Design Patterns for Angular Hotdraw

Kirita-Rose Escott
Victoria University of Wellington
Wellington, New Zealand
kirita-rose.escott@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington
Wellington, New Zealand
kjx@ecs.vuw.ac.nz

ABSTRACT

The number of web frameworks available for use is growing. Web developers need to learn how to use them effectively and efficiently. Working through the design patterns presented in this paper for the Angular Hotdraw application assists web developers in this task. Web developers should be able to make a start with a the Angular web framework and have a foundation to learn from.

KEYWORDS

design patterns, pattern mining, angular, web frameworks

ACM Reference Format:
Kirita-Rose Escott and James Noble. 2019. Design Patterns for Angular Hotdraw. In *24th European Conference on Pattern Languages of Programs (EuroPLoP '19), July 3–7, 2019, Irsee, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361149.3361185

INTRODUCTION

The number of web frameworks available for use is growing, therefore there is a need for developers to be able to learn how to use them efficiently and effectively. A developer needs to have some assistance to use a web framework for the first time. This paper presents a set of design patterns, from which to learn how to implement a simple version of the Hotdraw application using the Angular web framework [2, 5, 6, 12, 13]. By working through these patterns, developers should be able to make a start with Angular and have a foundation to learn from.

PATTERN MINING

The patterns discussed in this paper were mined by following a pattern mining process [7, 9, 11]. The process consists of discovering elements, clustering elements, extracting the essential message from each cluster and analysing the essential messages to produce design patterns. The patterns are mined during the implementation of Angular Hotdraw [3, 4].

HOTDRAW

The traditional definition of Hotdraw is that it is a framework for structured drawing editors[8]. It can be used to build editors for specialized two-dimensional drawings such as schematic diagrams, blueprints, music, or program designs. The elements of these drawings can have constraints between them, they can react to commands by the user, and they can be animated. The editors can be a complete application, or they can be a small part of a larger system [8]. In the context of this paper, Angular Hotdraw is a simple single page web application users can use to render, move and remove shapes on a canvas.

PATTERNS

This paper presents 10 patterns for Angular Hotdraw (see *Table 1*). The first pattern, Angular Architecture, outlines the basic Angular application architecture. Page Content, Component and Graphic Content describe how basic content can be rendered in the application. The patterns Pen, Shape, Point, and Event, describe how to respond to mouse events in order to render shapes and lines. Tool and Service are the final patterns which outline basic tool selection and use in the application.

1	Angular Architecture
2	Component
3	Page Content
4	Graphic Content
5	Shape
6	Pen
7	Point
8	Event
9	Tool
10	Service

Table 1 : Patterns

1 ANGULAR ARCHITECTURE PATTERN

Context: Angular is an application framework that offers you the ability to run your application over the web. An Angular web application is made up of a number of different elements. These elements need to be in some form of structure to enable the application to be able to access and manipulate them.

Problem: How do you describe the architecture of your application?

Solution: Create the root NgModule in the application module typescript file to describe the system architecture. The @NgModule() decorator is a function that takes a single metadata object, whose properties describe the module [6].

The properties discussed in this paper are declarations, imports, providers, and the bootstrap. The declarations are the components, directives and pipes that belong to the NgModule. The imports are classes exported by other modules that are needed by components declared in this NgModule. The providers are services which can become accessible in all elements of the app. Finally, the bootstrap in the main application view. Any element used throughout the application must be declared within the NgModule to avoid errors at compile time.

Example: An NgModule describes how the application elements fit together. Every application has at least one Angular module. There are three components that make up the general architecture of the Hotdraw application, AppComponent, CanvasComponent, and DrawingToolComponent. The Hotdraw application imports the BrowserModule. The DrawingToolService is a provider. The root NgModule sets the bootstrap property, The AppComponent is added to the property to be inserted into the browser DOM. This relationship can be shown in Figure 1 : NgModule Relationship Model.

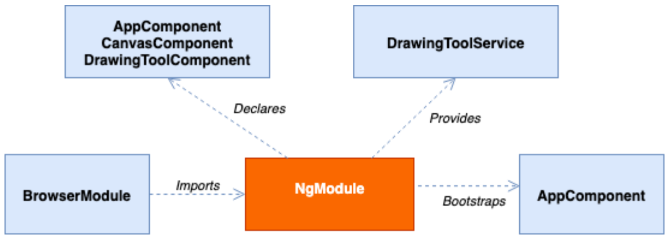

Figure 1: NgModule Relationship Model

The following code demonstrates how the root NgModule in Figure 1 is declared.

hotdraw/src/app/app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    DrawingToolComponent,
    CanvasComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [DrawingToolService],
  bootstrap: [AppComponent]
})
```

A more complex application, would have a larger number of elements surrounding the NgModule in the diagram and a larger number of declarations within the application module typescript file.

Related Patterns: Page Content (3) as the page is first declared in the system architecture.

2 COMPONENT PATTERN

Context: An application needs to know what the content is and how to modify it. In an application, components are building blocks. In the Angular Hotdraw application, there are three main component building blocks: `AppComponent`, `CanvasComponent` and `DrawingToolComponent`. Figure 2: Tool Component - Selector, is the page content defined in the `DrawingToolComponent`.

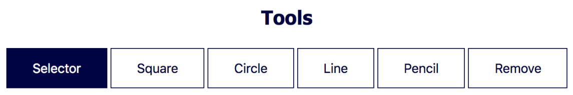

Figure 2: Tool Component - Selector

Problem: How do you create and modify the content of a Component?

Solution: Create a typescript file, an HTML file and a CSS file. These three files are are used to create and modify the content of a component in an Angular web application.

The typescript file is where the component is declared as a component and where its metadata is stored. The HTML file is the file where the content for this specific component is declared. The CSS file is where the variables used for styling this component are declared. Figure 3: Component Structure demonstrates this relationship.

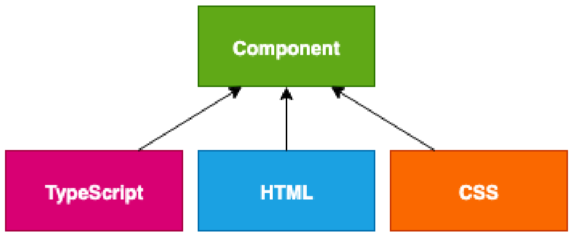

Figure 3: Component Structure

These files are tied together by declaring the `@Component` decorator in the typescript file. The `selector`, `templateUrl`, and `styleUrls` properties are used to reference the HTML and CSS files, as well as define a custom tag for this component. It should be noted that components can have multiple CSS files in cases where styling is shared to reduce code duplication.

Content can be declared in the same manner as in Page Content (3). To modify the content, declare interactive HTML elements such as buttons using the button tag, `<button></button>`. You can use the built in Angular directive `(click)` to determine what behaviour is performed when the user clicks on the button. Behaviour, or logic, is declared in methods in the typescript file.

Example: The following code demonstrates the declarations of the selector, HTML and CSS file for the `DrawingToolComponent` in the `@Component` decorator.

The selector is `drawing-tool`, this is the custom tag that can be called from other component's HTML files to display the component. The `drawing-tool.component.html` file is the HTML file where the contents of the component are declared. The component has only one CSS file in this case, the `drawing-tool.component.css` file, which is where the styling for this component is declared.

hotdraw/src/app/drawingtool.component.ts

```
@Component({
    selector: 'drawing-tool',
    templateUrl: './drawingtool.component.html',
    styleUrls: ['./drawingtool.component.css']
})
```

The following code demonstrates the declaration of the component's content in the `drawingtool.component.html` file. The `<h2></h2>` has been extended to include a class, which refers to a class declared in the CSS file. The `button-header` class determines the style of the header when it is rendered in the web browser. The built in Angular directive `[ngClass]` adds and removes CSS classes on an HTML element. This is demonstrated on the `button` elements which are styles are determined by a ternary operator. The `selectCurrentTool()` method is called with a parameter, which is the name of the tool being selected (ie. 'selector' for the selector tool or 'line' for the line tool.

hotdraw/src/app/drawingtool.component.html

```
<h2 class="button-header">
    Tools
</h2>
<button [ngClass]="selectedTool.getName()===
'selector'? 'selectedButton' :
'selectableButton'"(click)=
    "selectCurrentTool('selector')">
    Selector
</button>
...
<button [ngClass]="selectedTool.getName()===
'line' ? 'selectedButton' :
'selectableButton'"(click)=
    "selectCurrentTool('line')">
    Line
</button>
...
```

The following code demonstrates CSS styling of the component in the `drawingtool.component.css` file. The `button-header` class describes the `colour` and `font` of the header. The `selectableButton` and `selectedButton` classes describe the presentation variables of the buttons such as `background-color`, `padding`, and `font-size`. The classes defined below are applied to the `<h2></h2>` and `<button></button>` elements, described in the HTML file above, respectively.

hotdraw/src/app/drawingtool.component.css

```
.button-header{
    color: #000045;
    font-family: Tahoma;
}

.selectableButton {
    background-color: white;
    border-color: #000045;
    color: #000045;
    padding: 15px 32px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    margin: 4px 2px;
    cursor: pointer;
}

.selectedButton {
    background-color: #000045;
    border-color: #000045;
    color: white;
    padding: 15px 32px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    margin: 4px 2px;
    cursor: pointer;
}
```

Figure (4): Tool Component - Line, demonstrates the relationship between the three files in action. Either the `selectableButton` class or the `selected Button` class has been applied to each button as a result of the ternary. Referring back to Figure 2: Tool Component - Selector, you can see that the `selectedButton` class has been applied to the 'Selector' button, whereas below in Figure (4): Tool Component - Line, `selectedButton` class has been applied to the 'Line' button.

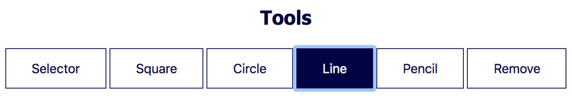

Figure 4: Tool Component - Line

Related Patterns: Tool (9) as it is the basic setup to enable tool selection in the Angular Hotdraw application.

3 PAGE CONTENT PATTERN

Context: A web application will present a user with an empty page by default. The application needs a method of determining page contents in order to know what to display to the application user.

Problem: How can you display content to users on a web page?

Solution: You can declare content in the HTML file, which will in turn be displayed to the user when the application is run. You can use predefined tags to label content such as 'heading', 'image', and 'paragraph'. You can also create custom tags which reference the HTML files of components you wish to display in the application. Browsers that run applications do not display the tags, but use them to render the content of the page. In the Angular web framework, page content is determined in the HTML file of the main application component. This component is typically named `AppComponent`.

Example: In the Angular Hotdraw application NgModule declares three components `AppComponent`, `DrawingToolComponent`, and `CanvasComponent`.

As `AppComponent` is the main application component, reference to other components, through the use of custom tags, as page content is determined there.

The following code, demonstrates the use of HTML to determine the page content of the Hotdraw application. The `<h1></h1>` tag is used to determine the 'heading' of the page. The `{{title}}` refers to a variable defined as `title = 'HotDraw Angular';` in the `AppComponent` type script file. The HTML includes the custom tags `<drawing-canvas></drawing-canvas>` and `<drawing-tool></drawing-tool>` which refer to the `CanvasComponent` and `DrawingToolComponent` respectively.

hotdraw/src/app/app.component.html

```
<div align="center">
    <h1 class="app-header">
        {{title}}
    </h1>
    <drawing-canvas></drawing-canvas>
</div>
<div align="center">
    <drawing-tool></drawing-tool>
</div>
```

The Hotdraw application example of this relationship is displayed in figure 5: Page Content. Figure 5 includes a key which demonstrates which page content refers to which component.

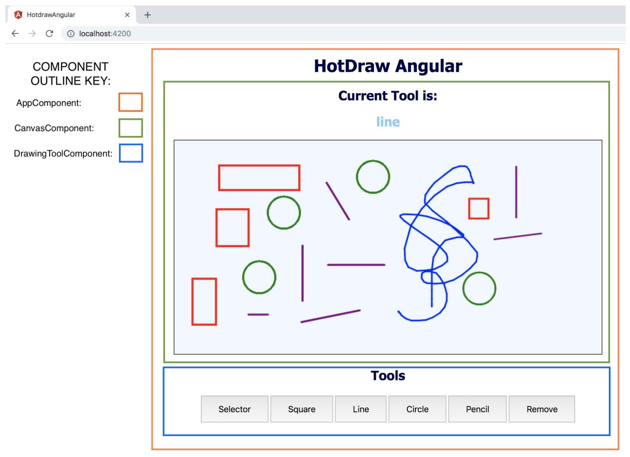

Figure 5: Page Content

Related Patterns: Component (2) as it makes use of Angular Components.

4 GRAPHIC CONTENT PATTERN

Context: An application can be interactive - where content can be created, modified and deleted in a dynamic manner by the user of the application. Examples of different kinds of graphical content to render are text, images, shapes and animations.

The application needs to know how to display the graphic content back to the user. Figure 6: Graphic Content demonstrates an example of a graphic shape, a triangle, being rendered as a part of the page content of an application called Simple-Canvas.

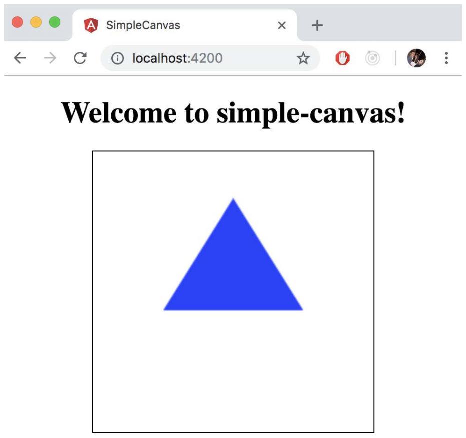

Figure 6: Graphic Content

Problem: How can a you implement functionality which allows an application user to dynamically generate and show their own graphic content?

Solution: Declare a `<canvas></canvas>` tag in the HTML file of the component to use the canvas element. You can use the HTML canvas element to display the graphics in a web browser. The `#canvas` descriptor allows the element to be referenced from the typescript file.

Graphics can be appended to and removed from the canvas element in the typescript file. The `@ViewChild` Angular property decorator configures a view to update when changes are made in the type script file. Declare a `HTMLCanvasElement` variable as a constant to construct a relationship between the `canvas` in the typescript file and the `<canvas>` tag in the HTML file. The canvas element can then be manipulated to display graphics determined by the user dynamically through logic in the typescript file.

Example: The following code demonstrates the declaration and use of the canvas element in the example displayed in Figure 6. The `style` determines the black border displayed around the width and height of the canvas element.

simple-canvas/src/app/app.component.html

```
<canvas #canvas style="border:1px solid #000;
width:300px; height:300px;"></canvas>
```

The code below demonstrates the `@ViewChild` being declared on the reference to the HTML canvas element to ensure changes are reflected as they occur. The `HTMLCanvasElement` is then declared to construct the relationship between the two canvas'.

simple-canvas/src/app/app.component.ts

```
@ViewChild('canvas') public canvas: ElementRef;
   ...
const canvasEl: HTMLCanvasElement =
                  this.canvas.nativeElement;
   ...
```

Related Patterns: Page Content (3) as the canvas is used to display graphical content on a page.

5 SHAPE PATTERN

Context: The Hotdraw application needs to be able to manipulate content on the canvas. Manipulation can include drawing, selecting, moving and deleting of the content.

Figure 7: Select Canvas Drawing demonstrates the selection of the square content on the canvas using the cursor.

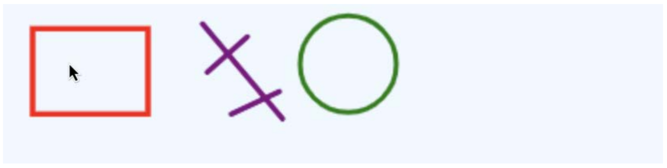

Figure 7: Select Canvas Drawing

Figure 8: Move Canvas Drawing demonstrates the moving of the square content from the left hand side to the right hand side on the canvas using the cursor.

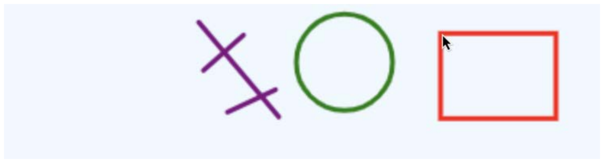

Figure 8: Move Canvas Drawing

Problem: How do you implement the functionality to enable the application user to manipulate dynamic graphical content?

Solution: Create and store content objects to enable dynamic manipulation of individual drawings on the canvas. Create an interface for the content objects to extend to avoid code duplication [10]. In the Hotdraw application, the `Shape` interface is extended by content objects. The relationship between the content objects and the Shape interface is demonstrated in Figure 9: Shape Interface.

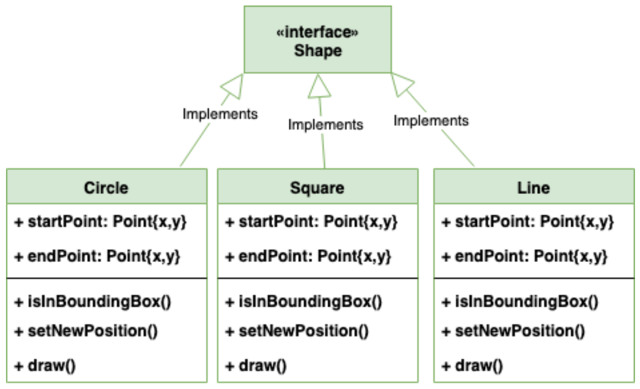

Figure 9: Shape Interface

These content objects have `startPoint` and `endPoint` variables and implement the `isInBoundingBox()`, `setNewPosition()`, and `draw()` methods.

You use the `startPoint` variable to determine the starting x,y position of the content object on the canvas. The `endPoint` is used to determine the size of the content object. This is discussed at greater length in Point (7).

`isInBoundingBox()` is a boolean method which returns true if the point the application user has clicked is inside this content object, otherwise it returns false.

`setNewPosition()` is a void method which updates the values of the `startPoint` and `endPoint` variables for this particular content object.

`draw()` uses the CanvasRenderingContext2D object discussed in Pen (6) to render this particular content object on the canvas to be displayed to the user.

Once you have created the content objects, store them in some form of data structure. The shared storage of all Shape content objects enables accessibility to one or all drawings on the canvas at once. You will be able to perform manipulations on the content objects by accessing them from the data structure and invoking the appropriate methods.

Example: The following code demonstrates the interface that all content objects, ie shapes, extend in the Hotdraw application.

hotdraw/src/app/shape.ts

```
export interface Shape {
  startPoint: Point;
  endPoint: Point

  isInBoundingBox(boundingBox: Point);
  setNewPosition(newPos: Point);
  draw(pen : CanvasRenderingContext2D);
}
```

The content objects are stored in an array of Shape objects. This allows dynamic accessibility for manipulation. The following code demonstrates an example of a `Circle` object which implements the Shape interface. As you can see, the `Circle` class implements the methods defined in the Shape interface to create a circle drawing that is manipulable.

hotdraw/src/app/shapes/circle.ts

```
export class Circle implements Shape {
    startPoint: Point
    endPoint: Point
    radius: number;

    constructor(startPoint: Point,
                          endPoint: Point) {
        this.startPoint = startPoint;
        this.endPoint = endPoint;
        this.radius = Math.sqrt(
        (Math.pow(
        (this.endPoint.x-this.startPoint.x), 2))
        + (Math.pow(
        (this.endPoint.x-this.startPoint.x), 2)))/2;
    }


    isInBoundingBox(boundingBox: Point) {
        return (boundingBox.x >=
            (this.startPoint.x this.radius)
        && boundingBox.y >=
            (this.startPoint.y - this.radius));
    }

    setNewPosition(newPoint: Point) {
        this.startPoint = newPoint;
    }
}
```

The following code demonstrates the creation and storage of a `Circle` object. An array of Shape objects is declared in the type script file. A `Circle` object is declared with two points as parameters and assigned to the shape variable. The shape variable is then added to the `shapes` array.

hotdraw/src/app/canvas.component.ts

```
shapes: Shape[];
    ...
var shape = new Circle(startPoint, endPoint);
    ...
shapes.push(shape);
```

The following code demonstrates the removal of all drawings on the canvas by removing all shape objects from the array of shapes and using the pen object to clear the canvas.

hotdraw/src/app/canvas.component.ts

```
clearCanvas(){
    this.shapes = [];
    this.pen.clearRect(0, 0,
        this.canvas.nativeElement.width,
            this.canvas.nativeElement.height);
}
```

The following code demonstrates iterating over the list and invoking each shape's `isInBoundingBox()` method with a predetermined `boundingBox` point parameter to determine if that shape is the one. The code below demonstrates moving the shape to a new location by calling the `setNewPosition()` method with the new point as a parameter.

hotdraw/src/app/tools/selector.ts

```
for (var i = 0; i < shapes.length; i++) {
    var selected = shapes[i];
        if (selected.isInBoundingBox(startPoint)) {
            selected.setNewPosition(endPoint);
            break;
        }
}
```

Related Patterns: Graphic Content (4) as it manipulates content that is on the canvas. Pen (6) as the shape utilizes the pen object declared to draw on the canvas.

6 PEN PATTERN

Context: The Hotdraw application needs to be able to use a tool to 'draw' the dynamic graphic content. Figure 6: Graphic Content demonstrates the use of a tool to draw and fill a blue triangle.

Problem: How to do you implement funcitonality to imitate the use of a pen on paper in an application?

Solution: Declare an instance of the CanvasRenderingContext2D object. The CanvasRenderingContext2D interface is a part of the Canvas API which provides the 2D rendering context for the drawing on the surface of the <canvas> element [1]. The CanvasRenderingContext2D is obtained by passing '2d' to the HTMLCanvasElement.getContext() method. You can use this object to issue drawing commands to the HTML canvas.

The object is able to draw a line or multiple lines by starting a path by calling the `beginPath()` method. The object can move to a point on the canvas by calling the `moveTo()` method with x and y co ordinates as parameters. Similarly, the object can define a line from this point to another by calling `lineTo()` method with x and y co ordinates as parameters. The object object can withdraw from the the canvas, in the way one would take the ten off of the paper, by calling `closePath()` method.

The object is also able to determine the style of the drawing by setting variables such as the outline and fill colour of the drawing. For the outline, the object can set the width of the outline by assigning a value to the `lineWidth` variable. The object can also set the colour of the outline by assigning ac value to the `strokeStyle` variable. The `stroke()` method then draws the path you have defined. The object can determine the fill colour of the content by assigning a value to the `fillStyle` variable, before calling the `fill()` method to actually fill the drawing.

The CanvasRenderingContext2D encompasses the ability to draw a number of different shapes on the canvas. Shapes such as square, rectable, circle and line, drawn on the canvas can be created not only statically, but also dynamically. The object can call the `strokeRect()` method to draw a square by taking the starting x, y point as two parameters as well as the width and height. The starting x, y point and the ending x, y point are used to determine the width and height. The x, y point relationship is discussed in detail in Point(7).

Example: The code below demonstrates an instance of the CanvasRenderingContext2D object being obtained and declared as the pen variable. At this point the pen variable has the ability to draw graphics on the canvas in the Simple-Canvas application. TheCanvasRenderingContext2D was declared in the same manner in the Hotdraw application.

simple-canvas/src/app/app.component.ts

```
private pen: CanvasRenderingContext2D;
...
this.pen = canvasEl.getContext('2d');
...
```

The following code demonstrates the pen variable invoking the `beginPath()`, `moveTo()`, `lineTo()`, and `closePath()` methods to draw a triangle on the canvas in the Simple-Canvas example.

simple-canvas/src/app/app.component.ts

```
// the triangle
 this.pen.beginPath();
 this.pen.moveTo(150, 25);
 this.pen.lineTo(75, 85);
 this.pen.lineTo(225, 85);
 this.pen.closePath();
```

The code below demonstrates setting of the outline and fill color variables of the triangle in the Simple-Canvas example.

simple-canvas/src/app/app.component.ts

```
// the outline
 this.pen.lineWidth = 5;
 this.pen.strokeStyle = '#666666';
 this.pen.stroke();
// the fill color
 this.pen.fillStyle = "#ffff00";
 this.pen.fill();
```

Figure 10: Graphic Content Pen demonstrates the graphics rendered on the canvas by the pen variable in the Simple-Canvas application. Note that it is slightly different to the example shown in Figure 6, this is because we have used the pen variable to add an outline and change the colour of the triangle.

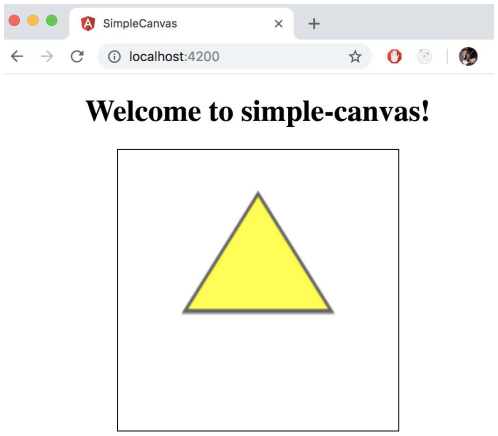

Figure 10: Graphic Content Pen

The Simple-Canvas application example demonstrates the use of the pen variable to create one simple shape on the canvas. However, the pen encompasses the ability to draw a number of different shapes on the canvas.

The Hotdraw application uses the pen variable to render different kinds of drawings at different points on the canvas such as squares, circles and lines. The following code demonstrates the pen variable invoking the setting the `strokeStyle` variable to the color red, before invoking the `strokeRect()` and `closePath()` methods to draw a square.

hotdraw/src/app/shapes/square.ts

```
draw(pen) {
    pen.strokeStyle = 'red';
   pen.strokeRect(this.startX, this.startY, this.endX -
    this.startX, this.endY - this.startY);
    pen.closePath();
}
```

The Figure 11: Hotdraw Canvas Pen demonstrates the result of the pen variable issuing drawing commands to the canvas and imitating the use of a pen in the Hotdraw application. The pen variable has been used to draw squares, circles and lines, as well as freehand drawings.

Related Patterns: Graphic Content (4) as the pen is used to draw content on the HTML canvas.

7 POINT PATTERN

Context: The Hotdraw application needs to be able to create, select, move and delete shapes from the HTML canvas based on an x,y coordinate. The HTML canvas is interactive, which means this is performed by executing a set of mouse events on the canvas, which

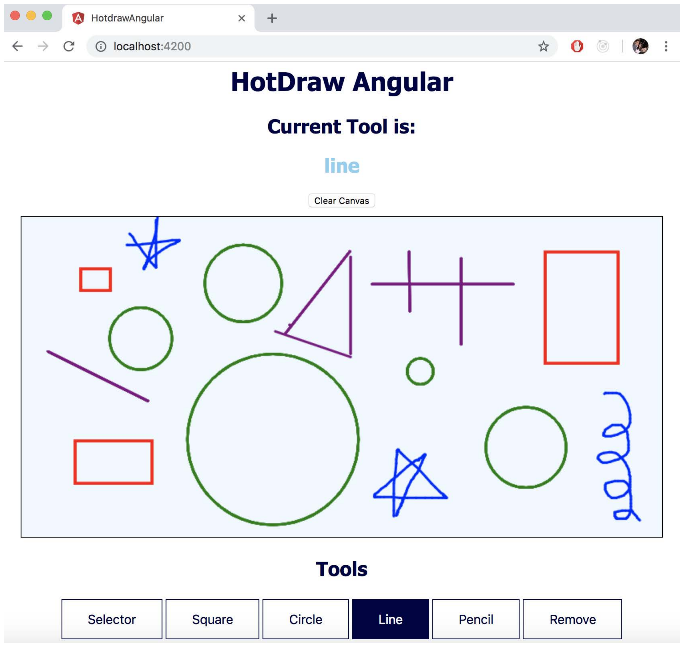

Figure 11: Hotdraw Canvas Pen

is discussed in more detail in Event (8).

Figure 12: Point X,Y Coordinate demonstrates a `Square` content object on the canvas. The x,y coordinates displayed are reference to the positions on the canvas where mouse events were responded to. The Hotdraw application needs to be able to respond to these events by identifying, recording and performing logic on the necessary set of x,y coordinates.

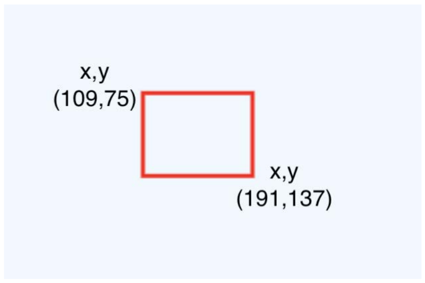

Figure 12: Point X,Y Coordinate

Problem: How do you record sets of x,y coordinates as points on a page in a web browser?

Solution: Declare a Point object which has an x variable and a y variable. Point objects are used to determine position and size of content objects on the canvas.

The `startPoint` variable of the content object is the point on the canvas where the content object begins. The `endPoint` variable of the content object is the point on the canvas where the content object ends. The difference between the two points is used to calculate the size of the content object.

You use the `startPoint, width` and `height` variables in the content object's `draw()` method to tell the pen where to draw on the canvas.

Example: The following code demonstrates the Point object. The `x` variable refers to the x coordinate and the `y` variable refers to the y coordinate.

hotdraw/src/app/point.ts

```
export class Point {
    x : number;
    y : number;
}
```

The following code demonstrates the relationship between the points and the content object by using the `Square` content object of the Hotdraw application as an example. The width is calculated by subtracting the `startPoint.x` from the `endPoint.x` and the height is calculated by subtracting the `startPoint.y` from the `endPoint.y`. `strokeRect()` method.

hotdraw/src/app/shapes/square.ts

```
export class Square implements Shape{
    startPoint: Point
    endPoint: Point
    width: number;
    height: number;

    constructor(startPoint: Point, endPoint: Point) {
        this.startPoint = startPoint;
        this.endPoint = endPoint;
        this.width = this.endPoint.x -
                this.startPoint.x;
        this.height = this.endPoint.y -
                this.startPoint.y;
    }

    ...

    draw(pen){
        ...
        pen.strokeRect(this.startPoint.x,
            this.startPoint.y, this.width,
                this.height);
        ...
    }
}
```

Figure 13: Point demonstrates the relationship between the `Point` object and the `Shape` content object on the canvas. You can apply the formula demonstrated above to the `startPoint` and `endPoint` variables to calculate the width and height variables: *width: 191 - 109 = 82 height: 137 -75 = 62.*

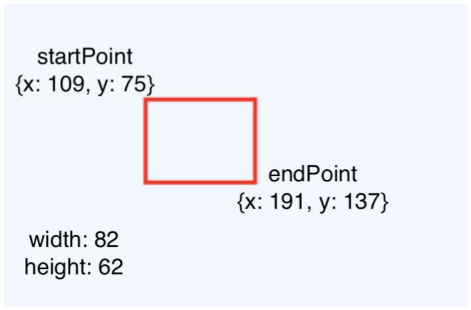

Figure 13: Point

Related Patterns: Shape(5), Pen(6) and Graphic Content(4), as it determines the x,y coordinates of shapes that the CanvasRenderingContext2D pen object draws on the canvas.

8 EVENT PATTERN

Context: An application needs to know how to respond to certain user input. A user of the Angular Hotdraw application inputs to the application through the mouse. The different uses of the mouse creates an event, such as mousedown, mousemove and mouseup.

Problem: How do you respond to user input in an application?

Solution: Define an event observer and subscriber relationship to observe and subscribe to certain events. The observer constantly listens for events. The observer must first be declared, before it is able to be subscribed to by the subscriber. The subscriber then determines the logic to be performed upon the firing of the event.

Example: The Hotdraw application example imports an observer and a subscriber from the Reactive Extensions for JavaScript (RxJS) to define this relationship. The code below demonstrates this import.

hotdraw/src/app/canvas.component.ts

```
import { Observable, Subscription } from 'rxjs';
```

The mousedown and mouseup events performed on the canvas are observed in the Hotdraw application. The code below demonstrates the the declaration of the observer, which listens out for the mousedown event. The `mouseDownObserver` specifically observes the `mousedown` event on the HTML canvas element defined in Graphic Content (4). The
`mouseDownObserver` is alerted each time the mousedown event is performed on the canvas.

hotdraw/src/app/canvas.component.ts

```
mouseDownObserver: Observable<any> =
Observable.fromEvent(canvasEl, 'mousedown');
```

The subscriber is subscribed to the
`mouseDownObserver`. The code below demonstrates the definition

of the mouseDownSubscriber, which includes the logic to be performed each time the mousedown event is performed on the canvas.

The logic assigns the x,y coordinates from the MouseEvent to a Point variable previously declared. The point on the canvas where the user pressed down, is saved to the startPoint variable that can be used outside of the subscription.

hotdraw/src/app/canvas.component.ts

```
mouseDownSubscriber : Subscription =
this.mouseDownObserver.
subscribe((res: MouseEvent) => {
      startPoint = {
        x: res.clientX - rect.left,
        y: res.clientY - rect.top
      };
    });
```

The Hotdraw application is also interested in observing the mouseup event. An observer is defined to listen specifically for the mouseup event. The following code demonstrates the declaration of the second observer in the Hotdraw application.

hotdraw/src/app/canvas.component.ts

```
mouseUpObserver: Observable<any> =
Observable.fromEvent(canvasEl, 'mouseup');
```

The subscriber determines what logic is to be performed each and every time the mouseup event is performed on the canvas. The logic assigns the x,y coordinates from the MouseEvent to a Point variable previously declared.

The coordinates are assigned to the endPoint variable instead of the startPoint variable. The drawingTool.performAction() method is called with the shapes array, startPoint and endPoint as parameters to perform further logic. Each time the mouseup event is performed on the canvas, the endPoint variable is updated and the drawingTool.performAction() method is called. The drawingTool.performAction() is discussed at a greater length in Tool (9).

hotdraw/src/app/canvas.component.ts

```
mouseUpSubscriber : Subscription =
this.mouseUpObserver.
subscribe((res: MouseEvent) => {
    endPoint = {
        x: res.clientX - rect.left,
        y: res.clientY - rect.top
    };
this.drawingTool.performAction
        (this.shapes, startPoint, endPoint);
});
```

Related Patterns: Graphic Content(4) and Point(7) as the observer/subscriber listens for mouse events on the canvas to determine points.

9 TOOL PATTERN

Context: The application needs to be able to distinguish which logic to perform in response to the application user’s mouse events.

Problem: How do you determine what action to perform on the canvas?

Solution: Declare a tool object for each of the actions, and have one dedicated variable to represent which tool is currently selected. Create an interface for tool objects to extend to save code duplication. In the Hotdraw application, the Tool interface is extended by tool objects. The relationship between the tool objects and the Tool interface is demonstrated in Figure 14: Tool Interface.

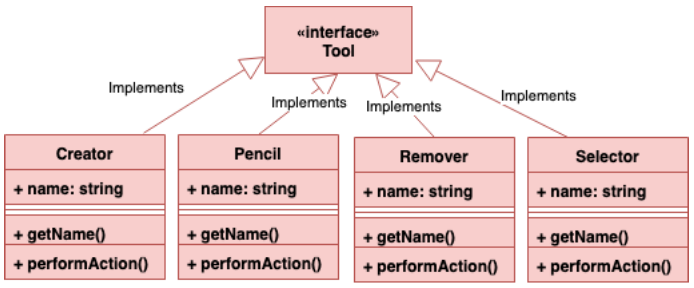

Figure 14: Tool Interface

These tool objects have a name variable and implement the getName() and performAction() methods.

You use the name variable to determine the name of the tool.

getName() returns the name of the tool object as a string.

performAction() performs the specified logic of the tool, for example, the ’Selector’ tool’s performAction() method selects the shape content object at a particular point (x,y coordinate) on the canvas.

Once you have created the tool objects, you will be able to perform manipulations on the shape content objects by invoking the performAction() method.

Example: The following code from the Hotdraw application demonstrates the Tool interface. The performAction() method takes an array of shape content objects and two point objects as parameters. The method adds, removes or modifies a shape content object in the array before returning it to the canvas component.

hotdraw/src/app/tool.ts

```
export interface Tool {
    name : string;

    getName();

    performAction(shapes: Shape[],
                 startPoint: Point,
                    endPoint: Point);
}
```

The following code demonstrates the Remover tool which removes the shape content object at a particular point (x,y coordinate) on the canvas.

The method searches the array for the shape content object at a point on the canvas selected by the user of the application. When a match is found, the shape content object is removed from the array. The modified array of shapes is returned to the canvas component.

hotdraw/src/app/tools/creator.ts

```
export class Remover implements Tool{
  name: string;

    constructor(name: string){
        this.name = name;
    }

     getName() {
        return this.name;
    }

    performAction(shapes: Shape[],
        startPoint: Point,
            endPoint: Point) {
        if (shapes.length == 0) { return; }

        for (var i = 0; i < shapes.length; i++) {
          var selected = shapes[i];
          if (selected.isInBoundingBox(startPoint)) {
            shapes.splice(i, 1);
            break;
          }
        }

        return shapes;
    }
}
```

A dedicated variable, drawingTool, which represents the currently selected tool, is first declared and then updated in the canvas component when a change in tool occurs. A change in tool is discussed in Service (10). The code below demonstrates the currently selected Tool object in the canvas component. The default drawingTool is the Selector.

hotdraw/src/app/canvas.component.ts

```
//default tool is selector;
drawingTool : Tool = new Selector();
```

The drawingTool variable is called each time a mouseup event occurs, discussed in Mouse Events(8), to perform the relevant logic. The canvas component displays the name of the currently selected tool above the canvas. Figure 15 : Currently Selected Tool Circle demonstrates that the current tool is Circle.

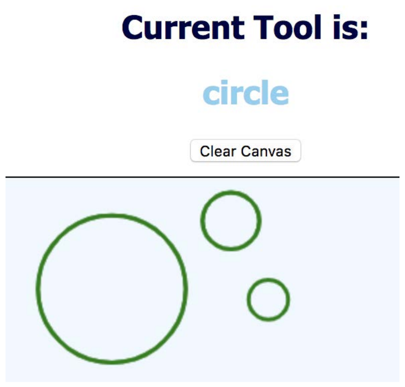

Figure 15: Currently Selected Tool Circle

The following code demonstrates the drawingTool.getName() method being invoked. The <h2> tag, discussed in Page Content(3), determines the heading. The heading is then referenced from the type script file in the same manner as the canvas element in Dynamic Graphic Content(4).

drawingTool is used to assign the value returned by the getName() method to the innerHTML value of the heading. The behaviour on the canvas should reflect the name of the tool displayed. For example, Figure 15 displays 'circle' as the currently selected tool. Thus, we would expect a circle to be created and displayed in response to a mousedown and mouseup event on the canvas.

hotdraw/src/app/canvas.component.html

```
<h2 class="curr-tool" #h2 id="showTool">
</h2>
```

hotdraw/src/app/canvas.component.ts

```
@ViewChild('h2') public header: ElementRef;
    ...
const selectedTool: HTMLHeadElement =
    this.header.nativeElement;
    ...
selectedTool.innerHTML =
    this.drawingTool.getName();
```

Related Patterns: Graphic Content(4) as the Tool object determines which actions are performed on the canvas.

10 SERVICE PATTERN

Context: In Angular, components do not share variables between each other. A user can select, manipulate and change a variable in a component. Often, a different component needs to know about the change to the variable. An application needs to know how to share this information.

Problem: How do you communicate the change in variable between components?

Solution: Create an instance of a service using the `@Injectable()` decorator. An injectable service can be injected in multiple components as it can be provided as a singleton instance [5]. The service is used to communicate information between components.

Create an instance of an EventEmitter object in the service. Create a method in the service which invokes the EventEmitter to emit custom events, and register handlers for those events in components.

Declare the service in all components which need to communicate information between each other by passing an instance of the service to the constructor of those components.

In the component where the variable is changed, declare an instance of a listener using the `@HostListener()` decorator. The listener declares a DOM event to listen for, and provides a handler method to run when that event occurs[6]. Implement the handler method to invoke the method declared in the service.

In the component waiting for the change in variable, use the service to subscribe to the custom event emitted by the service. Implement the logic to be performed once the event is received. This logic will be performed each time the event is emitted by the EventEmitter in the service.

Example: The user of the Angular Hotdraw application selects the tool to be used in the `DrawingToolComponent` discussed in Tool(9). The choice of currently selected tool needs to be propagated to the `CanvasComponent`, where the logic is performed on the canvas.

The `DrawingToolService` is used to communicate the change in the tool variable between the two components. The following code demonstrates the declaration of the service. The service has an instance of an EventEmitter. The EventEmitter emits the change in the `Tool` object when the `changeTool()` method is invoked.

hotdraw/src/app/drawingtool.service.ts

```
@Injectable()
export class DrawingToolService {

  @Output() toolChange: EventEmitter<Tool>
                          = new EventEmitter();

  changeTool(tool: Tool)
  {
    this.toolChange.emit(tool);
  }

}
```

An instance of the service is passed in to the constructor of both the `DrawingToolComponent` and the `CanvasComponent`. The following code demonstrates first the constructor of the `DrawingToolComponent`, followed by that of the `CanvasComponent`.

hotdraw/src/app/drawingtool.component.ts

```
constructor(private
        drawingToolService: DrawingToolService) {
            this.selectedTool = new Selector();
}
```

hotdraw/src/app/canvas.component.ts

```
 constructor(private
        drawingToolService: DrawingToolService) {
    this.shapes = [];
 }
```

The `changeTool()` method of the service is invoked in the `DrawingToolComponent` when the `@HostListener()` handles the 'click' DOM event. This is demonstrated in the following code.

hotdraw/src/app/drawingtool.component.ts

```
@HostListener('click')
click() {
    this.drawingToolService
            .changeTool(this.selectedTool);
    }
```

The `CanvasComponent` uses the service to subscribe to the event emitted in the `changeTool()` method. The following code demonstrates the logic that is performed each time the change of tool variable custom event is emitted. The drawingTool, Tool object, variable is updated to be the variable passed through the service. The selected tool in the HTML is updated. The event observables are reset and the `captureAllEvents()` method is invoked. The `captureAllEvents()` method captures the events being performed on the canvas, discussed in Event 8.

hotdraw/src/app/canvas.component.ts

```
this.drawingToolService.toolChange
                .subscribe(currentTool => {
    this.drawingTool = currentTool;
    selectedTool.innerHTML =
                 this.drawingTool.getName();
    //once we know the tool, reset observable
    //and capture the mouse events
    this.mouseDownSubscriber.unsubscribe();
    this.mouseUpSubscriber.unsubscribe();
    this.captureAllEvents(canvasEl);
});
```

Figures 2, 4 demonstrate the change in tool discussed in Component (2). Figure 3 demonstrates the different components. The heading 'Current Tool is:' demonstrates the result of using the `DrawingToolService`.

Related Patterns: Tool (9) as it enables the change of tool. Component (2) as it facilitates the communication between components.

ACKNOWLEDGEMENTS

I would like to thank my Shepherd Christian Kohls who gave constructive and valuable feedback. I would also like to thank my co-author and my focus group who also gave valuable feedback.

REFERENCES

[1] 2005-2019. Canvas API. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

[2] Seyed Hossein Ahmadpanah. 2015. What is Angular. js?! (2015).

[3] Kent Beck and Ralph Johnson. 1994. Patterns generate architectures. In *Object-Oriented Programming*, Mario Tokoro and Remo Pareschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–149.

[4] Kent Cunningham, Ward Beck. 1994. A CRC Description of HotDraw. http://c2.com/doc/crc/draw.html

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional. http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1

[6] Google. 2010-2018. Tutorial: Tour of Heroes. https://angular.io/tutorial.

[7] Takashi Iba and Taichi Isaku. 2012. Holistic Pattern-Mining Patterns. In *19th Pattern Languages of Programs conference*. Citeseer.

[8] Ralph E. Johnson. 1992. Documenting Frameworks Using Patterns. *SIGPLAN Not.* 27, 10 (Oct. 1992), 63–76. https://doi.org/10.1145/141937.141943

[9] Gerard Meszaros and Jim Doble. 1997. A pattern language for pattern writing. In *Proceedings of International Conference on Pattern languages of program design (1997)*, Vol. 131.

[10] James Noble. 2000. Arguments and results. *Comput. J.* 43, 6 (2000), 439–450.

[11] Alice Sasabe, Tomoki Kaneko, Kaho Takahashi, and Takashi Iba. 2016. Pattern mining patterns: a search for the seeds of patterns. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*. The Hillside Group, 12.

[12] Jenifer Tidwell. 2010. *Designing interfaces: Patterns for effective interaction design.* " O'Reilly Media, Inc.".

[13] Tim Wellhausen and Andreas Fiesser. 2012. How to write a pattern?: a rough guide for first-time pattern authors. In *Proceedings of the 16th European Conference on Pattern Languages of Programs*. ACM, 5.