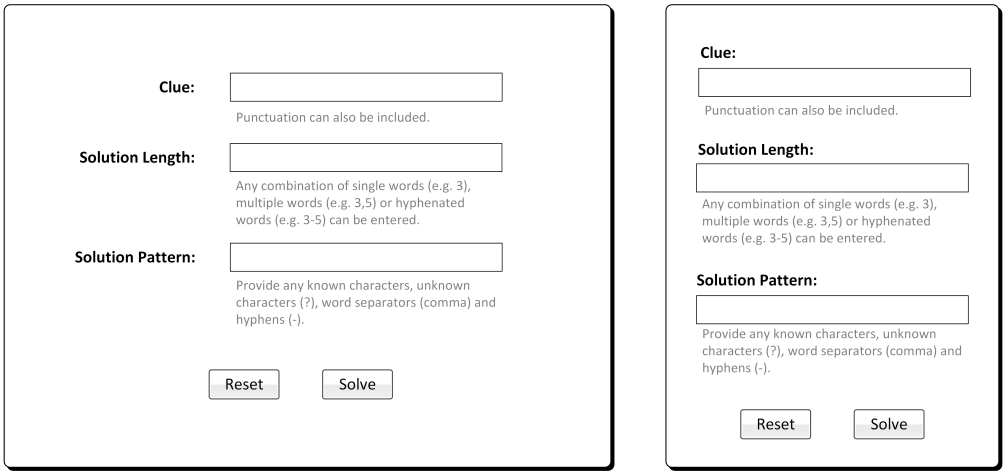

Figure 1.18: The input form to be completed by the end user

1.7.2 User Input

In order for the system to solve the clue, it must first be given the clue, along with additional supporting information. The additional information such as the solution length and pattern is vital to the system in order for it to compute the correct answer.

However the intended users of the system are likely to be operating upon some form of mobile device, meaning that a simple and power interface is required. It also means that space will be at a premium, and thus it can not afford to be wasted.

Figure 1.19 illustrates the design of the user input form. One of the most noticeable elements to the form design is how much space has been allocated to the input boxes.

The design of the inputs utilises a bi-column setup using the ratio of 1:3. This means that for every 1 pixel of space allocated to the left-hand labels, there will be 3 pixels of space allocated to the right-hand input boxes. This allows users of mobile devices to quickly select the input box, rather than having to tap a small area several times.

In order to assist the end user additional help blocks will be included, and give useful information to the user, such as describing the solution pattern format required.

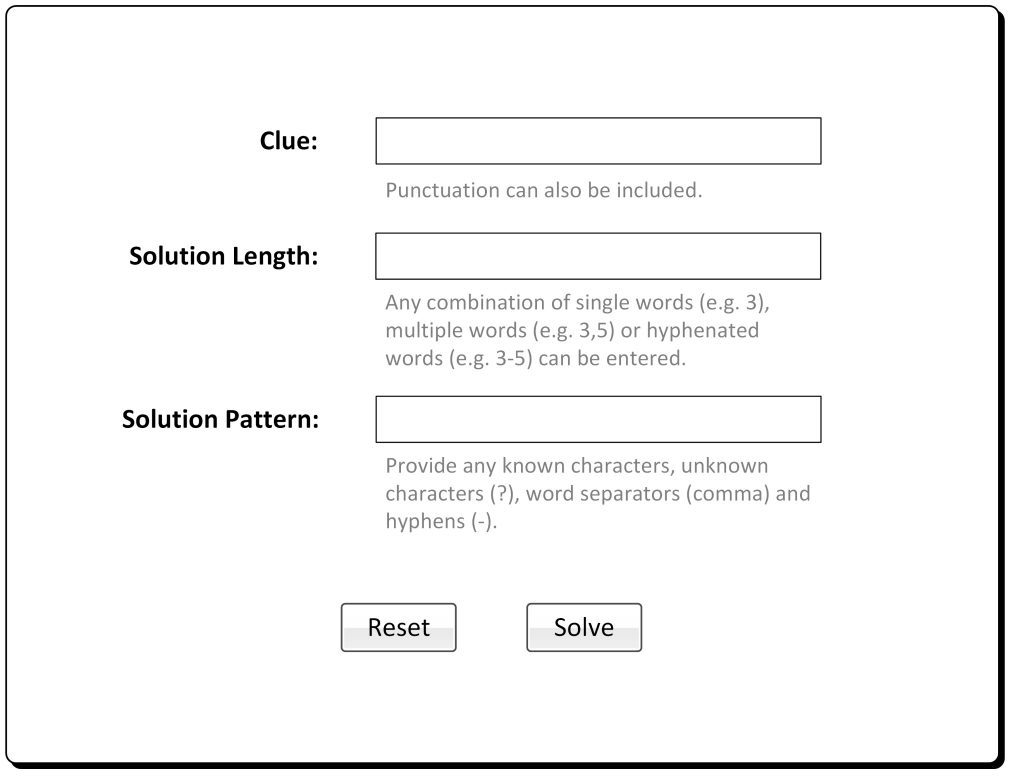

Figure 1.19: The input form to be completed by the end user

As the system is dealing with input from users, it is inevitable that a user will attempt to

input incorrect data. This will require validation to be performed, and if validation has failed then the user should be notified.

As previously mentioned space is a premium upon a mobile device. Therefore displaying a list of validation error messages at the top of the screen would not be utilising limited amount of space wisely.

In order to combat this issue, validation will be displayed 'inline' with the form input elements, as seen within figure 1.20. By utilising an 'inline' approach it recycles the screen space that has been made available by the form.

For validations that have passed the input outline will change to green, and for validations that have failed the input group will change to red. This will immediately alert the user to the various issues, and thus allows the user to fix the errors in a quicker and more efficient manner.

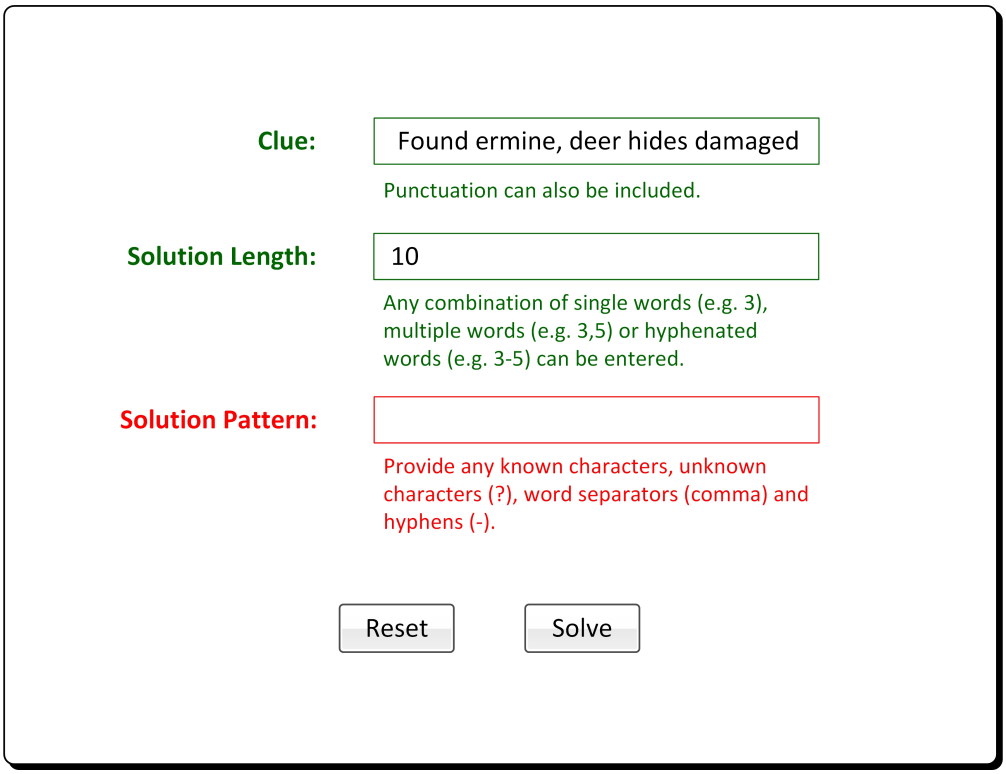

Figure 1.20: The input form indicating a validation error

1.7.3 Results

The displaying of the results follows on from some of the previous design decisions. Each potential solution is rendered into it's own panel, and will contain the confidence rating, and the solver type that managed to deduce the solution (as shown in Figure 1.21).

Within an open panel, additional information is displayed — known as the solution trace. A solution trace provides a step by step account of how the solution was able to be computed. The solution trace may help to teach users how a particular type of clue is solved.

The results are order by confidence rating in ascending order, and by default the first panel will be 'open', showing the solution trace. The reason for this is that the top answer is likely to be the answer the end user is looking for

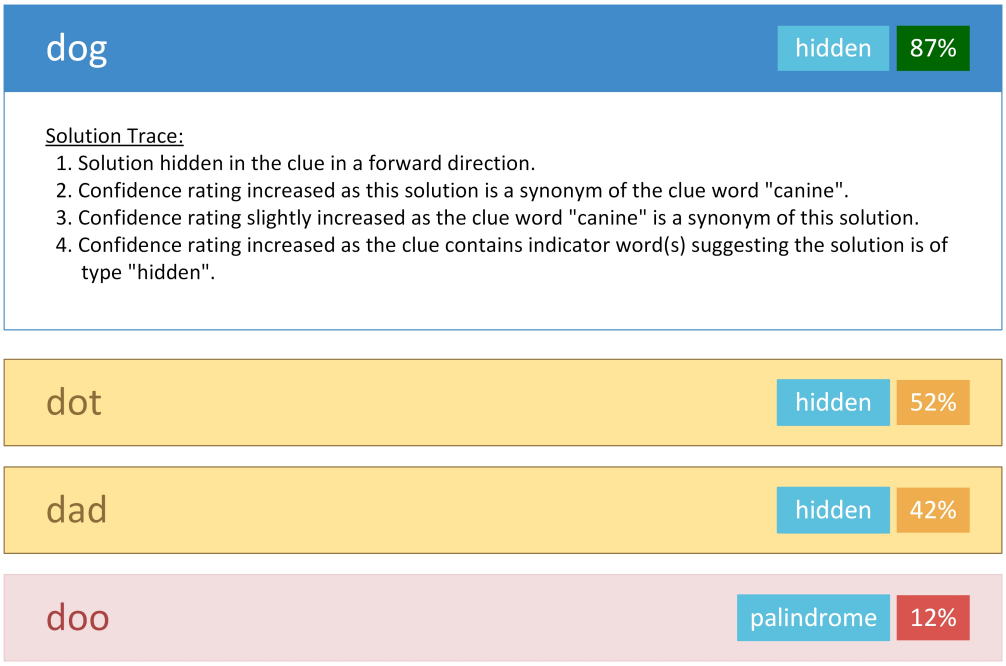

Figure 1.21: Results list displaying the top answer in blue

Each of the panels are "clickable", meaning that if any given panel is selected then the solution trace will be displayed, whilst previously selected panels will be 'closed' (as shown in Figure 1.22). The main reason behind this is to reduce the amount of scrolling a user has to do, especially for those without a mouse.

Each solution is awarded a colour based upon the confidence rating. Blue refers to a top answer, whilst the remaining colours (green, yellow and red) indicate the likely hood of the solution being correct.

This will allow the end user to immediately deduce that 'green' results are more likely to be correct in comparison to 'red results', and that a 'blue' result is likely to be correct.

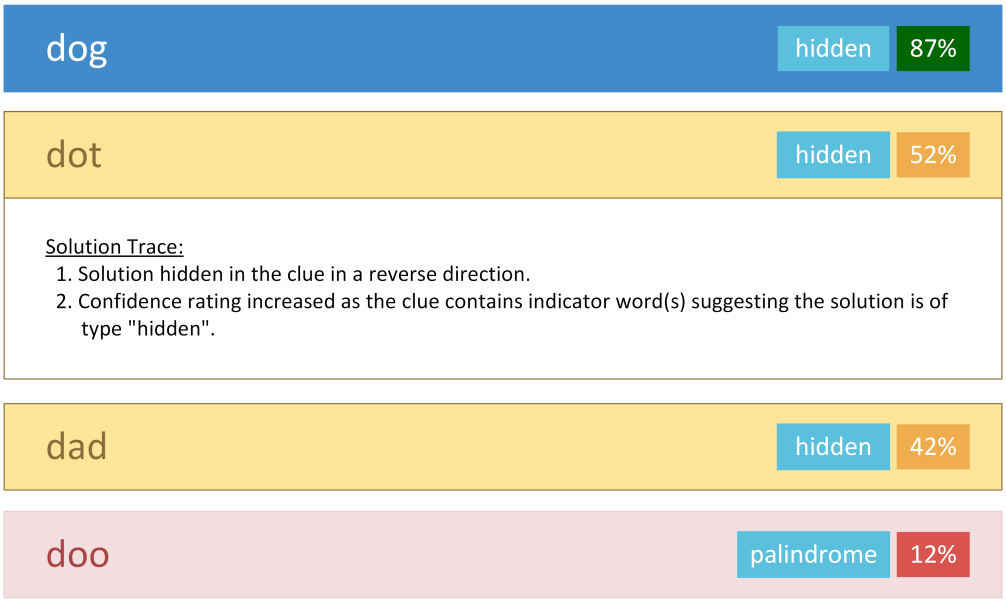

Figure 1.22: Results list displaying alternative solutions

Chapter 2

Implementation

2.1 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.

2.1.1 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.

2.1.2 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.

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

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;
}
```

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.

2.2 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 User Interface design section, which can be found on page 31.

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.

An example code snippet has been presented below 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>
```

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 2.1. Under a non-JavaScript supporting browser this would simply look like a normal page request that is taking it's time.

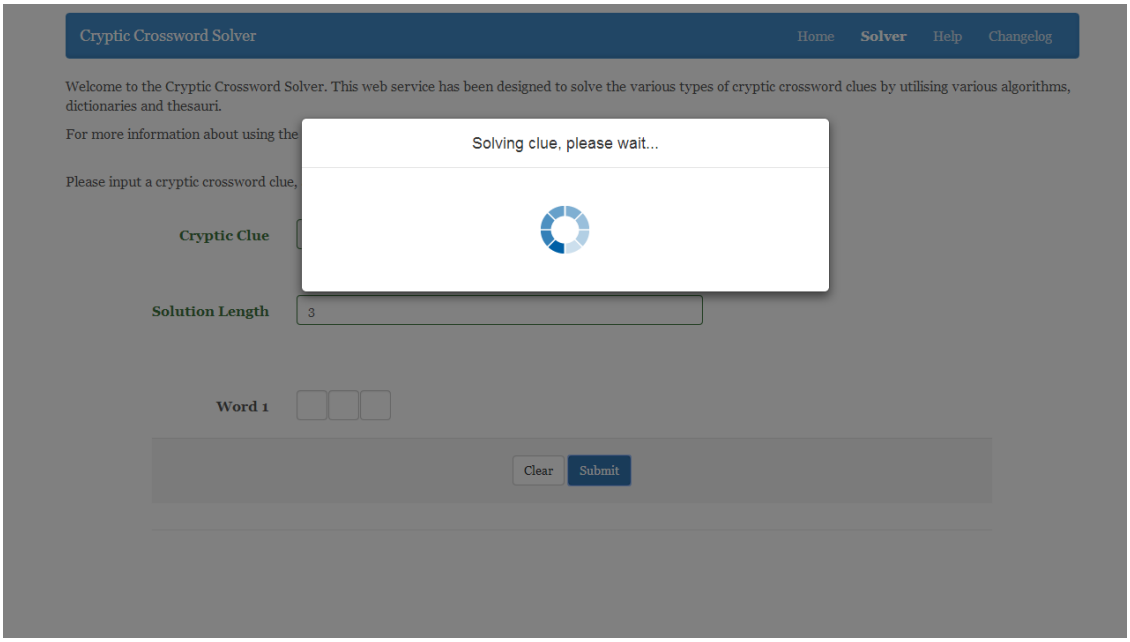

Figure 2.1: Message informing the user that a clue is currently being solved

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 2.2.

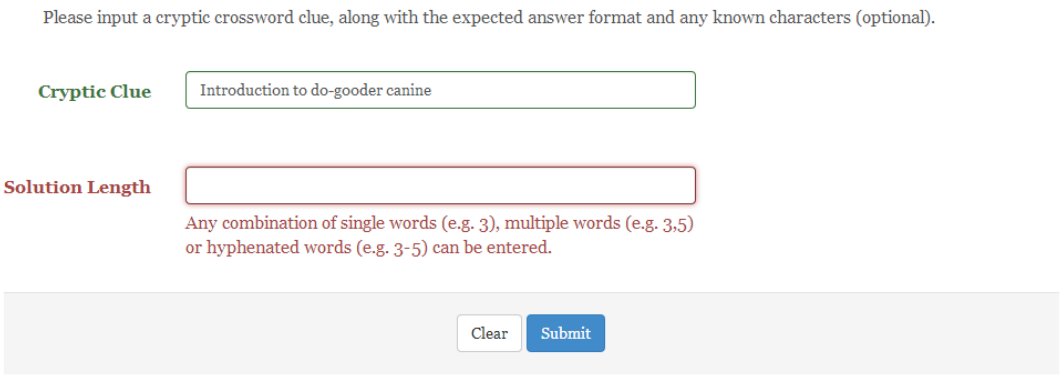

Figure 2.2: Live validation feedback ensures users are entering correct data

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.

Chapter 3

Testing

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

Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Lunn, K. (2003). *Software development with UML*. Palgrave, first edition.