



DISPLAYING HERALDIC BLAZONS

WILLIAM MATHEWSON

4th Year Project Report
Computer Science

SCHOOL OF INFORMATICS

UNIVERSITY OF EDINBURGH

2018

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 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

Copyright © 2018 William Mathewson

This L^AT_EX document class is derived from the ‘Tufte-L^AT_EX’ document class and is licensed under the Apache 2.0 license.

[HTTPS://GITHUB.COM/ANGUSP/TUFTE-LATEX](https://github.com/angusp/tufte-latex)

CONTENTS

1	<i>Introduction</i>	7
1.1	<i>Motivation</i>	7
1.2	<i>Contributions</i>	7
2	<i>Background</i>	9
2.1	<i>Heraldry</i>	9
2.2	<i>Related Works</i>	11
2.3	<i>Summary</i>	11
3	<i>Design</i>	13
3.1	<i>Core Concepts</i>	13
3.2	<i>External Dependencies</i>	14
3.2.1	<i>Development Dependencies</i>	14
3.3	<i>Iterative Design</i>	15
4	<i>Implementation</i>	17
4.1	<i>Basic Charge Rendering</i>	17
4.1.1	<i>First Design Iteration</i>	17
4.1.2	<i>First Design Implementation</i>	18
4.2	<i>Adding Quarterly Rendering</i>	19
4.2.1	<i>Second Design Iteration</i>	19
4.2.2	<i>Second Design Implementation</i>	20
4.2.3	<i>SOLID Design Principles</i>	21

4.3	<i>Refactoring Charge Rendering</i>	22
4.3.1	<i>Third Design Iteration</i>	22
4.3.2	<i>Third Design Implementation</i>	22
5	<i>Results and Discussion</i>	23
5.1	<i>Automated Testing</i>	23
5.2	<i>Rendering Testing</i>	23
6	<i>Conclusion</i>	25
6.1	<i>Overview</i>	25
6.2	<i>Further Work</i>	25
	<i>Bibliography</i>	27
	<i>Glossary</i>	29
	<i>Appendix A: Interfaces and Enums</i>	31
	<i>Appendix B: UML Diagrams</i>	33
B.1	<i>Second Design Iteration Diagrams</i>	33
B.2	<i>Third Design Iteration Diagrams</i>	34

1

INTRODUCTION

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, having been parsed by the back end. The back-end was written by my partner, Anthony Gallagher, and the writing of it will not be covered in this report. Important shared design elements will be covered however in Section 3.1.

1.1 *Motivation*

In 1874 — 4 years after his death — John Papworth's *Ordinary of British Armorial* was published.¹ In this work, he recorded approximately 50,000 entries of 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 honours project would make 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 that build family trees for people. Given the blazon, they would be able to construct the shields visually.

1.2 *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 algorithm

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 class 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 to you the many heraldic works of Charles Boutell and John Brooke-Little.

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 flagrant disregard for the rules introduces difficulty in parsing the blazons as the language loses some of its regularity.

Blazons have a few 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 2.1: 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

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.

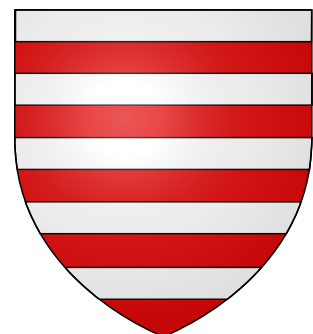


Figure 2.2: The shield of the town of Albert, France. *Barry of ten argent and gules*. Source: https://en.wikipedia.org/wiki/File:Blason_Albert.svg

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 used default CSS colours.

Blazons conventionally follow a form of starting 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 — drawn by the web app — in Figure 2.3.

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 (as drawn by the web app written for this project) can be seen in Figure 2.4.

Whilst the Scrope arms are prominent in heraldry, they are simplistic and indicative of mediæval 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 2.5.

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. The Heraldic Society advocates repeating tinctures to reduce ambiguity.⁵

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

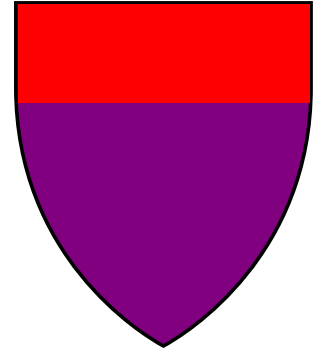


Figure 2.3: *Purpure, a chief Gules*.

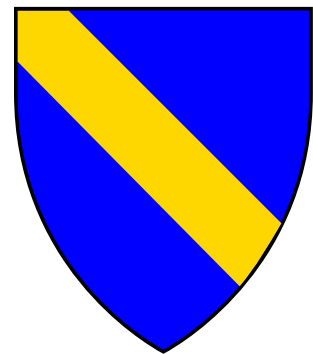
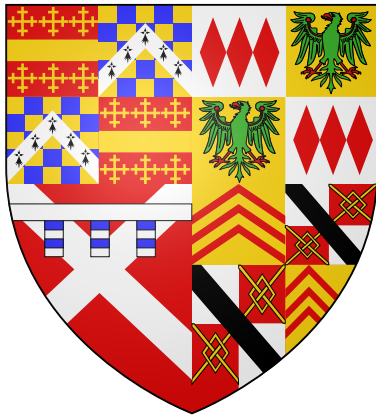


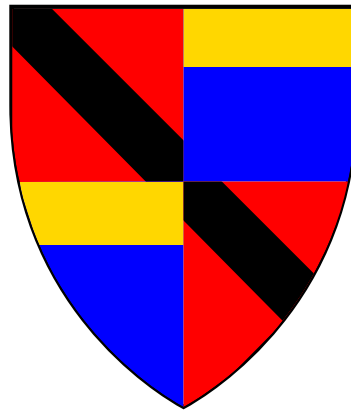
Figure 2.4: The Scrope escutcheon; *Azure, a bend Or*.

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.

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://en.wikipedia.org/wiki/File: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 2.5: Some examples of more complex coats of arms.

2.2 Related Works

Add more related works

Whilst many escutcheons have been drawn and uploaded to Wikimedia in Scalable Vector Graphics (SVG)⁶ format (some of which have been used in this report), many appear to have been created in Inkscape,⁷ rather than being created programmatically. Much work has been done in collecting and cataloguing blazons themselves — namely John Papworth as mentioned in Section 1.1.

⁶ Jon Ferraiolo, Fujisawa Jun and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. Accessed: October 2017. iuniverse, 2000.

⁷ 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 30.

3

DESIGN

3.1 Core Concepts

The two languages chosen for implementing this project were Python and TypeScript.¹ Python seemed like an obvious choice with its good support for Natural Language Processing (NLP) through the Natural Language Tool Kit (NLTK).² TypeScript is a typed superset of JavaScript written by Microsoft that compiles, or *transpiles*, to plain JavaScript. TypeScript was chosen as a nicer alternative to programming in pure JavaScript, thanks to the addition of powerful features such as types, access control and abstract classes.

The core design for this project centred around having a split stack; with a Python back-end parsing the blazon, serialising it into JSON and passing it to the TypeScript front-end, which then drew it onto the webpage. This allowed 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 schema for drawing payloads can be well-defined.

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

The TypeScript front-end received this payload, applied the *azure* CSS class to the field element, then drew a bend onto the field with an *or* CSS class.

Escutcheons were drawn using the SVG format for portability across browsers as well as the eponymous scalability of SVG images. This allowed drawn escutcheons to be embedded elsewhere with ease, either directly or through rendering the SVGs as other

¹ Microsoft Corporation. *TypeScript - Javascript That Scales*. Accessed: September 2017. URL: <https://www.typescriptlang.org/>.

² 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.

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.

```

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

```

Figure 3.1: Expected output from the Python back end, for a given blazon of “Azure, a bend Or”.

image formats via programmes like Inkscape.

For version control, `git`³ was used and all code was hosted on GitHub.

³Linus Torvalds and Junio Hamano. *Git: Fast Version Control System*. URL: <https://git-scm.com>.

3.2 External Dependencies

The front-end depended on a pair of libraries for SVG rendering and Document Object Model (DOM) manipulation: the selection library of `D3.js`⁴ and `jQuery`.⁵ `D3.js` provides many powerful functions for SVG and DOM manipulation, especially creating and editing SVG and HTML elements. It would also rely on `jQuery` for further DOM manipulation. The Mozilla Developer Network defines the DOM as such: “The Document Object Model (DOM) connects web pages to scripts or programming languages. Usually that means JavaScript, but modelling HTML, SVG, or XML documents as objects is not part of the JavaScript language. The DOM model represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree; with them you can change the document’s structure, style or content. Nodes can have event handlers attached to them. Once an event is triggered, the event handlers get executed.”⁶

⁴Mike Bostock. *D3.js - Data-Driven Documents*. Accessed: September 2017. URL: <https://d3js.org>.

⁵The jQuery Foundation. *jQuery*. Accessed: October 2017. URL: <https://jquery.com/>.

For further assets, `Sass`⁷ was used as a CSS pre-processor and `Bootstrap`⁸ was used for the base styling. `Webpack`⁹ was used to transpile TypeScript down to JavaScript — minifying and uglifying it in the process — and concatenate all source files and their dependencies into one main JavaScript *bundle* 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 have smaller asset payloads to download than the original, raw source code.

⁶MDN Contributors. *Document Object Model (DOM) - Web APIs* | MDN. Accessed: March 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

⁷Hampton Catlin, Natalie Weizenbaum and Chris Eppstein. *Sass: Syntactically Awesome Stylesheets*. Accessed: September 2017. URL: <https://sass-lang.com/>.

⁸Thomas Park. *Bootstrap: Flatly*. Accessed: September 2017. URL: <https://bootstrap.com/flatly/>.

⁹JS Foundation. *webpack*. Accessed: October 2017. URL: <https://webpack.js.org>.

3.2.1 Development Dependencies

To maintain code readability and prevent bugs, `TSLint`¹⁰ was used

¹⁰Palantir Technologies. *TSLint*. Accessed: September 2017. URL: <https://palantir.github.io/tslint/>.

and set up to automatically run as part of the Travis Continuous Integration (CI)¹¹ service, causing a build to fail if the linter detected a style violation. For unit testing, Jest¹² (with ts-jest¹³ for TypeScript support) was used, especially for its powerful mocking and expectation matcher functionality. Automated documentation generation was provided by TypeDoc.¹⁴

3.3 *Iterative Design*

Iterative design was used as the design process for writing the code for this project. Iterative design is a cyclical process of designing, prototyping and evaluating. 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 allows to address different functionality as separate tasks, building on top of one another. It also allows to heavily refactor a project whilst staying in one cycle.

¹¹ Travis CI GmbH. *Travis CI - Test and Deploy with Confidence*. Accessed: November 2017. URL: <https://travis-ci.com>.

¹² Inc. Facebook. *Jest - Delightful JavaScript Testing*. Accessed: February 2018. URL: <https://facebook.github.io/jest/>.

¹³ Kulshekhar Kabra. *ts-jest*. Accessed: February 2018. URL: <https://github.com/kulshekhar/ts-jest>.

¹⁴ TypeDoc Contributors. *TypeDoc - Documentation generator for TypeScript projects*. Accessed: February 2018. URL: <http://typedoc.org>.

4

IMPLEMENTATION

4.1 *Basic Charge Rendering*

4.1.1 *First Design Iteration*

The first design iteration had a specific focus on basic charge drawing, with a plan to begin a second iteration for adding quarterly rendering with lessons learnt from implementing charge drawing in the first iteration. The second design iteration is described in Section 4.2.1.

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 helped support a reliable entry point for the drawing logic as it was able to easily select the shield element to begin appending other SVG elements to. Appending these SVG elements allowed layering to be achieved as SVG orders layers based on the order of elements in the document. This was used to great effect later 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 payload, applied the value as the CSS class for the shield and then moved onto the charges. It iterated over the *charges* array in the payload, 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 payload, all would be drawn according to the array ordering.

4.1.2 First Design 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 transpiled 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}.

The web app had a single entry point of `drawShield(blazon)`, where `blazon` was the whole JSON payload returned from the `/_parse` endpoint. (See Figure 3.1 for an example payload.) This presented a problem initially 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`) causes TypeScript to produce an error 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 2 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.

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)`.

When each charge node is created in `drawCharge`, it is assigned an id. This id is formed from the name of the charge, followed by a random number in the range 1–512 inclusive with the hope that the range is large enough to lack overlaps.

To draw shapes in SVG, a path node requires a `'d'` attributes which contains the commands for drawing said shape. To generate all these attributes, I drew all the charge shapes in Inkscape and extracted the `'d'` attribute 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 problems when using the *chief* charge (a chief charge being displayed in Figure 2.3). The chief charge was drawn using a `rect` 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

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 `A()` and `B()` should be defined immediately after function `A()`.

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

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

a `ChargeShapes` class to encapsulate the charges and their attributes. This class provided one public member, `chargePaths` which was of the type `Map<string, Map<string, string>>`. Having 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.

The final part of `drawCharge` applied a transform to the path if the payload 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 4.1.

Whilst this simple implementation worked well for drawing basic escutcheons, like Figure 2.3 and Figure 4.1, it wasn't able to draw more complex shields like those seen in Figure 2.5.

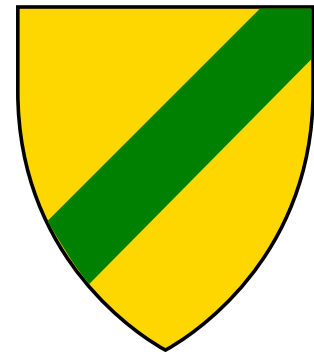


Figure 4.1: Or, a bend sinister Vert.

4.2 Adding Quarterly Rendering

4.2.1 Second Design Iteration

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 also 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 follow the principles of delegation, decoupling, the Single Responsibility⁵ and Open/Closed principles.⁶

The Single Responsibility principle dictates that a class should have one, and only one, reason to change. In *Design Principles and Design Patterns*,⁷ Martin defines a responsibility as a reason to change, such that a class should only change if that one responsibility changes. 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 use 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. Thus, another class depending on the class extending the abstract class can rely on the interface without having to know about the internals.

⁵ Robert C Martin. 'Design Principles and Design Patterns'. In: *Object Mentor* 1.34 (2000), p. 597.

⁶ Martin, 'Design Principles and Design Patterns'.

⁷ Martin, 'Design Principles and Design Patterns'.

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.

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 having a field `charges: ICharge[]`, it would contain a field `quarters: IBlazon[]`. This worked as quarters are treated as their own small 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 2.5. It was then possible to change functionality of the app depending on the contents of the `field` key.

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 its child elements and any of its attributes are also inherited. I used the `<g>` elements for both grouping together charges and quarters, but also explicitly named layers for the shield outline and the charges within.

4.2.2 Second Design Implementation

`Blazon` took 2 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. The `data` argument was to take the full JSON payload 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 also populated extra fields with selections of the whole shield element and the `<g id="charge_layers">` element, defined within the `svg` selection. The `chargesLayer` field is needed for telling the renderer classes where to draw their shapes. Depend-

UML diagrams for all classes mentioned in this subsection can be found in Appendix B.1.

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

ing on the contents of the payload, the constructor then instantiates new `ChargeRenderer` objects for all the charges, or instantiates 4 new `Quarter` objects for all the quarters.

The new `Quarter` class relied on the order of the `quarters` array in the payload to determine which quarter it was; with numbering starting at 0 in the top left quarter, going left to right, ending at 3 in the bottom right.

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

`QuarterRenderer`

4.2.3 SOLID Design Principles

Refactor all of this, see photos, break principles into relevant sections.

SOLID object-oriented design is a collection of 5 design principles:

- *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;
- *The Interface Segregation Principle* — make fine grained interfaces that are client specific and
- *The Dependency Inversion Principle* — depend on abstractions, not on concretions.

4.3 *Refactoring Charge Rendering*

4.3.1 *Third Design Iteration*

4.3.2 *Third Design Implementation*

5

RESULTS AND DISCUSSION

5.1 *Automated Testing*

5.2 *Rendering Testing*

6

CONCLUSION

6.1 *Overview*

6.2 *Further Work*

BIBLIOGRAPHY

- 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.
- Contributors, MDN. *Document Object Model (DOM) - Web APIs* | MDN. Accessed: March 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.
- Contributors, TypeDoc. *TypeDoc - Documentation generator for TypeScript projects*. Accessed: February 2018. URL: <http://typedoc.org>.
- Corporation, Microsoft. *TypeScript - Javascript That Scales*. Accessed: September 2017. URL: <https://www.typescriptlang.org/>.
- 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.
- Foundation, JS. *webpack*. Accessed: October 2017. URL: <https://webpack.js.org>.
- Foundation, The jQuery. *jQuery*. Accessed: October 2017. URL: <https://jquery.com/>.
- GmbH, Travis CI. *Travis CI - Test and Deploy with Confidence*. Accessed: November 2017. URL: <https://travis-ci.com>.

- 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.
- 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.
- Park, Thomas. *Bootswatch: Flatly*. Accessed: September 2017. URL: <https://bootswatch.com/flatly/>.
- Team, Inkscape. *Inkscape: A vector drawing tool*. Accessed: October 2017. URL: <http://www.inkscape.org>.
- Technologies, Palantir. *TSLint*. Accessed: September 2017. URL: <https://palantir.github.io/tslint/>.
- Torvalds, Linus and Junio Hamano. *Git: Fast Version Control System*. URL: <https://git-scm.com>.

GLOSSARY

blazon The text describing how an escutcheon is to be drawn. 7, 9–11, 13, 14, 17, 18, 20

charge Small emblems, such as fleur-de-lis and lions. 7, 9–11, 17–21

CI Continuous Integration. 15

DOM Document Object Model. See Section 3.2 for a description of the DOM. 14, 33

escutcheon The shield in the coat of arms. 7, 9–11, 19

field The background colour of the escutcheon. 9, 11, 18, 20, 21

JSON JavaScript Object Notation. 13

minification Minification of JavaScript assets involves stripping out all unnecessary whitespace and tokens. 14

NLP Natural Language Processing. 13

NLTK Natural Language Tool Kit. 13

ordinary Geometric shapes on the escutcheon. 9–11

payload The JSON payload given to the web app from the parser. An example can be seen in Figure 3.1. 13, 17–21

quarter A quarter within a divided escutcheon. Examples of quarterly escutcheons can be seen in Figure 2.5. 7, 9, 20, 21

SVG Scalable Vector Graphics. 11, 13, 14, 17, 18, 20, 21

tincture The colours and patterns for charges, ordinaries and fields. 9–11

transpile Transpilation involves compiling from language to another, in this case compiling from TypeScript to JavaScript. 13, 14, 18

uglification Uglification transforms the JavaScript code by renaming all variables and functions into short, obfuscated names to reduce the footprint of the assets. 14

variation Variations describe how the field or charge is patterned. Variations can indicate patterns such as chequered or coloured lines. 9, 10

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[];  
}
```


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

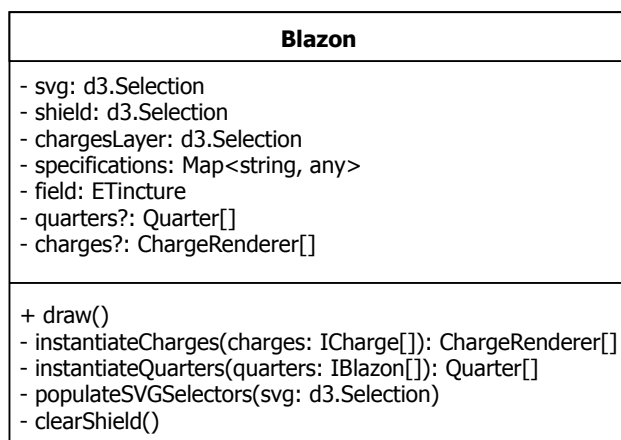


Figure B.1: Blazon UML.

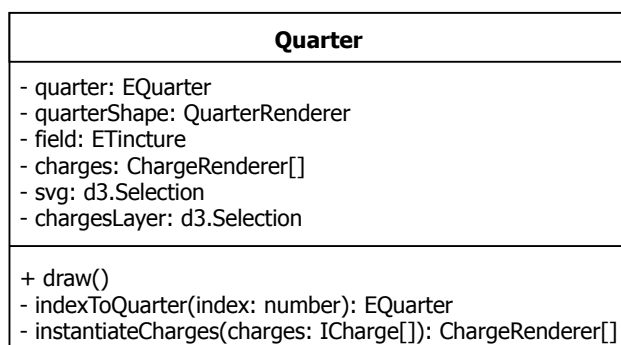


Figure B.2: Quarter UML.

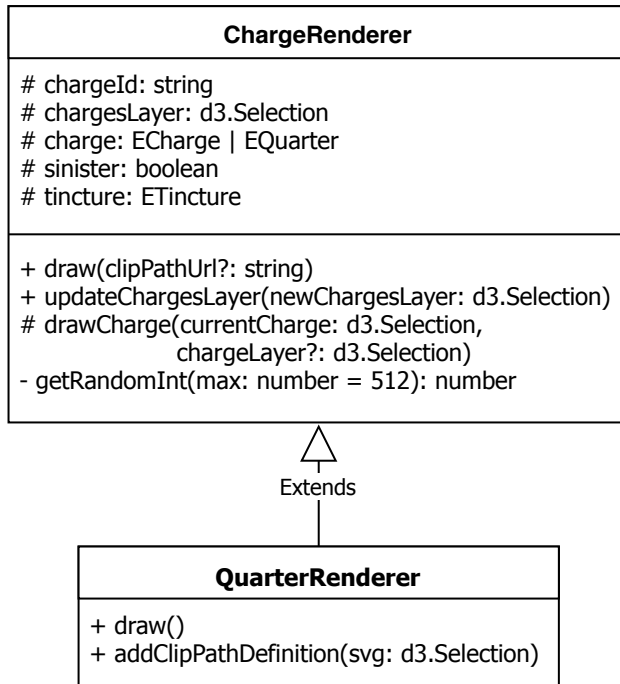


Figure B.3: ChargeRenderer hierarchy and methods.

B.2 Third Design Iteration Diagrams

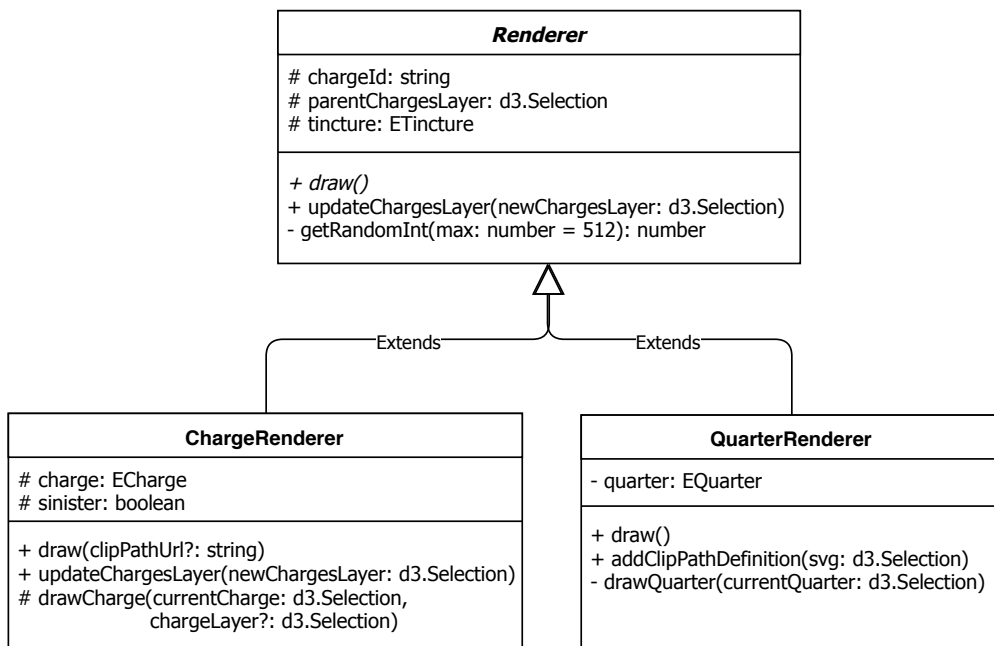


Figure B.4: Renderer hierarchy and methods.

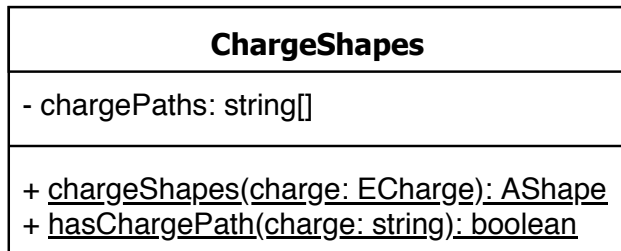


Figure B.5: ChargeShapes UML.

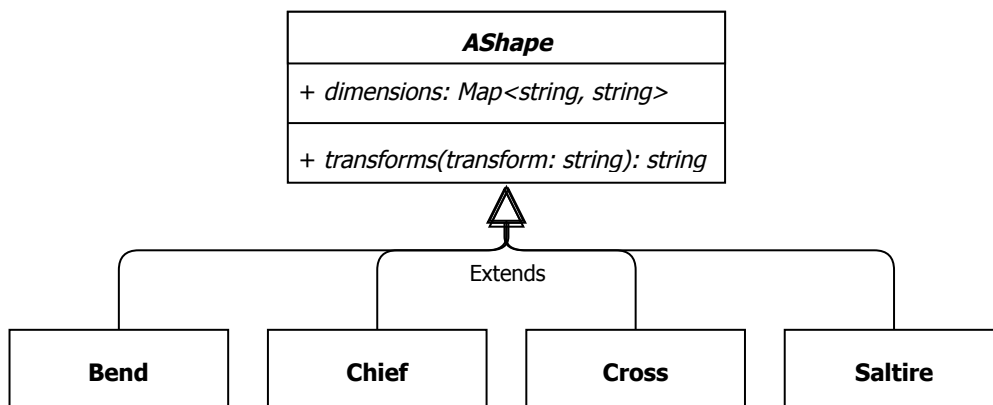


Figure B.6: AShape hierarchy and methods.