# 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, particularly Connie and Paul, who helped keep me going through the long, arduous journey that was this project and 4$^{th}$ 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

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 Armorials* 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.


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 flagrant 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 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.

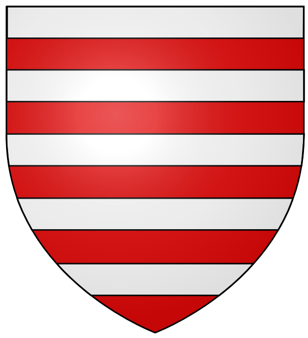


Figure 2.2: The shield of the town of Albert, France. *Barry of ten argent and gules*. Source: `https://en.wikipedia.org/wiki/File:Blason_Albert.svg`

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.

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

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.

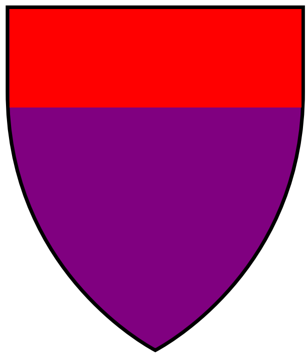


Figure 2.3: *Purpure, a chief Gules.*

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.

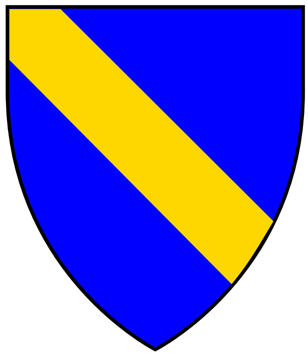


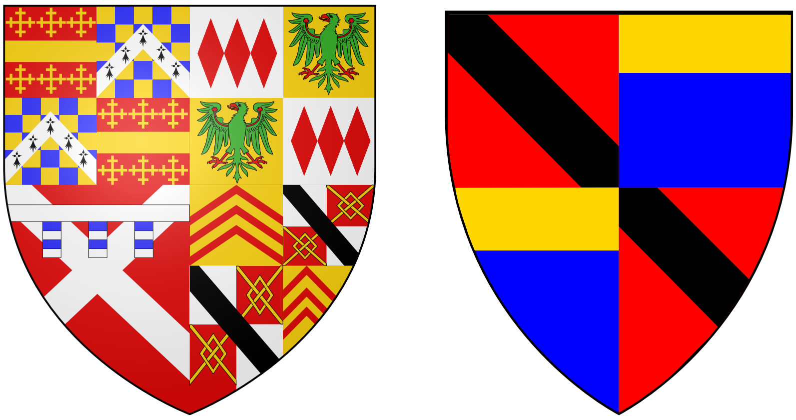
Figure 2.4: The Scrope escutcheon; *Azure, a bend Or*.

[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.

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]

[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 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 36.

# 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. As a note to the reader, to maintain interoperability with JavaScript, TypeScript uses trailing types, rather than preceding types as in C. An example being that an `int` defined in C would be `int limit`, but in TypeScript, this would be `limit: number`. (JavaScript/TypeScript also has a unified `number` type to handle both `float`s and `int`s.)

[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.

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.

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 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

```
{
  "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".

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 image formats via programmes like Inkscape.

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

[3] Linus 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]

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

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

[6] 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[7] was used as a CSS pre-processor and Bootswatch[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

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

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

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

the original, raw source code.

### 3.2.1 Development Dependencies

To maintain code readability and prevent bugs, TSLint[10] was used 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, Jest[12] (with ts-jest[13] for TypeScript support) was used, especially for its powerful mocking and expectation matcher functionality. Automated documentation generation was provided by TypeDoc.[14]

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

[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`.

## 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.

# 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].

[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()`.

#### *`drawShield(blazon)`*

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.

[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 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 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.

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

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.

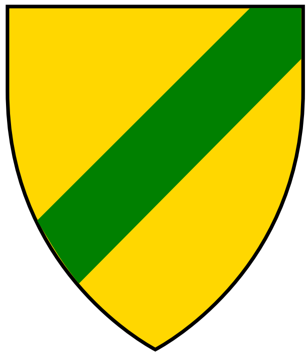


Figure 4.1: *Or, a bend sinister Vert.*

### 4.1.3 First Design Evaluation

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.

## 4.2 Adding Quarterly Rendering

### 4.2.1 Second Design Iteration

#### 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 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 Responsibility[5] and Open/Closed principles.[6]

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

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

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.

[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.

## *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 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 2.5. 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 values of the enumerable options being the `id`s of the SVG elements and the `path`s in the `ChargeShapes.chargePaths Map`.

### *Using SVGs for Great Good*

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*

The Figure references in the following headings refer to the UML diagrams in Appendix B.1.

### `Blazon` *(Figure B.1)*

`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. Depending 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.

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

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

### `Quarter` *(Figure B.2)*

The new `Quarter` class relied on the order of the `quarters` array in the payload 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 `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 Section 4.2.2) 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.

### *`ChargeRenderer` (Figure B.3)*

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 `chargesLayer` that charges were being drawn in to. I decided to add a new method to update the `chargesLayer` after `ChargeRenderer` instantiation.

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

### *`QuarterRenderer` (Figure B.3)*

`QuarterRenderer` extended `ChargeRenderer`, with few changes to what it inherited. In the `constructor`, 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.

### 4.2.3 *Second Design Evaluation*

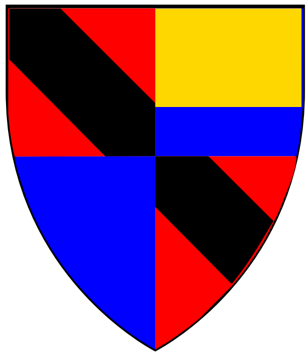


Figure 4.2: The second iteration rendering of Figure 2.5

This design iteration had brought quarterly rendering a lot closer to fruition, as seen in Figure 4.2, but I had difficulties maintaining clean code whilst adding the functionality to apply quarter-specific transformations to scale and move charges. To add this, I had to redesign both how `ChargeShapes` and how both `Renderer` classes worked.

## 4.3 *Refactoring Charge Rendering*

### 4.3.1 *Third Design Iteration*

#### `Renderer` *Hierarchy*

This new design made use of the *Liskov Substitution* and Open/Closed principles[9] (the Open/Closed principle being described in Section 4.2.1). The Liskov Substitution principle dictates that derived classes must be substitutable for their base class. This is well implemented with abstract classes 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 principle, 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 helped refine and enforce the previously specified API of render classes having a main entry-point of `draw()`. Using an abstract class not only gave assurances to classes that used its implementations that there would always be a `draw()` function, but also gave child classes the shared functions that both required whilst allowing them to specialise in their individual use cases. It also meant that previously ambiguous attribute names could be named more specifically for the particular `Renderer` implementation. An example of this being that where `ChargeRenderer` previously had an attribute of `charge` with a union type of `ECharge | EQuarter`, the `charge` attribute could now have a type of just `ECharge` and `QuarterRenderer` could have an attribute of `quarter` with type `EQuarter`.

#### `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` in the `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` 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 *Third Design Implementation*

The Figure references in the following headings refer to the UML diagrams in Appendix B.2.

#### `Renderer` *(Figure B.4)*

The new `Renderer` class contained 3 protected, inherited attributes, `chargeId: string`, `tincture: ETincture` and `parentChargesLayer: d3.Selection`. Whilst the clas had 3 attributes, only `tincture` and `parentChargesLayer` were assigned in the constructor; inherited classes were given responsibility of generating their own `ids`. As mentioned in Section 4.3.1, `Renderer` defined 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.

#### `ChargeRenderer` *(Figure B.4)*

Now inheriting from `Renderer`, `ChargeRenderer` had just two attributes defined on it: `charge: ECharge` and `sinister: boolean`. The `draw` method now had an optional argument of `quarter: EQuarter` which would correspond to the `id` of the `<clipPath>` that contained this charge. As before, the method performed environment setup in the SVG document ready for the new charge, then called out to `drawCharge`, however after performing all this, `draw` now also called `applyTransforms`. `applyTransforms` took in the `currentCharge` element and an optional `quarter` argument, built a `transform` string based on the parameters, fetched the appropriate transformation from `ChargeShapes` and applied it to the `currentCharge`.

### `QuarterRenderer` *(Figure B.4)*

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

### `AShape` *and* `AQuarterShape` *(Figure B.6)*

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

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

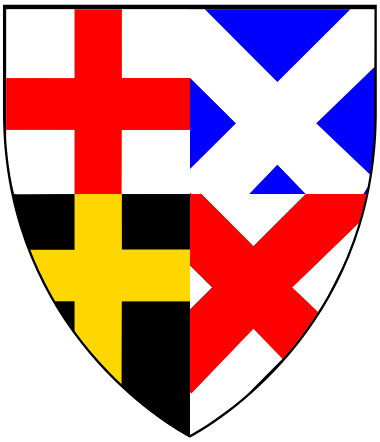
### `ChargeShapes` *(Figure B.5) and* `QuarterShapes`

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

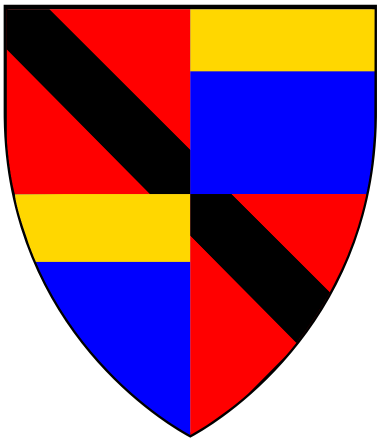
Again, to mirror the new `ChargeShapes` implementation, `QuarterShapes` was also defined with a sole `quarterShapes(quarter: EQuarter)` which returned a new object for the relevant `AQuarterShape`. The implementations for both `QuarterShapes` and `AQuarterShape` were unnecessarily verbose, but were done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

### 4.3.3 *Third Iteration Evaluation*

The major goals of the project were now realised, with quarterly shields being reliably drawn. Some examples as drawn by the web app can be seen in Figure 4.3.



(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 4.3: Some quarterly examples drawn by the web app.

# 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. 5, 9, 11–13, 15, 16, 19, 20, 22, 23

*charge* Small emblems, such as fleur-de-lis and lions. 9, 11–13, 19–27

*CI* Continuous Integration. 17

*DOM* Document Object Model. See Section 3.2 for a description of the DOM. 16, 39

*escutcheon* The shield in the coat of arms. 9, 11–13, 21, 22

*field* The background colour of the escutcheon. 11, 13, 20, 22, 24

*JSON* JavaScript Object Notation. 15

*minification* Minification of JavaScript assets involves stripping out all unnecessary whitespace and tokens. 16

*NLP* Natural Language Processing. 15

*NLTK* Natural Language Tool Kit. 15

*ordinary* Geometric shapes on the escutcheon. 11–13

*payload* The JSON payload given to the web app from the parser. An example can be seen in Figure 3.1. 15, 19–23

*quarter* A quarter within a divided escutcheon. Examples of quarterly escutcheons can be seen in Figure 2.5. 9, 11, 22–24

*SVG* Scalable Vector Graphics. 6, 13, 16, 19–24, 26

*tincture* The colours and patterns for charges, ordinaries and fields. 11–13 *transpile* Transpilation involves compiling from language to another, in this case compiling from TypeScript to JavaScript. 15, 16, 20

*uglification* Uglification transforms the JavaScript code by renaming all variables and functions into short, obfuscated names to reduce the footprint of the assets. 16

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

# 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

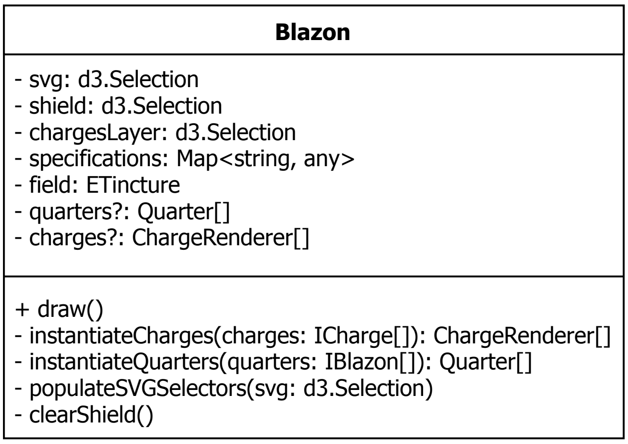



Figure B.1: `Blazon` UML.

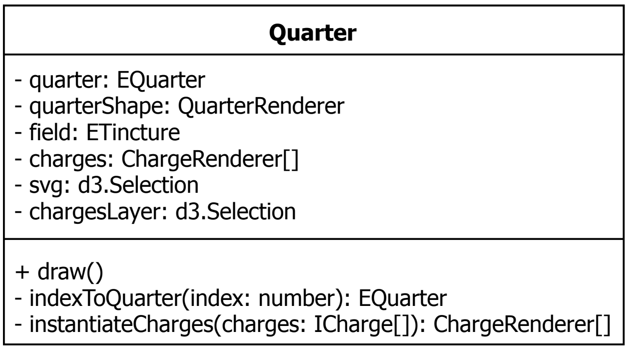



Figure B.2: `Quarter` UML.

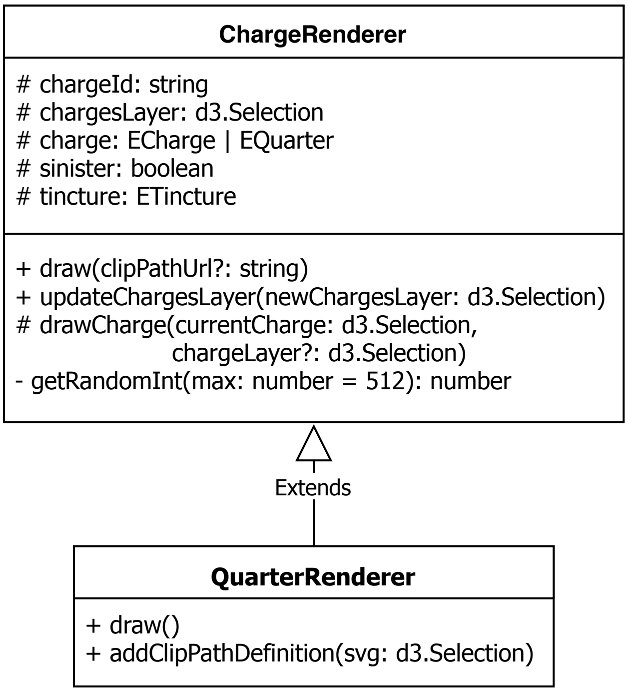



Figure B.3: `ChargeRenderer` hierarchy and methods.

## B.2 *Third Design Iteration Diagrams*

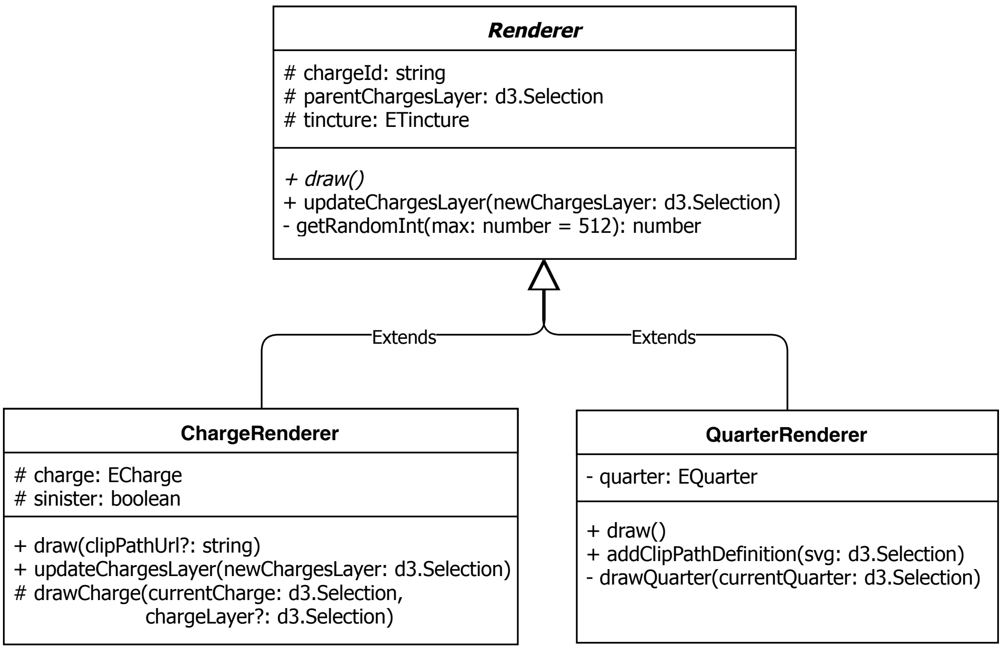



Figure B.4: `Renderer` hierarchy and methods.

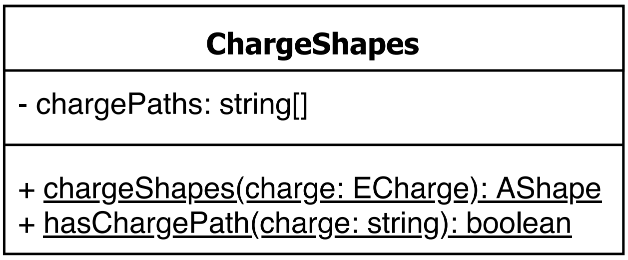



Figure B.5: `ChargeShapes` UML.

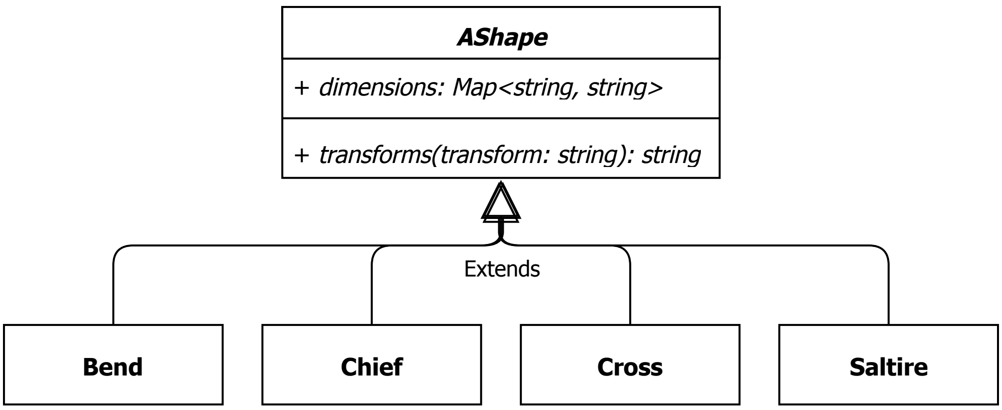



Figure B.6: `AShape` hierarchy and methods.