Displaying Heraldic Blazons

WILLIAM MATHEWSON

4th Year Project Report
Computer Science

SCHOOL OF INFORMATICS

UNIVERSITY OF EDINBURGH

2018

Abstract

In this project I describe how I designed and implemented a system to render heraldic blazons. Blazons are the text describing how shields in coats of arms are to be drawn. Through use of good software engineering principles, such as decoupling and delegation, I built a modular system that rendered the blazons with scope for further extension in the future. When comparing against a similar system, it had significantly better performance with more features.

Acknowledgements

I would like to thank my supervisor, Julian Bradfield, for his help and advice as I was writing this project. I would also like to thank all my friends on Level 9 of Appleton Tower, particularly Connie, Nicole, Paul, Dan and Simon, who helped keep me going through the long, arduous journey that was this project and 4th year as a whole.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*William Mathewson*)

4TH YEAR PROJECT REPORT

COMPUTER SCIENCE
SCHOOL OF INFORMATICS
UNIVERSITY OF EDINBURGH

CONTENTS

1 INTRODUCTION

1.1 Motivation

In 1874 — 4 years after his death — John Papworth's *Ordinary of British Armorials* was published.[1] In this work, he recorded approximately 50,000 descriptions of families' coats of arms, none annotated.

[1] S. M. Collins. 'Papworth and his Ordinary'. In: *The Antiquaries Journal* 22.1 (1942). Accessed: January 2018, pp. 6–7. DOI: 10.1017/S0003581500003668.

This project makes it possible to have these descriptions, or *blazons* as they are termed in heraldry (see 2.1), drawn freely for people to view. This has potential application for ancestry companies. Given the blazon, they would be able to construct the shields visually.

1.2 Project Aim

This project was written as a paired project and as such, I only wrote the side of the project that deals with rendering the blazons, rather than the parsing. The parsing was implemented by my partner, Anthony Gallagher, and, as such, this will not be covered in this report. Important shared design elements will, however, be covered in Section 3.1.

The aim for this project was to design and implement a web app to render the parsed blazon by drawing charges and ordinaries.

1.3 Contributions

In this honours project, my contributions included:

- Drawing the charges and quarters used on the shield, or *escutcheon*,
- Writing the base web server and
- Writing the quarter and charge drawing algorithms.

1.4 *Report Structure*

Following this chapter, the report is broken up into 5 further chapters with the following content:

- **Chapter 2** presents the background surrounding the problem, with a beginner's guide to basic heraldry terms and the history surrounding heraldry, as well as other works related to this project;
- **Chapter 3** describes the overarching design decisions taken to guide the project in its implementation, along with its dependencies;
- **Chapter 4** provides an in-depth discussion of the iterative design and implementation process as I worked on solving the problem presented;
- **Chapter 5** evaluates the results, with regard to automated testing, other tools solving a similar, if not the same, problem, and shortcomings that the final implementation had; and
- **Chapter 6** concludes the work with further work that could be undertaken.

2

BACKGROUND

2.1 *Heraldry*

Many families, countries and organisations — primarily in Europe — have coats of arms. Coats of arms were initially used on shields on the battlefield to identify individual knights, but later came to be used as flags and banners for individuals and families of the upper classes at court. The Royal Coat of Arms of the United Kingdom, belonging to the British monarch, can be seen in Figure 2.1. If the reader wishes to learn more about heraldry and its history, I would recommend the heraldic works of Charles Boutell and John Brooke-Little.

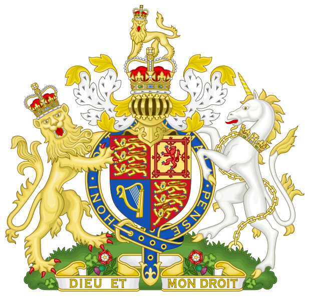


Figure 2.1: The Royal Coat of Arms of the United Kingdom. Source: `https://upload.wikimedia.org/wikipedia/commons/9/98/Royal_Coat_of_Arms_of_the_United_Kingdom.svg`

At the centre of a coat of arms is a shield known as an *escutcheon*. The language used to describe how the escutcheon is to be drawn is known as a *blazon*. Blazons have been used since the Norman conquest and have been refined to a regular language in the process,[1] although, as John Brooke-Little said, "many of the supposedly hard and fast rules laid down in heraldic manuals [including those by heralds] are often ignored."[2] This blatant disregard for the rules introduces difficulty in parsing the blazons as the language loses some of its regularity.

[1] Charles Boutell. *Heraldry, historical and popular*. Third edition. Accessed: January 2018. London, Bentley, 1864, pp. 8–9.

[2] J. P. Brooke-Little. *An Heraldic Alphabet*. New and revised edition. Accessed: January 2018. London, Robson Books, 1985, p. 52.

Blazons have 5 key attributes:

- The *field*, which is the background colour of the shield or quarter;
- *Ordinaries*, which are geometric shapes (as seen in Figure 2.4, bearing a golden slash, or *bend*);
- *Charges*, which are small emblems, such as fleur-de-lis and lions;
- *Variations*, which describe how the field or charge is patterned. Variations can indicate patterns such as chequered or coloured lines (as seen in Figure 2.2); and
- *Tinctures*, which are the colours and patterns for charges, ordinaries and fields.

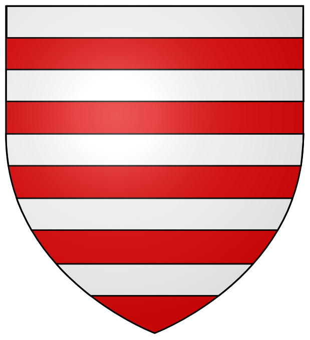


Figure 2.2: The shield of the town of Albert, France. *Barry of ten argent and gules*. Source: `https://upload.wikimedia.org/wikipedia/commons/e/ee/Blason_Albert.svg`

The tinctures are derived from Norman French and are divided into 3 groups, typically known as *metals*, *colours* and *furs*. In British heraldry, the colours are also derived from Norman French and so the names appear archaic. In heraldry, blue is *azure* and red is *gules* for instance. The metals are *or* and *argent*, for gold and silver respectively. Whilst the tinctures are linked to colours, the College of Arms does not specify which shade of that colour is required for the tinctures, leaving it to the artist to decide.[3] In this case, I have applied default CSS colours, using the English names, for instance, 'red' for *gules*.

[3] *FAQs: heraldry - College of Arms*. Accessed: January 2018. College of Arms. URL: http://www.college-of-arms.gov.uk/resources/faqs.

Blazons conventionally start with the *tincture* or *variation* of the field. After the description of the field, *ordinaries* and *charges* are named with their tinctures. An example of this is "*Purpure, a chief Gules*". This blazon describes an escutcheon with a field of *purpure* (purple), with a *Chief* ordinary (a bar across the top of the shield) of *gules* (red). This can be seen in Figure 2.3.

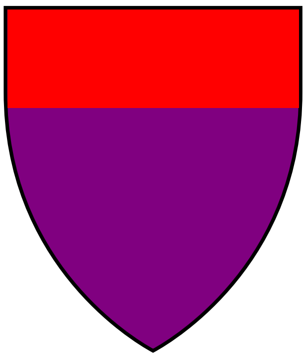


Figure 2.3: *Purpure, a chief Gules*, as drawn by the project web app.

A simple, but notable, blazon is that of the Scrope family. In the 14th century, the Baron Scrope brought a case action against Sir Robert Grosvenor when he noticed that they both had the same coat of arms. Many witnesses gave evidence in the case, including Geoffrey Chaucer.[4] The case was ultimately decided in Scrope's favour. The Scrope coat of arms has a blazon of *Azure, a bend Or*; a depiction of this can be seen in Figure 2.4.

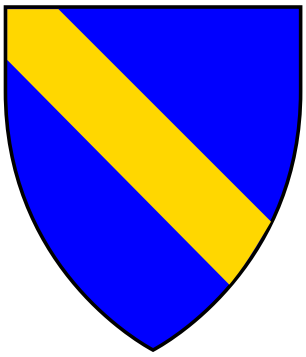


Figure 2.4: The Scrope escutcheon; *Azure, a bend Or*, as drawn by the project web app.

[4] Sir N. Harris Nicolas. *The Controversy between Sir Richard Scrope and Sir Robert Grosvenor in the Court of Chivalry*. Accessed: January 2018. London, Bentley, 1832, p. 404.

Whilst the Scrope arms are prominent in heraldry, they are simplistic and indicative of medieval arms. Coats of arms became more complex as they developed through the centuries, with instances of *quarterly* shields, *grand-quarterlies* — quarterlies within quarterlies — and *differenced* arms. *Differenced* arms involve adding an ordinary over an existing coat of arms. This was typically used to differentiate similar looking coats of arms, especially between father and sons. Common examples of differentiated shields are seen in duchies' coats of arms, particularly those which were given to Charles II's illegitimate children. Examples of more complex shields can be seen in Figure 2.6.

For a time, it was considered bad form to repeat a *tincture* in a blazon, and use a reference to the tincture's previous use. The Heraldic Society gives an example as such: "'*Azure on a fess argent three billets azure*' [would have been written as] '*Azure on a fess argent three billets of the first*'". The '*of the first*' refers to the field's tincture of azure. This blazon describes a blue shield, with a white bar horizontally across the middle with 3 white rectangles arranged along the bar. A rendering of this can be seen in Figure 2.5. The Heraldic Society advocates repeating tinctures to reduce ambiguity.[5]

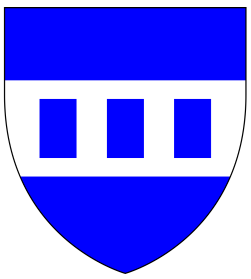
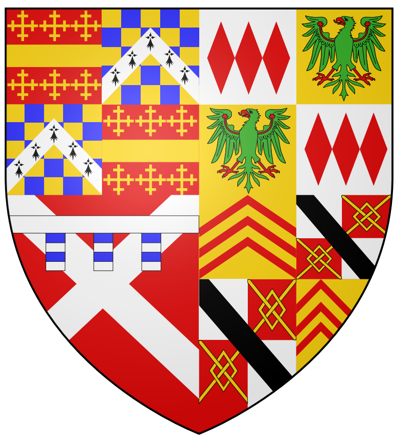
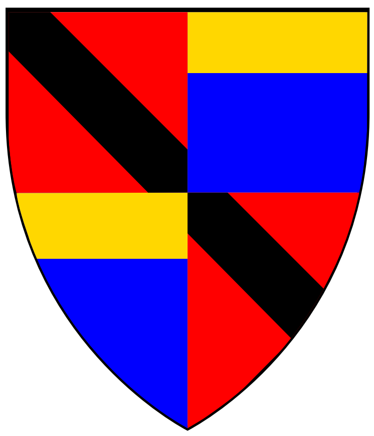


Figure 2.5: *Azure on a fess argent three billets azure* as rendered by *pyBlazon*. See Section 2.2.

[5] *Blazon in CoA | The Coat of Arms*. Accessed: January 2018. The Heraldic Society. URL: http://www.the-coat-of-arms.co.uk/blazon-in-coa/.



(a) Neville, 16th Earl of Warwick's coat of arms. An example of grand-quarterlies and differenced arms. Source: `https://upload.wikimedia.org/wikipedia/commons/d/d1/Neville_Warwick_Arms.svg`.



(b) A quarterly shield drawn by the web app. *Quarterly: 1st and 4th: Gules, a bend Sable; 2nd and 3rd: Azure, a chief Or.*

Figure 2.6: 2 examples of more complex coats of arms.

2.2 Related Works

Rendering blazons has been partially implemented before: Robert Billard wrote *Blazons!*[6] for Windows 3.1 and 95 and Mark Shoulson and Arnt Richard Johansen created *pyBlazon*[7] in 2008. *Blazon!* does not parse a blazon for automatic rendering, but provides an environment with predefined charges and ordinaries to build one's own escutcheons. *pyBlazon* solves this problem exceptionally well, but does not render quarterly shields. Thus, a major goal of this project is to support this. *pyBlazon*, as the name suggests, is implemented in Python[8] and hosted in a PHP webserver. My intention to implement the rendering in TypeScript[9] (see Section 3.1) would aid portability, as the code would be able to run on all modern browsers, whilst also being decoupled from the back-end, allowing pluggable implementations of the parsing.

[6] Robert Billard. *Blazons!* Accessed: March 2018. 1998. URL: `http://www.harnmaster.us/blazon.html`.

[7] Mark Shoulson and Arnt Richard Johansen. *pyBlazon: Blazonry to SVG Converter*. Accessed: March 2018. 2008. URL: `http://web.meson.org/pyBlazon/`.

[8] Python Software Foundation. *Python*. Accessed: April 2018. URL: `https://www.python.org/`.

[9] Microsoft Corporation. *TypeScript - Javascript That Scales*. Accessed: September 2017. URL: `https://www.typescriptlang.org/`.

Despite the existence of these tools, they seem rarely used. Many escutcheons drawn and uploaded to WikiMedia in Scalable Vector Graphics (SVG)[10] format appear to have been created in Inkscape.[11] Much work has been done in collecting and cataloguing blazons themselves (especially John Papworth as mentioned in Section 1.1) giving plenty of examples against which to test the finally completed application.

[10] Jon Ferraiolo, Fujisawa Jun and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. Accessed: October 2017. iuniverse, 2000.

[11] Inkscape Team. *Inkscape: A vector drawing tool*. Accessed: October 2017. URL: `http://www.inkscape.org`.

2.3 Summary

In this chapter, we covered basic heraldry, including core terminology, as well as works related to the project. Core heraldry terminology includes:

- *Escutcheon* — the shield in the coat of arms;
- *Field* — the background of the escutcheon;
- *Ordinaries* — geometric shapes on the escutcheon;
- *Charges* — small emblems, such as fleur-de-lis and lions; and
- *Tincture* — the colours and patterns for charges, ordinaries and fields.

All relevant heraldry terminology may be found in the Glossary on page 41.

3

DESIGN

3.1 Core Concepts

The two languages chosen for implementing this project were Python[1] and TypeScript.[2] Python was an optimal choice due to its good support for Natural Language Processing (NLP) through the Natural Language Tool Kit (NLTK).[3] TypeScript is a typed superset of JavaScript written by Microsoft that compiles to plain JavaScript. TypeScript was chosen as a more expressive alternative to programming in pure JavaScript, thanks to the addition of powerful features such as type assertions, access control and abstract classes. As a note to the reader, to maintain interoperability with JavaScript, TypeScript adds type definitions after the variable name, rather than before it, as in C. An example being that an `int` defined in C would be '`int limit`', but in TypeScript, this would be '`limit: number`' [4].

[1] Python Software Foundation, *Python*.

[2] Microsoft Corporation, *TypeScript - Javascript That Scales*.

[3] Steven Bird and Edward Loper. 'NLTK: the natural language toolkit'. In: *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics. 2004, p. 31.

[4] JavaScript/TypeScript uses a unified `number` type to handle both `floats` and `ints`.

The core design for this project centres around having a split stack; with a Python back-end parsing the blazon, serialising it into JSON[5] and passing it to the TypeScript front-end, which would then draw it onto the webpage. This allows for large amounts of flexibility, enabling the two halves of the project to be developed in tandem with the Separation of Concerns principle being adhered to throughout. It also allows for pluggable rendering implementations as the JSON for drawing payloads can be specified.

[5] JavaScript Object Notation (JSON) is a lightweight data-interchange format, consisting of key-value pairs, array data types and other serialisable data types (such as strings, numbers and booleans). JSON is derived from JavaScript's associative array-style data type, `Object`. An example of JSON can be seen in Figure 3.1.

The Python back-end receives a JSON payload from the webpage containing the blazon; it parses the blazon using a Context-Free Grammar (CFG) parser and identifies the most important parts of the blazon. It then serialises these back into a JSON response to be sent to the webpage for rendering. The specification stated that if the webpage was given a blazon of "Azure, a bend Or", it would return the JSON response seen in Figure 3.1.

```
{
  "field": "azure",
  "charges": [{
    "charge": "bend",
    "tincture": "or"
  }]
}
```

Figure 3.1: Expected response from the Python back-end, for a given blazon of "Azure, a bend Or".

The TypeScript front-end receives this response, applies the `azure` CSS class to the field element, then draws a bend onto the field with an `or` CSS class. The rendering of this response can be seen in Figure 2.4. The front-end does not use a framework and is written in plain TypeScript.

Escutcheons are drawn using the SVG format for portability across browsers as well as the inherent scalability of SVG images. This allows drawn escutcheons to be embedded elsewhere with ease, either directly or through rendering the SVGs as other image formats via programs like Inkscape.

For version control, `git`[6] was used and all code was hosted on GitHub.

[6] Linus Torvalds and Junio Hamano. *Git: Fast Version Control System.* URL: `https://git-scm.com`.

3.2 *External Dependencies*

The front-end depends on a pair of libraries for SVG rendering and Document Object Model (DOM) manipulation: the selection library of D3.js[7] and jQuery.[8] D3.js and jQuery both provide many powerful functions for SVG and DOM manipulation. The World Wide Web Consortium (W3C) DOM standard defines the DOM as "a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document." The Mozilla Developer Network adds to this, saying the DOM "represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects.".[9]

[7] Mike Bostock. *D3.js - Data-Driven Documents.* Accessed: September 2017. URL: `https://d3js.org`.

[8] The jQuery Foundation. *jQuery.* Accessed: October 2017. URL: `https://jquery.com/`.

[9] MDN Contributors. *Document Object Model (DOM) - Web APIs | MDN.* Accessed: March 2018. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`.

For further assets, Sass[10] (a CSS extension language) was used as a CSS pre-processor in combination with Bootswatch[11] (a predefined CSS theme) for the base styling. Webpack[12] compiled TypeScript down to JavaScript — minifying and uglifying it in the process — and concatenated all source files and their dependencies into one main JavaScript file. *Minification* of JavaScript assets involves stripping out all unnecessary whitespace and tokens. *Uglification* transforms the JavaScript code by renaming all variables and functions into short, obfuscated names to reduce the footprint of the assets. These two techniques can decrease loading times of web apps as the browser has to download smaller asset resources than the original,

[10] Hampton Catlin, Natalie Weizenbaum and Chris Eppstein. *Sass: Syntactically Awesome Stylesheets.* Accessed: September 2017. URL: `https://sass-lang.com/`.

[11] Thomas Park. *Bootswatch: Flatly.* Accessed: September 2017. URL: `https://bootswatch.com/flatly/`.

[12] JS Foundation. *webpack.* Accessed: October 2017. URL: `https://webpack.js.org`.

raw source code.

3.2.1 *Development Dependencies*

To maintain code quality, TSLint[13] (a TypeScript code linter) was set up to automatically run as part of the Travis Continuous Integration (CI)[14] service, causing a build to fail if the linter detected a style violation. For unit testing, Jest[15] (with ts-jest[16] for TypeScript support) was selected especially for its powerful mocking and expectation matcher functionality. Automated documentation generation was provided by TypeDoc.[17]

[13] Palantir Technologies. *TSLint*. Accessed: September 2017. URL: `https://palantir.github.io/tslint/`.

[14] Travis CI GmbH. *Travis CI - Test and Deploy with Confidence*. Accessed: November 2017. URL: `https://travis-ci.com`.

[15] Facebook, Inc. *Jest - Delightful JavaScript Testing*. Accessed: February 2018. URL: `https://facebook.github.io/jest/`.

[16] Kulshekhar Kabra. *ts-jest*. Accessed: February 2018. URL: `https://github.com/kulshekhar/ts-jest`.

[17] TypeDoc Contributors. *TypeDoc - Documentation generator for TypeScript projects*. Accessed: February 2018. URL: `http://typedoc.org`.

3.3 *Iterative Design*

An iterative design process was used for writing the code for this project. Iterative design is a cyclical process of designing, prototyping and evaluating. In other words, one designs and prototypes a new feature before evaluating the final feature design. If the design is acceptable, the new feature is implemented, otherwise the cycle restarts. An iterative design process enables one to address different functionality as separate tasks, building on top of one another. It also allows for extensive refactors of a project whilst staying within one cycle.

4

IMPLEMENTATION

4.1 *Basic Charge Rendering*

4.1.1 *Design*

The first design iteration had a specific focus on basic charge drawing. The second iteration for adding quarterly rendering, would rely on the results from this iteration.

I decided to have all drawing logic defined in the client, existing in a single module with minimum dependencies. The shield outline was rendered on the page on load as part of the HTML template. This supported a reliable entry point for the drawing logic as selecting the shield element was straightforward. Having selected the element, the web app could begin appending other SVG elements to it and enabled layering to be achieved. This is due to SVG ordering layers based on the order of elements in the document. This was used to great effect when drawing quarters (see Section 4.2.1).

The front-end was designed around functional paradigms; breaking up major functionality into functions that would deal with smaller, encapsulated functionality, such as adding extra layers to the HTML template or clearing the shield when drawing a new blazon. This allowed for a stable API, as the single point of access function would not be renamed, but all other functions may be changed and updated separately. As described in Section 3.1, the front-end first accessed the `field` value in the parsed JSON response, applied the value as the CSS class for the shield and then moved onto the charges. It iterated over the `charges` array in the JSON response, drawing each charge onto the shield and applying the tincture as the CSS class. Due to SVG layering, as mentioned earlier, if there were multiple charges specified in the response, all would be drawn according to the array ordering.

4.1.2 *Implementation*

As described in Section 4.1.1, the initial approach was to have all methods in the core `index.ts` file that would be compiled and loaded in the browser. This meant a smaller footprint when the code was bundled by Webpack and easier maintenance as all relevant functions were next to one another, following the Step-Down Rule[1,2].

[1] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Accessed: February 2018. Pearson Education, 2009, p. 37.

[2] The Step-Down Rule dictates that if function `A()` calls function `B()` and `C()` in its function body, functions `B()` and `C()` should be defined immediately after function `A()`.

`drawShield(blazon)`

The web app had a single entry point of `drawShield(blazon)`, where `blazon` was the whole JSON response returned from the `/_parse` endpoint on the web server (see Figure 3.1 for an example JSON response). Initially this presented a problem as TypeScript didn't handle the unstructured parsed data well due to it being a JavaScript `Object`[3] instance. Attempting to access members of this object (such as `field`) caused TypeScript to produce an error stating that the contents might be undefined and thus return a `null` object. To fix this, I designed `interfaces` with the expected fields in the payload; one for the whole object, `IBlazon`, and one for the charges array contained within, `ICharge`. Similarly, to avoid problems with string matching, I defined a pair of `enums` to represent the supported tinctures and charges, `ETincture` and `ECharge` respectively. As discussed in Section 4.2, I later added another `enum` for quarters. All `interfaces` and `enums` can be found in Appendix A. Having fixed this data problem, `drawShield(blazon)` was now able to access members of the blazon object safely.

[3] In JavaScript, an `Object` works as both an associative array and a basis for classes and inheritance through its `prototype` field.

`clearShield()`

To avoid the problem of overlapping charges, I had to write a `clearShield()` method that would iterate over all the `path` nodes in the SVG document, and delete them. This, however, promptly deleted the shield outline, so I had to add a check to prevent deleting `path` nodes with a `#shield` id, instead only removing the CSS class. Having cleared the shield of any possible obstructions, the `drawShield` method would then assign the contents of the `field` value as the CSS class and iterate over the `charges` array, passing each `charge` to `drawCharge(charge: ICharge)`.

`drawCharge()` and `ChargeShapes`

When a charge node is created in `drawCharge`, it is assigned an id. This id was formed from the name of the charge, followed by a random number in the range 1–512 inclusive. This was later changed to rely on a cryptographically secure, random number generator, generating unsigned 16-bit integers.

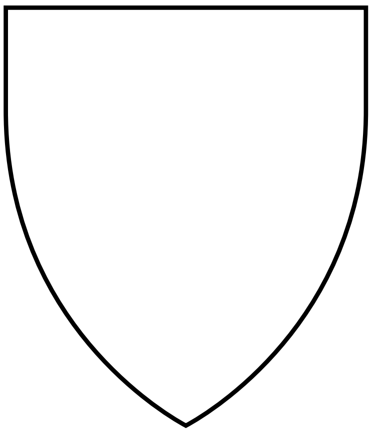
To draw shapes in SVG a `<path>` node requires a '`d`' attribute which contains the commands for drawing that shape. To generate all these attributes, I drew all the charge shapes in Inkscape and extracted the '`d`' attributes from the generated SVGs. At first, I put a `Map`[4] of the charges and their paths in the global scope, available for all functions to access. This worked for charges that were produced using a `path` node with a '`d`' attribute, but introduced a problem when using the chief charge (a chief charge being displayed in Figure 2.3). The chief charge was drawn using a `<rect>`[5] node which required '`x`' and '`y`' co-ordinates to specify a starting point and `height` and `width` attributes to describe the size of the rectangle. To address this, I wrote a `ChargeShapes` class to encapsulate the charges and their attributes. This class provided one public member, `chargePaths` which was of type `Map<string, Map<string, string>>`. Representing this as a `Map` allowed `drawCharge` to first check if the app knew how to draw the charge by checking whether `ChargeShapes.chargePaths` contained the charge as a key. If the charge had an entry, then `drawCharge` would iterate over the attribute `Map` and apply them to the `<path>` or `<rect>` node, before finally applying the CSS class.

[4] A `Map` here is a TypeScript/JavaScript data type, also known as a `HashMap` or an associative array.

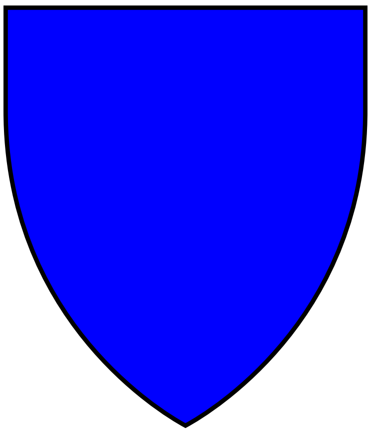
[5] A `<rect>` node in SVG is used for drawing rectangular shapes.

A visual representation of `drawCharge`'s execution may be seen in Figure 4.1.

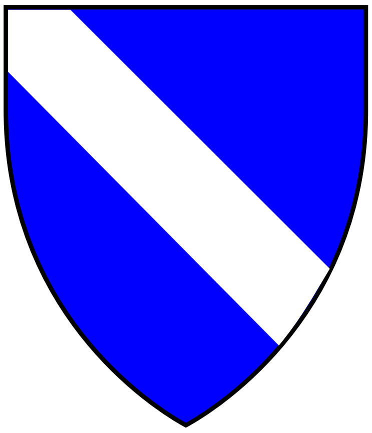
The final part of `drawCharge` applied a transform to the `path` if the JSON response included a boolean flag, `sinister`, to indicate that the bend charge should be flipped. This writes to an attribute `transform` which applies a `matrix` transformation that flips the charge followed by a `translate` transformation to move it into place. An example of a sinister bend can be seen in Figure 4.2.



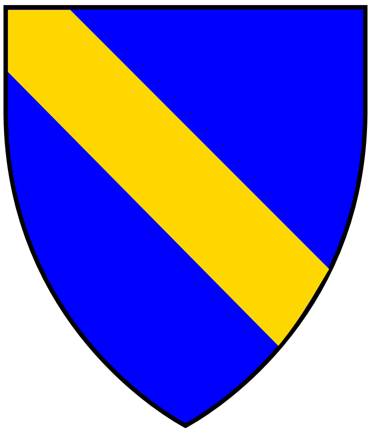
(a) Initial escutcheon outline on the webpage.



(b) First stage: applying the `field` CSS class.



(c) Second stage: drawing the charge onto shield.



(d) Final stage: applying the `tincture` CSS class on charge.

Figure 4.1: The process of `drawCharge()` rendering a Scrope escutcheon.

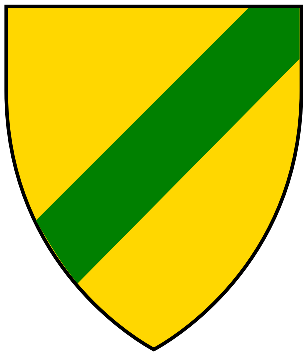


Figure 4.2: *Or, a bend sinister Vert.*

4.1.3 *Evaluation*

Whilst this simple implementation worked well for drawing basic escutcheons, like Figure 2.3 and Figure 4.2, it wasn't able to draw more complex shields like those seen in Figure 2.6 and thus was not meeting the project aims.

4.2 *Adding Quarterly Rendering*

4.2.1 *Design*

Design Principles

Whilst prototyping for adding functionality to render quarterly shields, I found that it was going to be impossible to maintain the initial, atomic design laid out in Section 4.1.1 whilst keeping the code clean and readable. This started the second design iteration of the project. In this iteration, I designed a new, modular system that leaned more heavily into Object-Oriented paradigms than functional ones. This new design was written to enforce delegation and decoupling, whilst following the Single Responsibility[6] and Open/Closed principles.[7]

[6] Robert C Martin. 'Design Principles and Design Patterns'. In: *Object Mentor* 1.34 (2000), p. 597.

[7] Martin, 'Design Principles and Design Patterns'.

The Single Responsibility principle dictates that a class should have one, and only one, reason to change. If another responsibility is introduced, it should be given its own class. This principle works well with decoupling as responsibilities can be changed in individual classes without affecting other classes that depend upon it. The Open/Closed principle dictates that a class should be open to extension but closed to modification. A typical example of this would be using abstract classes; the interface specified by the abstract class is closed to modification, but the child classes may extend the functionality in their implementation.

To comply with these principles, all major sections of functionality, including blazon payload parsing, charge and quarter rendering, were encapsulated in their own classes with clear names and well-defined, shared APIs.

Top-Level Design

The new design had a top-level class of `Blazon` which had a single public function, `draw()`. The `draw` method, as before, would clear the shield and then delegate drawing responsibility to specialised `charge` and `quarter` renderers by calling their `draw()` methods. This `Blazon` class became the new entry-point, being instantiated and called in a `main()` function in `index.ts`. In a similar fashion to the top-level `Blazon` class, a `Quarter` class was defined to enable proper delegation for rendering both the quarter *and* the charges contained within the quarter. The new `ChargeRenderer` class would contain most major logic for drawing, as well as `id` generation. `QuarterRenderer` extended `ChargeRenderer` to add quarter-specific logic, while also being able to call up to it to draw the contained charges.

For a quarterly blazon, the payload would have the `field` value set to "`quarterly`" and rather then having a field `charges: ICharge[]`, it would contain a field `quarters: IBlazon[]`. This worked as quarters are treated as their own small escutcheons in heraldry and are described as such. An example of a quarterly blazon would be *Quarterly: 1st and 4th: Gules, a bend Sable; 2nd and 3rd: Azure, a chief Or*, as seen in Figure 2.6. The expected JSON response for this blazon can be seen in Figure 4.3. It was then possible to change functionality of the app depending on the contents of the `field` key. To account for the new quarters that needed to be drawn, a new `enum EQuarter` was designed, with the members being the positions of the SVG elements and the `paths` in the `ChargeShapes.chargePaths Map`.

Applying SVG Properties

In this refactor, I also made use of more SVG properties: clip paths and the `<g>` element. Clip paths allowed defining a path that cropped the element it was defined on. This was particularly useful for drawing charges inside of quarters as the edges outside of the quarter would be cropped out by the clip path defined for that quarter. The `<g>` element is a grouping element for SVG; it applies all transformations defined on it to all of its child elements and any of its attributes are also inherited. I used the `<g>` elements for both grouping together charges and quarters, and explicitly named layers for the shield outline and the charges within.

```
{
  "field": "quarterly",
  "quarters": [
    {
      "field": "gules",
      "charges": [
        {
          "charge": "bend",
          "tincture": "sable"
        }
      ]
    },
    {
      "field": "azure",
      "charges": [
        {
          "charge": "chief",
          "tincture": "or"
        }
      ]
    },
    {
      "field": "azure",
      "charges": [
        {
          "charge": "chief",
          "tincture": "or"
        }
      ]
    },
    {
      "field": "gules",
      "charges": [
        {
          "charge": "bend",
          "tincture": "sable"
        }
      ]
    }
  ]
}
```

Figure 4.3: Expected JSON response from the Python back-end for a given blazon of *Quarterly: 1st and 4th: Gules, a bend Sable; 2nd and 3rd: Azure, a chief Or*.

4.2.2 Implementation

Blazon

Blazon took two arguments in its constructor: svg and data. svg was a D3.js[8] data type that contained a selector for the <svg> element in the HTML document, for appending elements to. This data type is henceforth referred to as d3.Selection. The data argument took the full JSON response returned by the parser. The constructor would then turn the data Object into a Map object for more reliable access. To get more fine-grained access to the SVG document, the constructor populated extra fields by selecting the whole shield element and the <g id="charge_layers"> element, these being defined within the svg selection. The chargesLayer field is required for telling the renderer classes where to draw their shapes. Depending on the contents of the JSON response, the constructor would then instantiate either new ChargeRenderer objects, or 4 new Quarter objects.

[8] Bostock, *D3.js - Data-Driven Documents.*

clearShield was extracted from index.ts into Blazon, but with changes to account for the many <path> nodes as well as <clipPath> elements. The method iterated over child elements of the <g id="charge_layers">, which were <g>, <path> and <rect> nodes, and <clipPaths> with an id beginning with "quarterly_", deleting them in the process. It also stripped the shield node of its CSS class.

A UML diagram for Blazon may be seen in Figure B.1 in Appendix B.1.

Quarter

The new Quarter class relied on the order of the quarters array in the response to determine which quarter it was representing; with numbering starting at 0 in the top left quarter, going left to right, ending at 3 in the bottom right. The ordering of the quarters array may be seen in Figure 4.3. The constructor required the array index as its first argument, with field: ETincture, charges: ICharge[], svg and chargesLayer as its other arguments. With these parameters, the constructor selected the quarter specified by the index, instantiated ChargeRenderer objects — borrowing a method from Blazon — and a QuarterRenderer object. The draw() method of Quarter first called the draw() method on the QuarterRenderer object to draw the quarter path onto the shield, then called addClipPathDefinition(svg) (see ChargeRenderer) to add a new clip path to the SVG document. Having used QuarterRenderer to draw a new quarter path, it selected it, assigned it to a locally scoped constant, quarterLayer, and iterated over the charges. Before calling the draw method on the charge object, it called the updateChargesLayer on the charge object, passing it the `quarterLayer` to instruct the charge to render in the quarter.

A UML diagram for `Quarter` may be seen in Figure B.2 in Appendix B.1.

`ChargeRenderer`

The `constructor` for `ChargeRenderer` expected a `chargesLayer`, a `tincture: ETincture`, a `charge: ECharge | EQuarter` and a `sinister: boolean` argument. The `tincture` argument was used by both the `ChargeRenderer` itself and the `QuarterRenderer`, for the colour of the charge and field respectively. The union type signature of `charge` allowed it to draw both charges and quarters, as appropriate. For the drawing logic itself, I extracted the `drawCharge` function from the `index.ts` file into the `ChargeRenderer` `draw()` method. However, in trying to keep extensibility from `ChargeRenderer` to `QuarterRenderer`, the logic was then extracted from the `draw` method into its own protected `drawCharge(currentCharge: d3.Selection, chargeLayer?: d3.Selection)` method. This freed up the `draw` method to handle setting up the SVG document for the quarters and charges to be drawn into, as well as applying clip paths.

Whilst implementing charge rendering within quarters, I found that I needed to change the layer that charges were being drawn in to. I decided to add a new method to allow updating the `chargesLayer` after a `ChargeRenderer` object had been instantiated.

The `getRandomInt` method was also extracted from the `index.ts` file into `ChargeRenderer` for `id` generation.

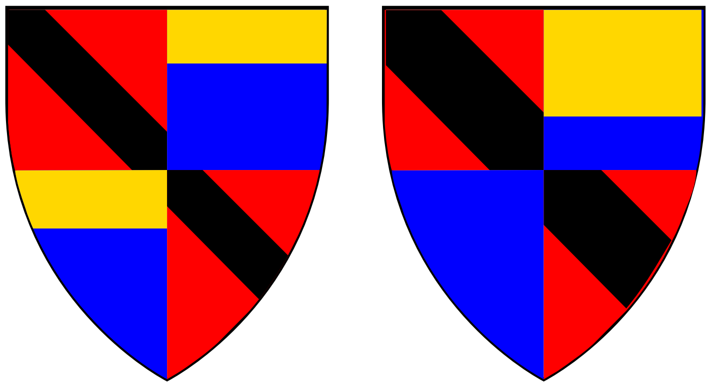
`QuarterRenderer`

`QuarterRenderer` extended `ChargeRenderer`, with few changes to what it inherited. Rather than generating a random `id` for the quarter, it used the `charge: ECharge | EQuarter` parameter (that would be a member of `EQuarter` in this case). A new method `addClipPathDefinition(svg)` was also added. In SVG documents, clip paths are defined using `<clipPath>` elements inside a `<defs>` element at the top of the document. The `addClipPathDefinition` method selected the `<defs>` element and appended a `<clipPath>` element, before updating the `chargesLayer` and calling the inherited `drawCharge` method.

A UML diagram for `ChargeRenderer` and `QuarterRenderer` may be seen in Figure B.3 in Appendix B.1.

4.2.3 *Evaluation*

This design iteration had brought quarterly rendering a lot closer to fruition, as seen in Figure 4.4, however proper scaling and translation of charges was yet to be implemented. The project aims were yet to be realised. When adding the functionality to apply quarter-specific transformations, I had difficulties maintaining clean, extensible code. To both satisfy the project aims and the code standards, I had to redesign how `ChargeShapes` and both `Renderer` classes worked.



(a) The expected output for the blazon *Quarterly: 1st and 4th: Gules, a bend Sable; 2nd and 3rd: Azure, a chief Or*.

(b) The second iteration output of this blazon.

Figure 4.4: Second iteration progress.

4.3 *Refactoring Charge Rendering*

4.3.1 *Design*

`Renderer` *Hierarchy*

This new design made use of the *Liskov Substitution* and Open/Closed principles.[9] The Liskov Substitution principle dictates that derived classes must be substitutable for their base class. Abstract classes are a good application for this principle as they provide a reliable interface and force inheriting classes to implement the abstracted methods defined within. As a reminder, the Open/Closed principle dictates that classes should be closed to modification but open to extension.

[9] Martin, 'Design Principles and Design Patterns'.

Applying these principles, I redesigned the rendering structure to have common methods and attributes belonging to both `ChargeRenderer` and `QuarterRenderer` in a new abstract parent class, `Renderer`. This refined and enforced the previously specified API of render classes having a main entry-point of `draw()`. Using an abstract class gave assurances to classes that used its implementations that there would always be a `draw()` function. It also gave child classes the shared functions that both required, whilst allowing them to specialise in their individual use cases. This enabled previously ambiguous attributes to be renamed more specifically for their particular `Renderer` implementation. An example is where `ChargeRenderer` previously had `charge: ECharge | EQuarter`, it now had `charge: ECharge`.

The planned execution for this design may be seen in Figure 4.5.

`ChargeShapes` and `QuarterShapes` Abstraction

In a similar fashion to the `Renderer` redesign, I devised a new design for the attributes and transforms for `ChargeShapes`. Rather than having all paths defined in a `Map` on `ChargeShapes`, a new abstract class, `AShape`, would be defined with a `dimensions` `Map` and a `transforms(transform: string): string` method. Each charge then extended `AShape`, implemented the two abstract properties and was imported by the `ChargeShapes` class. The `ChargeShapes` class was redesigned to have a boolean function, `hasChargePath(charge: string)`, to check if it knew the given charge. `ChargeShapes` also had a static function, `chargeShapes(charge: ECharge)`, which returned an instance of `AShape`. This design makes use of the *Dependency Inversion* principle,[10] which says that a class should depend on abstractions, rather than concretions. In this case, when renderers call out to `ChargeShapes` for the relevant `chargeShapes` object, they are returned an `AShape` object. This increases reusability as well as extensibility of charge rendering, as one just needs to create a new `AShape` class, import it into the `ChargeShapes` class and add it to the relevant methods.

[10] Martin, 'Design Principles and Design Patterns'.

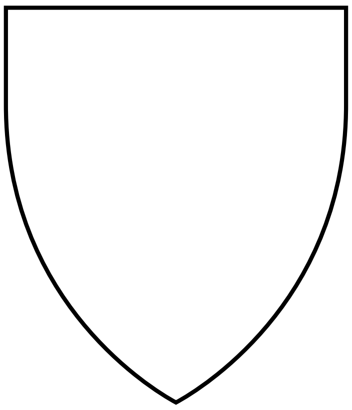
Quarter shapes were also extracted into their own hierarchy, mirroring that of the `AShape` hierarchy. `AQuarterShape` only exposed a `dimensions` `Map` however.

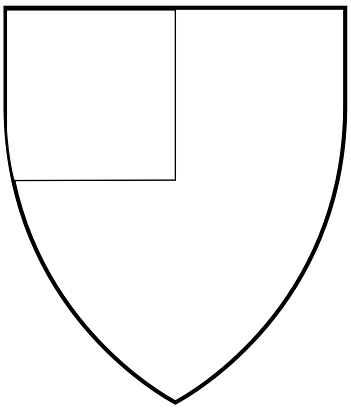
4.3.2 Implementation

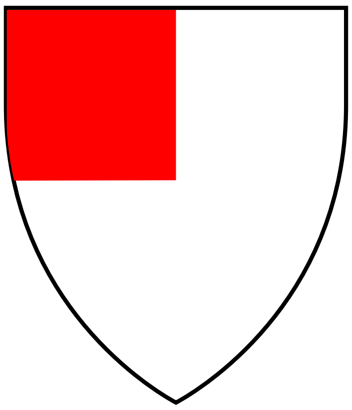
`Renderer`

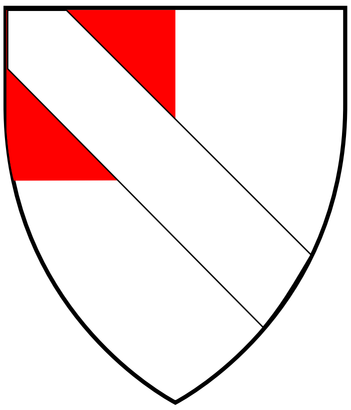
The new `Renderer` class contains 3 protected, inherited attributes, `chargeId: string`, `tincture: ETincture` and `parentChargesLayer: d3.Selection`. Whilst the class has 3 attributes, only `tincture` and `parentChargesLayer` are assigned in the constructor; inherited classes are given responsibility of generating their own `ids`. As mentioned in Design, `Renderer` defines an abstract method of `draw(): void` to be implemented in inherited classes. The `updateChargesLayer` method was extracted from `ChargeRenderer` into the parent class to be available for both child classes to inherit and use. `getRandomInt` was similarly extracted.

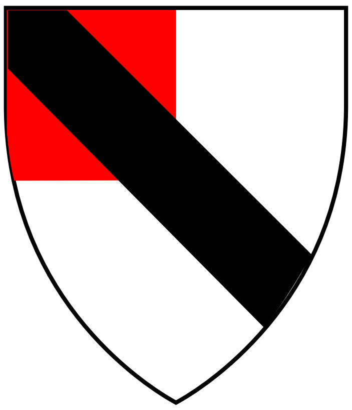
Figure 4.5: The process of the `Quarter` class rendering a quarter with a bend in it.

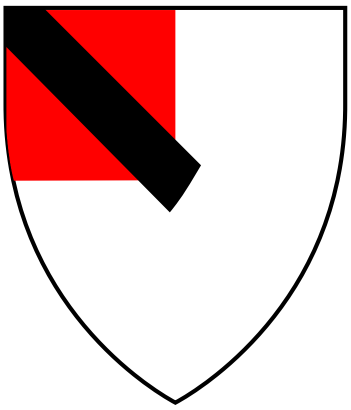

(a) Initial escutcheon outline on the webpage.

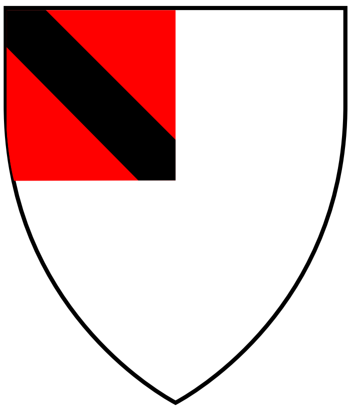

(b) Drawing on the quarter shape.


(c) Applying the `field` CSS class.


(d) Drawing on the charge shape.


(e) Applying the `tincture` CSS class.


(f) Applying the scaling transformation to the charge.


(g) Applying the clip path to the charge.

ChargeRenderer

Now inheriting from `Renderer`, `ChargeRenderer` has just two attributes defined on it: `charge: ECharge` and `sinister: boolean`. The `draw` method now has an optional argument of `quarter: EQuarter` which corresponds to the `id` of the `<clipPath>` that contains this charge. As before, the method performs environment setup in the SVG document, preparing for the new charge. It then calls out to `drawCharge`, however, after performing all this, `draw` now also calls `applyTransforms`. `applyTransforms` takes in the `currentCharge` element and an optional `quarter` argument, builds a `transform` string based on the parameters, fetches the appropriate transformation from `ChargeShapes` and applies it to the `currentCharge`.

QuarterRenderer

As with `ChargeRenderer`, `QuarterRenderer` now has only one attribute defined upon it: `quarter: EQuarter`. Whereas before, the `draw` method called out to `ChargeRenderer` to draw a quarter onto the shield, it now calls its own `drawQuarter` method. `drawQuarter` adopts much the same functionality as `drawCharge` previously held, but exclusively works on `QuarterShapes`. `addClipPathDefinition` remained unchanged.

A UML diagram for `Renderer`, `ChargeRenderer` and `QuarterRenderer` may be found in Figure B.4 in Appendix B.2.

AShape and AQuarterShape

As mentioned in Section 4.3.1, `AShape` defines a pair of abstractions: the `dimensions` `Map` and the `transforms` method. The `dimensions` `Map` is a simple `Map<string, string>` containing relevant entries previously defined in the `ChargeShapes` class. `transforms` takes a transform string as an argument. Implementations use a `switch` statement to match against it to specify the necessary transformation.

`AQuarterShape`, not needing any transformations, simply defines an abstract `dimensions` `Map` that parallels that of `AShape`.

A UML diagram for `AShape` may be found in Figure B.6 in Appendix B.2.

ChargeShapes and QuarterShapes

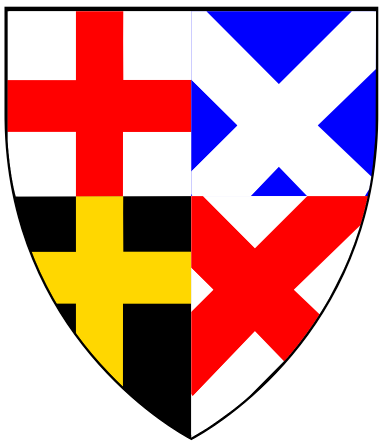
Having extracted all the previous dimensions into their own classes, `ChargeShapes` now provides a single point-of-access for all charges. It has a private, static array containing the names of each imported charge. To check whether the charge can be drawn, it exposes a boolean function, `hasChargePath(charge: string)`, that scans over the array to check whether the argument given to it is contained within. A new function `chargeShapes(charge: ECharge)` was defined to replace the `chargePaths Map`, returning a new object for the relevant `AShape`. When the `ChargeRenderer` calls this, it is able to chain methods together without concerns about which `AShape` it has been given.

Again, to mirror the new `ChargeShapes` implementation, `QuarterShapes` is defined with a sole `quarterShapes(quarter: EQuarter)` method which returns a new object for the relevant `AQuarterShape`. The implementations for both `QuarterShapes` and `AQuarterShape` were unnecessarily verbose, but were done purposefully to be keep the code open-ended and extensible for future functionality.

A UML diagram for `ChargeShapes` may be found in Figure B.5 in Appendix B.2.

4.3.3 *Evaluation*

The project aim that I was pursuing is now realised, with quarterly shields being reliably drawn. Some examples, as drawn by the web app, can be seen in Figure 4.6.


(a) A shield with the escutcheons of the 4 patron saints of the United Kingdom.

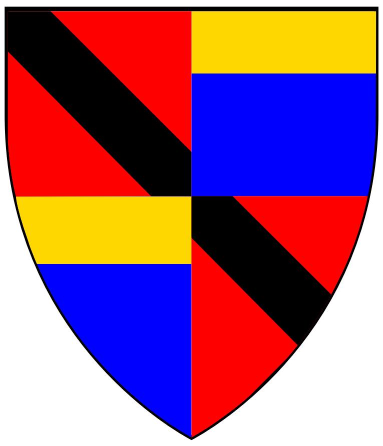

(b) *Quarterly: 1st and 4th: Gules, a bend Sable; 2nd and 3rd: Azure, a chief Or.*

Figure 4.6: Some quarterly examples drawn by the web app.

5 EVALUATION

5.1 Automated Testing

It was challenging to set up integration tests to ensure that the web app was calling the libraries with correct parameters. However, as mentioned in Section 3.2, I used Jest[1] to write unit tests for some modules, specifically to ensure implementations of AShape were returning the correct data depending on their inputs.

[1] Facebook, Inc., *Jest - Delightful JavaScript Testing*.

5.2 Comparison With Existing Solutions

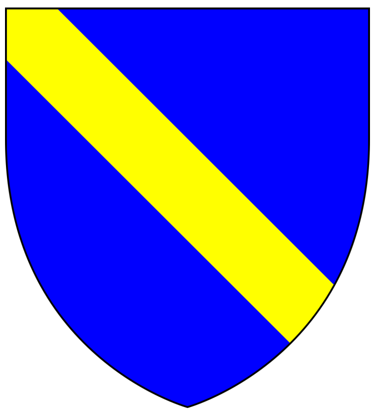
The other solution to this problem, as mentioned in Section 2.2, was pyBlazon.[2] A comparison of both pyBlazon and this project rendering the blazon for the Scrope escutcheon can be seen in figure 5.1. As noted in Section 2.1, there are no shades specified for a tincture, hence the difference in colours of pyBlazon's bend and this project's, the former being yellow and the latter gold.

[2] Shoulson and Johansen, *pyBlazon: Blazonry to SVG Converter*.

The web app for this project is about 3 times faster than pyBlazon in rendering the Scrope escutcheon. When measuring both sites using Firefox's performance tools on my laptop, the project web app took around **400ms** to render the blazon, whilst pyBlazon took around **12,000ms**. Whilst the project is considerably faster than pyBlazon, it should be noted that the project had pre-defined JSON to return and so didn't have to parse the blazon first.

When pyBlazon is given the blazon for the right-hand escutcheon in Figure 4.6, the website returns an HTTP `403 Forbidden` error, being unable to handle it. However, pyBlazon does not support quarterly escutcheons.

A restriction of my design was using just the British style of escutcheons, known as the heater style. Whilst this produces accurate results for British escutcheons, it produces incorrect renderings for other European escutcheons. Ignoring the laurel branches, the na-


(a) pyBlazon

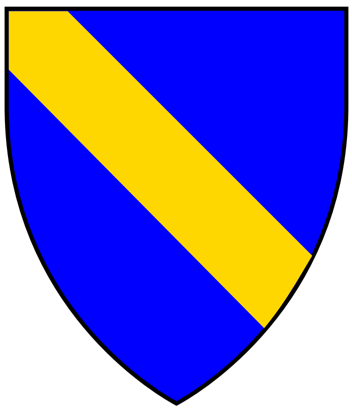
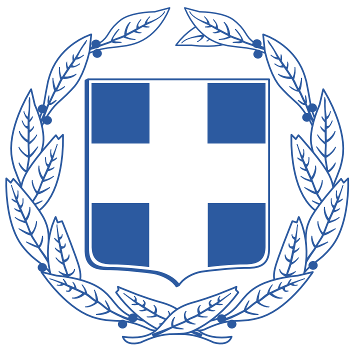

(b) This project

Figure 5.1: A comparison of the Scrope escutcheon (*Azure, a bend Or*) being rendered by both this project and pyBlazon.

tional escutcheon of Greece uses a more modern shield shape that the project does not support. As seen in Figure 5.2, the ordinary is correct and, as mentioned previously, the tincture is correct, shade and shield notwithstanding.


(a) Source: `https://upload.wikimedia.org/wikipedia/commons/7/7c/Coat_of_arms_of_Greece.svg`

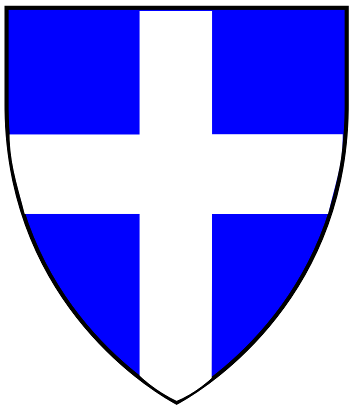

(b) This project

Figure 5.2: A comparison of the national escutcheon of Greece (*Azure, a cross Argent.*)

5.3 Limitations

Due to time constraints, we were not able to integrate the front- and back-ends, with the server configured to return mocked out responses for well-formed blazons.

In some quarters, the charges didn't fully fit, resulting in off-centred shapes to hide the visible field. An example of this can be seen in Figure 5.3, where the bottom right saltire is not aligned with the saltire above it. Similarly, the saltire in the top right corner should sit much higher within the quarter. This was due to drawing the saltire to match the dimensions of the shield outline, but not accounting for the differing dimensions of redrawing this within a quarter. A fix for this particular issue would be to draw the shapes much larger than their intended bounds and rely on SVG clip paths to keep the shape within the outline.

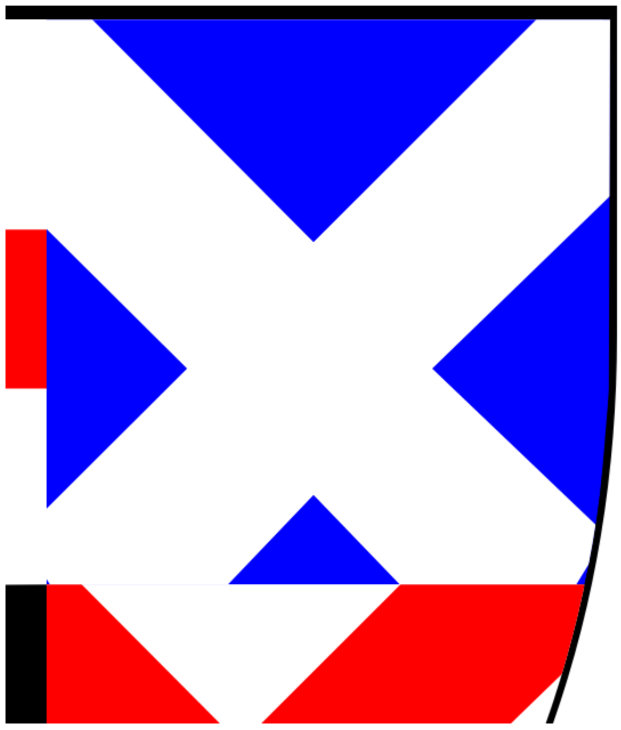


Figure 5.3: An enlarged image demonstrating the saltire not fulfilling the quarter in its bottom left and right corners.

The project is currently only designed to accept correctly-formed data and does not handle ill-formed data properly, short of throwing an exception. Ideally, it should be able to reject and recover from ill-formed responses received from the server, either through checking the validity of the data before working on it or through ignoring data that cannot be processed. More feedback to the user should also be given, should they enter a blazon that is unable to be parsed.

6 CONCLUSION

6.1 Overview

I achieved the project aim, designing and implementing a web app to render the blazon by drawing charges and ordinaries. Several blazons used in this report have been rendered by the web app, including Figures 2.3, 2.4 and 4.6.

I applied the good software engineering practices of delegating and decoupling, as well as the four principles laid out by Robert C Martin in Design Principles and Design Patterns:[1] *Single Responsibility*, *Open/Closed*, *Liskov Substitution* and *Dependency Inversion*. As a reminder these principles are defined as follows:

[1] Martin, 'Design Principles and Design Patterns'.

- *The Single Responsibility principle* — a class should have one, and only one, reason to change;
- *The Open/Closed principle* — you should be able to extend a class's behaviour, without modifying it;
- *The Liskov Substitution principle* — derived classes must be substitutable for their base classes, and
- *The Dependency Inversion principle* — depend on abstractions, not on concretions.

The Single Responsibility principle was used in all classes: splitting up any classes that started to develop overlapping functionality. The Open/Closed principle was applied in the `Renderer` hierarchy to ensure shared functions were defined in the parent class, whilst the child classes would both adhere to the same API for the rest of the web app to use. The Liskov Substitution principle applied to the `AShape` and `AQuarterShape` design, allowing for easy extension in the future should more shapes need to be drawn. The Dependency Inversion principle is applied to particular effect in the `ChargeShapes` method for chaining methods together without concerns of the underlying implementation.

6.2 *Further Work*

Initial further work would be to implement the fixes I have outlined in Section 5.3.

Other extra changes would be to add more shapes to `ChargeShapes`, such that the web app would be able to draw more blazons. Given the current implementation, there is potential to implement variations-based blazons through use of SVG clip paths. Similarly, adding support for drawing of charges such as lions and fleur-de-lis might wished to be added.

BIBLIOGRAPHY

Billard, Robert. *Blazons!* Accessed: March 2018. 1998. URL: `http://www.harnmaster.us/blazon.html`.

Bird, Steven and Edward Loper. 'NLTK: the natural language toolkit'. In: *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics. 2004, p. 31.

Blazon in CoA | The Coat of Arms. Accessed: January 2018. The Heraldic Society. URL: `http://www.the-coat-of-arms.co.uk/blazon-in-coa/`.

Bostock, Mike. *D3.js - Data-Driven Documents*. Accessed: September 2017. URL: `https://d3js.org`.

Boutell, Charles. *Heraldry, historical and popular*. Third edition. Accessed: January 2018. London, Bentley, 1864, pp. 8–9.

Brooke-Little, J. P. *An Heraldic Alphabet*. New and revised edition. Accessed: January 2018. London, Robson Books, 1985, p. 52.

Catlin, Hampton, Natalie Weizenbaum and Chris Eppstein. *Sass: Syntactically Awesome Stylesheets*. Accessed: September 2017. URL: `https://sass-lang.com/`.

Collins, S. M. 'Papworth and his Ordinary'. In: *The Antiquaries Journal* 22.1 (1942). Accessed: January 2018, pp. 6–7. DOI: `10.1017/S0003581500003668`.

Facebook, Inc. *Jest - Delightful JavaScript Testing*. Accessed: February 2018. URL: `https://facebook.github.io/jest/`.

FAQs: heraldry - College of Arms. Accessed: January 2018. College of Arms. URL: `http://www.college-of-arms.gov.uk/resources/faqs`.

Ferraiolo, Jon, Fujisawa Jun and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. Accessed: October 2017. iuniverse, 2000.

Inkscape Team. *Inkscape: A vector drawing tool*. Accessed: October 2017. URL: `http://www.inkscape.org`.

JS Foundation. *webpack*. Accessed: October 2017. URL: `https://webpack.js.org`.

Kabra, Kulshekhar. *ts-jest*. Accessed: February 2018. URL: `https://github.com/kulshekhar/ts-jest`.

Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Accessed: February 2018. Pearson Education, 2009, p. 37.

– 'Design Principles and Design Patterns'. In: *Object Mentor* 1.34 (2000), p. 597.

MDN Contributors. *Document Object Model (DOM) - Web APIs | MDN*. Accessed: March 2018. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`.

Microsoft Corporation. *TypeScript - Javascript That Scales*. Accessed: September 2017. URL: `https://www.typescriptlang.org/`.

Nicolas, Sir N. Harris. *The Controversy between Sir Richard Scrope and Sir Robert Grosvenor in the Court of Chivalry*. Accessed: January 2018. London, Bentley, 1832, p. 404.

Palantir Technologies. *TSLint*. Accessed: September 2017. URL: `https://palantir.github.io/tslint/`.

Park, Thomas. *Bootswatch: Flatly*. Accessed: September 2017. URL: `https://bootswatch.com/flatly/`.

Python Software Foundation. *Python*. Accessed: April 2018. URL: `https://www.python.org/`.

Shoulson, Mark and Arnt Richard Johansen. *pyBlazon: Blazonry to SVG Converter*. Accessed: March 2018. 2008. URL: `http://web.meson.org/pyBlazon/`.

The jQuery Foundation. *jQuery*. Accessed: October 2017. URL: `https://jquery.com/`.

Torvalds, Linus and Junio Hamano. *Git: Fast Version Control System*. URL: `https://git-scm.com`.

Travis CI GmbH. *Travis CI - Test and Deploy with Confidence*. Accessed: November 2017. URL: `https://travis-ci.com`.

TypeDoc Contributors. *TypeDoc - Documentation generator for TypeScript projects*. Accessed: February 2018. URL: `http://typedoc.org`.

GLOSSARY

blazon The text describing how an escutcheon is to be drawn. 9, 11–13, 15, 16, 19, 20, 23–26, 33, 37, 38

charge Small emblems, such as fleur-de-lis and lions. 9, 11–14, 19–21, 23, 24, 27, 31, 37, 38

CI Continuous Integration. 17

DOM Document Object Model. See Section 3.2 for a description of the DOM. 16, 45

escutcheon The shield in the coat of arms. 9, 11–14, 22–24, 30, 33, 34

field The background colour of the escutcheon. 11, 14, 20, 24, 27

JSON JavaScript Object Notation. 15

minification Stripping out all unnecessary whitespace and tokens from code. 16

NLP Natural Language Processing. 15

NLTK Natural Language Tool Kit. 15

ordinary Geometric shapes on the escutcheon. 9, 11–14, 34, 37

payload The JSON data given to the web app from the parser. An example can be seen in Figure 3.1. 15, 20, 23, 24

quarter A quarter within a divided escutcheon. Examples of quarterly escutcheons can be seen in Figure 2.6. 9, 11, 23, 24, 27

SVG Scalable Vector Graphics. 13, 16, 19–21, 24, 26, 27, 31, 35, 38

tincture The colours and patterns for charges, ordinaries and fields. 11, 12, 14 *uglification* Transforming code by renaming all variables and functions into short, obfuscated names to reduce the footprint of assets. 16

variation How the field or charge is patterned. Variations can indicate patterns such as chequered or coloured lines. 11, 12, 38

W3C World Wide Web Consortium. 16

Appendix A

Interfaces and Enums

```
enum ETincture {
  /** For specifying Quarters */
  Quarterly = "quarterly",
  /** Gold/yellow */
  Or = "or",
  /** White */
  Argent = "argent",
  /** Blue */
  Azure = "azure",
  /** Red */
  Gules = "gules",
  /** Purple */
  Purpure = "purpure",
  /** Black */
  Sable = "sable",
  /** Green */
  Vert = "vert",
}

enum ECharge {
  Bend = "bend",
  Cross = "cross",
  Chief = "chief",
  Saltire = "saltire",
}

enum EQuarter {
  TL = "quarterly_tl",
  TR = "quarterly_tr",
  BL = "quarterly_bl",
  BR = "quarterly_br",
}

interface ICharge {
  charge: ECharge;
  sinister?: boolean;
  tincture?: ETincture;
}

interface IBlazon {
  field: ETincture;
  charges: ICharge[];
}
```

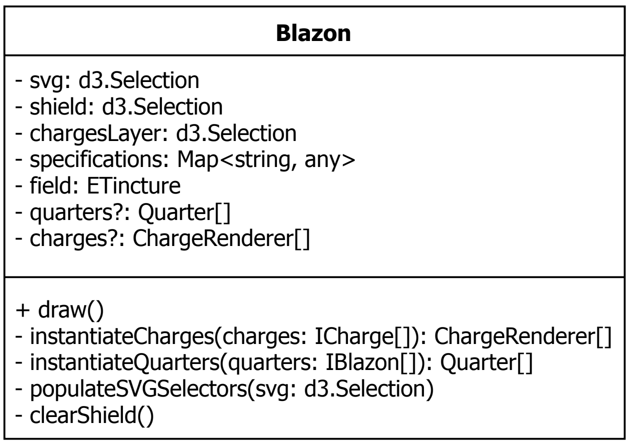
A question mark on a field in an `interface` denotes it as optional.

Appendix B
UML Diagrams

In these UML diagrams, `d3.Selection` is a data type defined by D3.js.[1] It contains a reference to an HTML element for use in DOM manipulation.

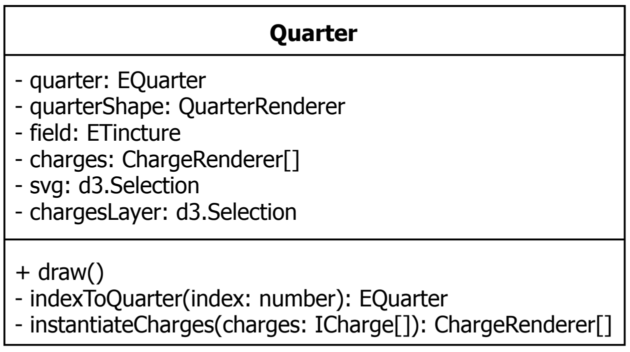
[1] Bostock, *D3.js - Data-Driven Documents*.

B.1 Second Design Iteration Diagrams



Blazon
- svg: d3.Selection
- shield: d3.Selection
- chargesLayer: d3.Selection
- specifications: Map<string, any>
- field: ETincture
- quarters?: Quarter[]
- charges?: ChargeRenderer[]
+ draw()
- instantiateCharges(charges: ICharge[]): ChargeRenderer[]
- instantiateQuarters(quarters: IBlazon[]): Quarter[]
- populateSVGSelectors(svg: d3.Selection)
- clearShield()

Figure B.1: `Blazon` UML.



Quarter
- quarter: EQuarter
- quarterShape: QuarterRenderer
- field: ETincture
- charges: ChargeRenderer[]
- svg: d3.Selection
- chargesLayer: d3.Selection
+ draw()
- indexToQuarter(index: number): EQuarter
- instantiateCharges(charges: ICharge[]): ChargeRenderer[]

Figure B.2: `Quarter` UML.

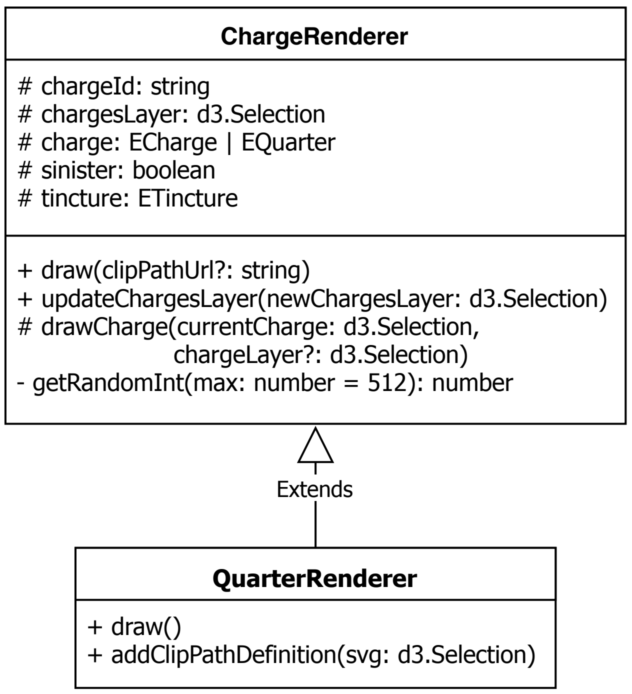

ChargeRenderer
# chargeId: string
# chargesLayer: d3.Selection
# charge: ECharge | EQuarter
# sinister: boolean
# tincture: ETincture
+ draw(clipPathUrl?: string)
+ updateChargesLayer(newChargesLayer: d3.Selection)
# drawCharge(currentCharge: d3.Selection, chargeLayer?: d3.Selection)
- getRandomInt(max: number = 512): number
Extends
QuarterRenderer
+ draw()
+ addClipPathDefinition(svg: d3.Selection)

Figure B.3: `ChargeRenderer` hierarchy and methods.

B.2 Third Design Iteration Diagrams

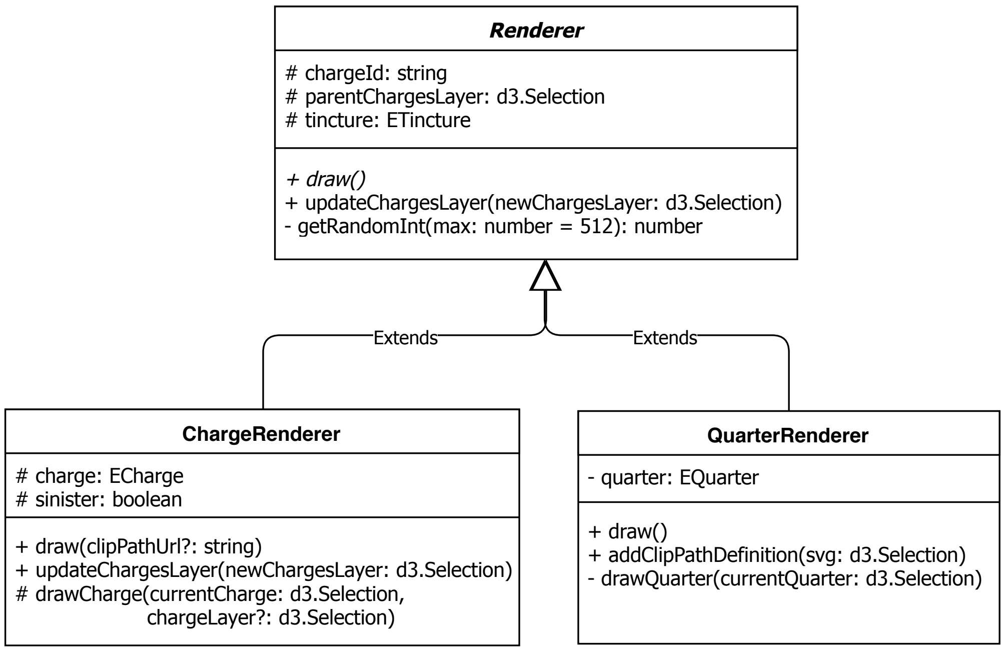

Renderer
# chargeId: string
# parentChargesLayer: d3.Selection
# tincture: ETincture
+ draw()
+ updateChargesLayer(newChargesLayer: d3.Selection)
- getRandomInt(max: number = 512): number
Extends
Extends
ChargeRenderer
# charge: ECharge
# sinister: boolean
+ draw(clipPathUrl?: string)
+ updateChargesLayer(newChargesLayer: d3.Selection)
# drawCharge(currentCharge: d3.Selection, chargeLayer?: d3.Selection)
QuarterRenderer
- quarter: EQuarter
+ draw()
+ addClipPathDefinition(svg: d3.Selection)
- drawQuarter(currentQuarter: d3.Selection)

Figure B.4: `Renderer` hierarchy and methods.

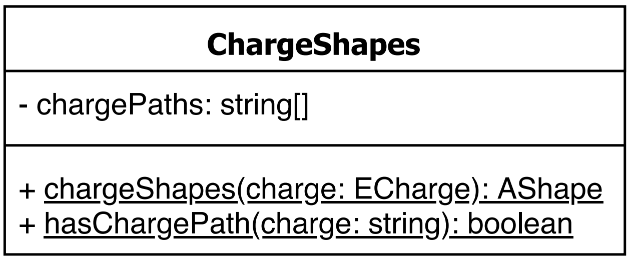

ChargeShapes
- chargePaths: string[]
+ chargeShapes(charge: ECharge): AShape
+ hasChargePath(charge: string): boolean

Figure B.5: `ChargeShapes` UML.

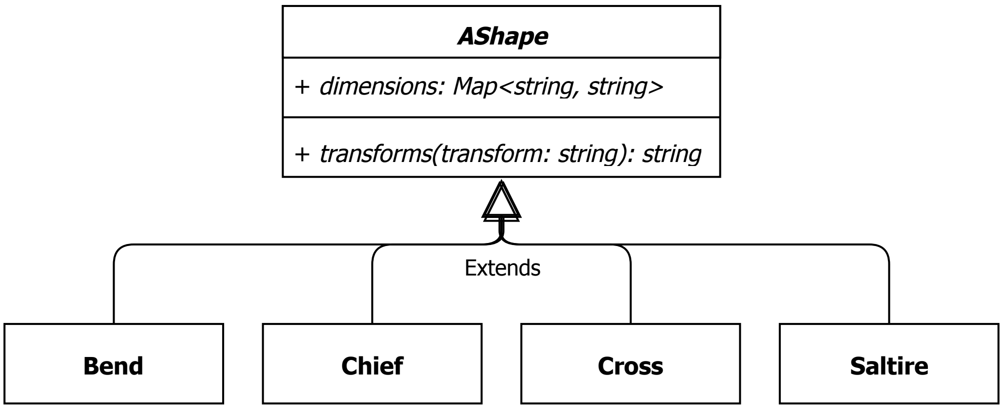

AShape
+ dimensions: Map<string, string>
+ transforms(transform: string): string
Extends
Bend
Chief
Cross
Saltire

Figure B.6: `AShape` hierarchy and methods.