Spring 2014

DART: DECLARATIVELY RENDERING 2D SKETCHES IN CANVAS

Jesus Rocha
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Computer Sciences Commons](#)

Recommended Citation

Rocha, Jesus, "DART: DECLARATIVELY RENDERING 2D SKETCHES IN CANVAS" (2014). *Master's Projects*. 369.

DOI: <https://doi.org/10.31979/etd.bd4c-tyfd>

https://scholarworks.sjsu.edu/etd_projects/369

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

DART:
DECLARATIVELY RENDERING 2D SKETCHES IN CANVAS

A Project
Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Jesus Rocha
Spring 2014

© 2014

Jesus Rocha

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

DART:
DECLARATIVELY RENDERING 2D SKETCHES IN CANVAS

by

Jesus Rocha

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2014

Dr. Cay Horstmann Department of Computer Science

Dr. Teng Moh Department of Computer Science

Professor Ronald Mak Department of Computer Science

ABSTRACT

DART: DECLARATIVELY RENDERING 2D SKETCHES IN CANVAS

Dart is a recent programming language for the web browser. It features structured programming through traditional class-based syntax, metaprogramming via mirror-based reflection and scriptable rendering of 2D shapes with the Canvas API. This project explores Dart as a potential host for a domain-specific language (DSL) to produce figures, like those found in entry-level programming books. The goal is for instructors or book authors to generate figures such as the ones that appear in a typical CS1/CS2 text book. Figures from the book are analyzed and a library for rendering them declaratively is developed. Dart was able to provide a convenient API even though not having advanced metaprogramming capability built-in.

ACKNOWLEDGEMENTS

I would like to give my sincere thanks and appreciation to Dr. Cay Horstmann for his continuous support, guidance and encouragement throughout the course of this project. My committee members, Dr. Teng Moh and Prof. Ronald Mak, thank you very much for your help, advice and support as well.

TABLE OF CONTENTS

1	Prior Work	8
1.1	Declarative vs. Programmatic Rendering	8
1.1.1	DOT, HTML & CSS	9
1.1.2	Processing	10
1.2	Domain-Specific Languages (DSL)	12
1.3	Diagrams for Algorithms and Data Structures	13
2	Hosts for Domain-Specific Languages	17
2.1	Metaprogramming in Ruby	17
2.1.1	Active Record in Rails	18
2.2	JavaFX	19
2.3	Scala	21
2.3.1	Operators	21
2.3.2	Call By Name	22
2.3.3	Implicit Conversions	23
2.3.4	Slick	23
2.4	Dart	24
2.4.1	Canvas	25
2.4.2	Reflection via Mirrors	27
2.4.3	Dynamic Programmability	29
2.5	Why Dart?	31
3	Design of DSL for Declarative Rendering in Dart	32
3.1	Project Requirements	32
3.2	Describing Diagrams with Code	32
3.3	Programmability	35
3.3.1	Metaprogramming	36
3.3.2	Object-Oriented Library	37
3.4	Layout Algorithms	39
4	API Specification & Manual	43
4.1	Factory Pattern	45
4.2	Factory Elements	45
5	Results and Conclusion	49
5.1	Big Java	49
5.2	Challenges	54
5.3	Conclusion	55
	References	56
	Appendix I: Catalog of Figures Drawn	58

LIST OF FIGURES

• Declarative DOT Example	8
• Programmatic Processing Example	11
• Sorting Algorithm Trace	14
• Balsa Visual	15
• Electronic Classroom	16
• JavaFX: Declarative and Programmatic	20
• Dart: Programmatic Drawing on Canvas	27
• Sample Figure to Draw	32
• Sample Figure using Dart	33
• Complex Figure Example	34
• Fibonacci Linked List	35
• Complex Figure using Dart	37
• Layout Algorithm Presets	39
• UML Design Diagram	44
• Base Figures Supported	45
• Survey of Figures in Big Java	49
• Appendix I: Catalog of Figures Drawn	58

1. Prior Work

This chapter introduces the notions of declarative programming, domain-specific languages and prior work in drawing diagrams; such as the one shown in Figure 1-1.

a) Declarative Code:

```
digraph structs {  
  node [shape=record];  
  struct1 [label="<f0> left|<f1> middle|<f2> right"];  
  struct2 [label="<f0> one|<f1> two"];  
  struct3 [label="hello\nworld |{ b |{c|<here> d|e}| f}| g | h"];  
  struct1:f1 -> struct2:f0;  
  struct1:f2 -> struct3:here;  
}
```

b) Diagram:

Figure 1-1: Declarative DOT [1] code is shown above (a). It is parsed by GraphViz to produce the diagram (b).

1.1. Declarative vs. Programmatic Rendering

Declarative rendering is a paradigm for defining the structure of elements within a diagram, document, form, or other visual artifact. Its purpose is served by removing the imperative need to compute the location of the elements. The declarative syntax expresses what the program should accomplish as opposed to how it will do so. Contrary to this is programmatic rendering where a

sequence of computations is provided by the author. Programmatic rendering gives complete control and allows factoring out repetitive tasks, but typically is at a lower level of abstraction.

1.1.1. DOT, HTML & CSS

GraphViz [2] is a graph visualization service that makes it simple to emit graph models in the DOT language. The DOT syntax contains declarative entities that need to be drawn. Those entities contain attributes that are recognized as properties within the context and applied to the configuration. With various attributes available for the graphs, nodes and edges [3] it is possible to generate many types of diagrams. GraphViz employs a collection algorithms that parse DOT to render visuals, like the one seen above. The syntax is at a higher level of abstraction which gets checked for validity at run time. DOT and GraphViz provide an integrated process rather than having to use a graphical interface to manually draw.

The DOT language is equivalent to graphs described over plain text. In DOT the graph is a data structure that contains elements, including other graphs. Edges connect the nodes within. The code in figure 1-1 shows an example of the DOT syntax which is purely declarative and a brief understanding for creating a graph is seen using DOT's declarative composition. In the syntax (a), on line 1 the special keyword `digraph` declares a directed-graph style; where edges point one way. It is followed by the graph name `structs`, and the braces enclose the data structure. Line 2 defines the default node along with any style changes that need to be applied. Three nodes are then defined in lines 3-5 that have a name or identifier, followed by properties, attributes or data within the brackets. In these 3 lines, it is noted that the value for the attribute `label` contains additional identification, encapsulation, and layout style. These are identified and interpreted as:

- **Tags:** serve as identifiers for the partitioned nodes
- **Pipe or Vertical Bar:** partitions the string data into a array-like element
- **New Line (\n):** allows the label in a string to continue on the next line
- **Brackets:** changes the layout from horizontal to vertical

Lines 6 and 7 conclude the node connections through the directed symbol and identifiers are used to provide precise connection for the edges.

Similar declarative languages exist in web applications, such as the hypertext markup language (HTML) and cascading style sheets (CSS) which are used to produce the structure and presentation of documents or pages. The syntax in HTML describes many components that makeup a document; for example, headings, paragraphs, spans, divisions, sections and articles belong to it. On the other hand, CSS defines the presentation and visual characteristics of the HTML. Both HTML and CSS are purely declarative like DOT.

1.1.2. Processing

Processing [4] is a programming language and environment initially created to teach computer programming fundamentals within a visual context. It is often used by graphics professionals to serve as the foundation for electronic sketchbooks. The environment is written in Java and programs are translated to it before being run. The nature of Processing is composed for programmatic rendering, as seen below in the steps taken to achieve the visual display.

Consider the following code fragement:

```
ArrayList<Square> squares;

void setup( ) {
  squares = new ArrayList<Square>( );
  for( int y=0; y < 5; y++ ) {
    for( int x=0; x < 5; x++ ) {
      if( x % 2 == 0 && y % 2 == 1 || x % 2 == 1 && y % 2 == 0 ) {
        squares.add( new Square( color( 40*x+50, 0, 0 ), x*100, y*100 ) );
      } else {
        squares.add( new Square( color( 0, 0, 40*x+50 ), x*100, y*100 ) );
      }
    }
  }
  size( 500, 500 );
  noLoop( );
}

void draw( ) {
  for( int i = squares.size()-1; i >= 0; i-- ) {
    Square sq = squares.get( i );
    sq.display( );
  }
}
```

```

class Square {
  color c;
  float x;
  float y;

  Square( color colr, float Xpos, float Ypos ) {
    c = colr;
    x = Xpos;
    y = Ypos;
  }

  void display( ) {
    stroke( 0 );
    fill( c );
    rect( x, y, 100, 100 );
  }
}

```

The result is shown in Figure 1-2.

Figure 1-2: The programmatic code (a) in Processing explicitly computes color and position for each square to produce this drawing (b)

The imperative procedures of how the visual is rendered is visible with Processing when compared to DOT. In declarative rendering, the established syntax is a definition for what needs to happen, whereas procedural programming uses syntax to explicitly describe how the program will accomplish its goal. In either case, both are domain-specific languages (DSLs) to produce visual artifacts.

1.2. Domain-Specific Languages (DSL)

Domain-specific languages (DSLs) [5] are languages tailored to a specific application domain, and offer gains in expressiveness and ease of use when compared with general-purpose programming languages in their domain application. The use of domain-specific languages is for the encapsulation of domain knowledge within a syntax that reads close to real sentences in the domain.

According to some experts, "domain-specific languages" are defined as a computer programming language of limited expressiveness focus on particular domain [6]. In this definition, the key elements are:

- **Programming language**; that is usable by humans to instruct computers.
- **Limited expressiveness**; carries bare minimum number of features needed to support domain.
- **Domain focus**; smaller ones make a limited scope language worthwhile
- **Language fluency**; should have a sense of feeling natural

The goal of a domain-specific language is targeted at a particular type of problem, rather than many. For example, the HyperText Markup Language (HTML) is used to represent a web page visually within a web browser. HTML solves the problem of representing a website through a visual interface, and the language itself is static and declarative. HTML is defined for limited graphical capability, for which its specific duty is to render information architecture, or page structure through specifications of semantics resembling a real document. The attributes associated with a page are widths, margins, paddings and borders. HTML also allows for design elements, such as color and background, to be declared and change the appearance of a page as

well. CSS, on the other hand, takes the visual presentation out of HTML and into a separate syntax. CSS is declarative for the properties that makeup the presentation of HTML elements.

The Structured Query Language (SQL) is a domain-specific language designed for data held in databases. Its syntax provides insight to what actions will take place on the data. The high levels of abstraction is suited for immediate queries, such as modification and retrieval of data, without the need to know how it is done. SQL provides a closer relationship to domain experts by allowing them to contribute directly.

In DSLs it is customary to distinguish between internal and external implementations. An internal domain-specific language resides inside a general purpose programming language. Examples are Rails (see section 2.1), JavaFX Script(see section 2.2) and Slick (see section 2.3). An internal DSL should have the feel of putting together whole sentences, rather than a sequence of disconnected commands. As a result, only a subset of the host language is used while items such as conditions, loops, and variables are avoided. External DSL's, on the other hand, are represented through an separate language, which need not be a programming language, like SQL, DOT, CSS and any derivative of the Standard Generalized Markup Language (SGML); such as HTML.

1.3. Diagrams for Algorithms and Data Structures

The imperative study of algorithms and its effects on data structures has been a passion for Robert Sedgewick in the last 35 years. He is a professor at Princeton University and has authored many books in this field. Sedgewick even went on to engineer systems that render diagrams beginning with the Brown Algorithms Simulator and Animator (BALSA) at Brown Univerity [7], to more recently for illustrations in his recent book; *Algorithms*[8]. His software is not generally available but here is a diagram from his book.

Visual traces of elementary sorting algorithms

Figure 1-3: Throughout Robert Sedgewick's book, *Algorithms*, visual illustrations are used to describe properties of algorithms at various time intervals. The internal state of a data structure is shown at different iterations.

Figure 1-4: BALSA exposes an algorithms detail with multiple dynamic views to illustrate the process of sorting data structures

The video *Sorting out Sorting* presented by Baecker [9] in 1981 is often described as the initial turning point for algorithm animation. Sedgewick took this idea from video to software in 1984 with the development of BALSA. BALSA is a conceptual framework developed to facilitate analyzing algorithms through visuals, like the figure above.

Figure 1-5: “Electronic Classroom” at Brown, a specially built lecture hall housing 60 powerful graphics-based workstations to teach the learning of algorithms using BALSA.

Ever since BALSA several other systems have been developed for algorithm animations. This survey [10],[11],[12],[13],[14] looks at additional work on interactive learning. Most of this work is not a language, library, or framework but each new algorithm must be explicitly programmed. This is time consuming.

In this work I am not concerned with animations, instead chose diagrams for printed books and web pages. However, techniques that I developed would be extensible to animations.

2. Hosts for Domain-Specific Languages

In this chapter, I examine several examples of internal DSLs and the language mechanisms to make their realization possible. With general-purpose programming languages, expressions for domain-specific content in a natural manner is limited. The same generic syntax of the language along with the custom semantic constructs provide the only viable solution in the attempt to express domain knowledge. A common structuring mechanism is a class library [15], which is a reusable implementation for certain tasks to be completed in a particular domain. The library encapsulates domain knowledge and provides abstractions at the semantic level, however it does not provide the right abstractions at the syntactic level.

In this section the syntactic mechanisms that allow more sophisticated expression of domain knowledge are examined through a number of choices. Metaprogramming in Ruby was used to produce domain-specific languages such as ActiveRecord for the Ruby on Rails framework. JavaFX Script leveraged both declarative and programmatic rendering to produce visuals. The strongly-typed Scala language has capabilities to create DSL's through implicit conversions and operator syntax.

2.1. Metaprogramming in Ruby

Metaprogramming is a paradigm where a program has knowledge towards other code as a first class data type. In traditional programs the code executes with data consumed as input then produced as output. Metaprogramming inverts this relationship where programs consume or produce code as data, thus when the generated program runs, it executes data which is code. A program with such capability is able to create and modify other code, including its own. An example of this is seen in Ruby [16] which is a dynamic programming language with metaprogramming that can be used to create domain-specific languages. As an example, consider the following code:

```
class Hash

  def method_missing( key, *args )
    text = key.to_s
```

```

    if text[ -1,1 ] == "="
      self[ text.chop.to_sym ] = args[0]
    else
      self[key]
    end

  end
end

```

which adds the dynamic methods into the Hash class at runtime. The `method_missing` definition added to the Hash class allows undefined methods to be caught when they are not available in instances of the class. The Hash class thus acquired dynamic retrieval or insertion using any named key as if the key were the name of a method. The following code would then be acceptable,

```

h = Hash.new

h.one = 1
h.two = 2

```

where the methods `one` and `two` are equivalent to:

```

h[:one] = 1
h[:two] = 2

```

Ruby's rich API for reflection, used inside of `method_missing`, examines information about itself. This allows hash values to be queried and set in order to emulate the dynamic allocation of getters and setters for any key in the Hash class.

2.1.1. Active Record in Rails

Metaprogramming facilitates writing domain-specific languages, for example the Ruby on Rails framework. Rails is an internal domain-specific language for web applications. The framework leverages an ActiveRecord base class to produce an object that wraps a row in a database table and encapsulates the database access while adding domain logic on the data as well. This is repeated for all other tables in the application without having to reconfigure. ActiveRecord is realized using `method_missing`, extensive reflection, dynamic evaluations and run-time extensions.

These objects infer the data as attributes dynamically from the table definition which they are linked to. Consider the following example:

```
client1 = Client.first
client1.name = "Joe"
client1.save
```

which shows interactions with a database table and uses dynamic evaluation to set a column, as an attribute, from a table. ActiveRecord queries the table metadata to automatically produce attribute getters and setters in this case. Additionally, the read and write methods are also inherited. ActiveRecord with metaprogramming is able to provide a consistent wrapper across all data domains. As a result database queries can be completely abstracted for all types of tables and programs work directly on the data instead without the need to worry about how the table access with queries work.

2.2. JavaFX

From 2000 - 2010, web-based applications emerged which are called Rich Internet Applications, or RIA. JavaFX belongs to this group, and features hardware-accelerated graphics for high-performance rendering. JavaFX is a framework from Sun which at its core runs on JavaFX Script, a declarative scripting language[17]. JavaFX Script mixes declarative syntax for layout with programmability of behavior. Here is an example for a declarative construction of a diagram (shown in Figure 2-3).

```
import java.lang.*;
import javafx.ui.*;
import javafx.ui.canvas.*;
Frame {
  centerOnScreen: true
  visible: true
  height: 500
  width: 500
  title: "Foreach demo..."
  onClose: operation() { System.exit( 0 ); }
  content: ScrollPane {
    background: white
    view: Canvas {
      content: bind foreach (i in [1..8], j in [1..8])
```

```

Rect {
  x: i*30
  y: j*30
  width: 30
  height: 30
  fill: Color { red: ( 100+i ) green: ( 100+j ) blue: ( 100+( i*j ) ) }
  stroke: white
  strokeWidth: 1
}
}
}

```

Figure 2-1: The JavaFX Script diagram was produced using declarative syntax mixed with programmability.

JavaFX Script has an integrated declarative structure with programmability, however the language lacks the dynamic nature of scripting languages. JavaFX Script is strongly-typed, i.e., each variable has a type and assigning it an incompatible value (or using in an expression with

incompatible values) is a compile-time error. Strongly-typed languages are compiled prior to execution, as opposed to scripting and dynamic languages where the code is evaluated at run-time. The advantage of strongly-typed is in the ability to catch errors at compile-time.

The JavaFX Script language was removed from JavaFX 2.0 in late 2011, and JavaFX is now merely a Java library. Thus declarative programming is no longer an option with JavaFX.

2.3. Scala

Scala [18] is a strongly-typed language that fuses the object-oriented and functional paradigm. It has many features that enable the construction of internal DSLs. These are shown in the examples below.

2.3.1. Operators

In Scala all arithmetic operators are methods that can be called without parenthesis and neither the method separator (i.e, period). The option to override and add new operators is also available where the operators are not bound to specific selection. As a result the language provides flexibility to write macros using custom methods names which can be any combination of symbols. Consider the following code fragement to add a new operator:

```
object Implicits {  
  implicit class RichInt( i: Int ) {  
    def +(x: Double) : Double = {  
      i + i*(x/100)  
    }  
  }  
}
```

Here the RichInt class was extended to add a new operator that treats its parameter, or right-side value, as a percentage. The two ways for writing the addition of 20% to an integer are:

```
( 10 ).+%( 20 )
```

and

```
10 +% 20
```

2.3.2. Call By Name

Next, in Scala's method definition the execution of the parameter, or block, can be deferred until a later time by a feature known as *call-by-name*. To use this feature the symbol (`=>`) is placed between the parameter name and type as seen here:

```
def repeat( n:Int, block: => Unit ) {  
  for( i < 1 to n ) {  
    block  
  }  
}
```

This allows the second parameter for the repeat method to be delayed and the first parameter was programmed to imposes control for the number of loops the second method is called. Currying can then be introduced by rewriting the function as:

```
def repeat( n:Int )( block: => Unit ) {  
  for( i < 1 to n ) {  
    block  
  }  
}
```

This gives the syntax expressiveness to emulate control abstractions, which allows Scala to define its own syntactic mechanisms which can then be used as:

```
repeat( 3 ) {  
  print( 'hello' )  
}
```

2.3.3. Implicit Conversions

In Scala, this can be rewritten again to accomplish a different syntax for the repeat method.

Using the operator overload feature along with call by name the new definition is:

```
object Implicit {  
  implicit class RichInt(i: Int) {  
    def *[A](f: => A): Unit = {  
      def loop(curr: Int): Unit = {  
        if(curr > 0) {  
          f  
          loop(curr - 1)  
        }  
      }  
      loop(i)  
    }  
  }  
}
```

```

    }
  }
}

```

Then code can then be written as:

```

2 * {
  print( 'hello' )
}

```

This would not work originally because integers do not have a method operator (*) for the block enclosed in brackets, however, with implicit conversions any methods, including operators, can be created for enriching existing libraries [19]. To achieve this the RichInt is an enriched type of Int, and is implicitly convert to that type while the *call-by-name* option is used to defer execution of the block. The method operator (*) delegates to a countdown procedure where at each iteration the block, code inside the curly braces, is executed once.

2.3.4. Slick

Slick is a Scala object-relational mapper (ORM) and domain-specific language (DSL) that provides an integrated and typesafe way for making queries to databases. The following example shows a query on the database using Scala syntax that treats the table like a collection.

```

class Coffees(tag: Tag) extends Table[(String, Double)](tag, "COFFEES") {
  def name = column[String]("COF_NAME", 0.PrimaryKey)
  def price = column[Double]("PRICE")
  def * = (name, price)
}

val coffees: TableQuery[Coffees] = TableQuery[Coffees]

coffees.filter(_.price < 10.0)

```

In the last line of code, the query is seen taking an object of type Coffee and returning a Query object. The Query object employs a standard collection method, filter, which implicitly takes in price, the less-than (<) symbol and the value (10.0) to produce a parse tree for the expression. If instead of 10.0 the supplied parameter was the string '10.0' then the call would fail because price is a Column[Double]. This allows errors to be detected early and not rely on the SQL equivalent,


```
"select * from COFFEES where PRICE < 10.0"
```

which can be vulnerable to typos and fail at run-time. The result with Slick is the ability to treat a table like a collection within the Scala syntax and not have SQL queries exposed.

In Slick the same query above can be defined as:

```
for { c <- coffees if c.price < 10.0 } yield c
```

which is a lightweight notation for expressing sequence of comprehensions. This does not look like a convenient alternative at first, however, when multiple tables are joined it uses Scala's for-comprehension to translate into methods, like filter, acting on the Query collection. As an example, consider adding a column to the Coffee table which points to an id from a Supplier table that includes a name of Column[String]. In order to query the same price range and show the name of supplier to each coffee this can be written as:

```
for {  
  c <- coffees  
  s <- suppliers if c.supId === s.id && c.price < 10.0  
} yield ( c.name, s.name )
```

In Scala's desugaring of for-comprehensions this would be translated to

```
coffees.flatMap(c => suppliers.filter(s => c.supId === s.id && c.price < 10.0).map((c.name, s.name)))
```

which chains one collection inside of the other. In the end you have another Query object produced that employs a parse tree for the expression to evaluate into an SQL expression. The Scala syntax leverages using for, if and yield in a natural manner for control abstractions in programming as opposed to select, from, and where which is native SQL syntax.

2.4. Dart

Dart [20] is a general-purpose programming language release for production in November 2012. Its application is for the development of scalable web applications.. There are many traditional features and characteristics equipped in its design; a class-based syntax, dynamic programmability, and a rich application-programming interface (APIs) suited for the modern

web. The language has an optional-typed mechanism where programs can be written without type annotations for prototyping and with them to detect incompatible types before run-time. In addition, programs in Dart are initialized similar to C++/Java by using declaring dependencies first and then code begins within the `main()` function.

2.4.1. Canvas

Canvas is an element for rendering graphics in HTML5. In any modern web client, it supplies an API to allow rendering of polygons, lines and text that can be previewed on the browser window. The following list are abstract entities that can be represented visually:

- rectangles
- polygons
- circles
- arcs
- text
- lines & closed paths
- bezier & quadratic curves

The API also features methods to apply and retrieve attribute information from the primitive drawings . Such properties include:

- fill color
- stroke color
- font size and name
- measure text width

The fill color is used for text and all closed path polygons, including rectangles and circles. The stroke color is for text, closed path polygons, arcs, lines and curves. Font size and name are set at the same time and adhere to the text. For rendering text, a function to measure width is available to determine the spanned width using either the default or a custom font and size.

Here is an example of a Dart program using the Canvas API in HTML5. Notice it is embedded into HTML and has a `main` function which is the runtime execution entry point.

```
<!DOCTYPE html>

<html>
  <head>
```

```

<meta charset="utf-8">
<title>Dart Canvas Example</title>
</head>
<body>

<canvas id="dart_canvas_example" height='500' width='500'>
  Your browser does not support Canvas
</canvas>

<script type="application/dart">

import 'dart:html';
import 'dart:math' as Math;

void main() {
  CanvasElement handle = querySelector("#dart_canvas_example");
  CanvasRenderingContext2D context = handle.getContext("2d");
  context.lineWidth = 5;

  //the nose polyline
  context.beginPath();
  context.moveTo(250, 275);
  context.lineTo( 200, 375 );
  context.lineTo( 250, 375 );
  context.stroke();

  //convert deg to radians
  var d2R = Math.PI/180;

  //the eyebrow: circular arcs
  context.beginPath();
  context.arc( 190, 220, 40, 210*d2R, 300*d2R );
  context.stroke();
  context.beginPath();
  context.arc( 310, 220, 40, 240*d2R, 330*d2R );
  context.stroke();

  //the eyeball circles
  context.beginPath();
  context.beginPath();
  context.arc( 190,250,40,0,2*Math.PI,false );
  context.stroke();
  context.beginPath();
  context.beginPath();
  context.arc( 310,250,40,0,2*Math.PI,false );
  context.stroke();

  //the lower lip: cubic Bezier curve
  context.moveTo(200, 400);

```

```

context.bezierCurveTo(225,475,275,475,300,400);

//the upper lip: quadratic Bezier curve
context.moveTo(200, 400);
context.quadraticCurveTo(250,450,300,400);
context.stroke();
}
</script>
</body>
</html>

```

The result is shown in Figure 2-4.

Figure 2-2: Dart with Canvas API code is used to produce the drawing

The Dart code above will run natively only on the Dartium browser, which is equipped with a run-time virtual-machine for Dart. For other browsers, the compiler *dart2js* produces a JavaScript equivalent script to allow running the code in all modern browsers.

2.4.2. Reflection via Mirrors

Dart features a mirror library which provides reflective capability for metaprogramming. The library allows introspection of definitions and structure for any class or instances during run-time. It also allows execution on any of the defined methods. Furthermore, the modification of a

class properties, similar to the dynamic nature of the Hash class example in Ruby, can be reproduced as seen in the next section.

According to Bracha[21], the Dart mirrors API is decoupled from the core language as a standalone library. It gets included only where needed as opposed to languages built from the ground up with it, like Ruby.

The code example below shows introspection on the class `Point` followed with invoking the same instance method between modification of a local variable. The `ClassMirror` instance was first reflected using the `reflectClass` method, and then the names were parsed to list all declared items. On the second half of the example, an initialized instance of `Point` is examined using an `InstanceMirror`. The instance of `Point` was reflected and then a method was invoked using the `invoke` method. This requires the first parameter to be a symbol representing the method to invoke followed by the parameters that are required for that method. In reflection, symbols become identifiers for the names of variables and methods. A `Symbol` can be created by using the literal notation, `#symbol`, or with the constructor, `new Symbol('symbol')`, that consumes a string. The method that makes changes to internal variables is called `setField`. It takes in one parameter that must be a symbol and the second is the new value.

Here is a complete example.

```
import 'dart:mirrors';

class Point {
  int x1, x2, y1, y2;
  Point( this.x1, this.x2, this.y1, this.y2);
  double slope() {
    return (x2 - x1)/(y2 - y1);
  }
}

void main() {
  // EXAMPLE 1
  ClassMirror cm = reflectClass(Point);
  print( cm );
  for (var m in cm.declarations.values){
    print(MirrorSystem.getName(m.simpleName));
  }
}
```

```
// EXAMPLE 2
Point p = new Point( 0, 10, 0, 10);
InstanceMirror m1 = reflect(p);
print( m1 );
print( m1.invoke(#slope, []).reflectee );
m1.setField( new Symbol('x1'), 2);
print( m1.invoke(#slope, []).reflectee );
}
```

The example produced this output.

```
ClassMirror on 'Point'
x1
x2
y1
y2
slope
Point

InstanceMirror on Instance of 'Point'
1.0
0.8
```

2.4.3. Dynamic Programmability

The flexibility of dynamically-typed languages relies on late binding where the references between various components of a program are resolved at runtime, as opposed to static languages in which such references are bound at compile-time. Dynamic programming in a class-based language is used to inspect and modify class properties at run-time.

Consider the following class:

```
import 'dart:mirrors'; // needed for MirrorSystem

class Node {
  Map<Symbol,dynamic> _setup = { };

  Node( );

  noSuchMethod( Invocation invocation ){
    String str = MirrorSystem.getName( invocation.memberName );
    if( str.endsWith( '=' ) ) {
      _setup[ new Symbol( str.substring(0, str.length-1 ) ) ] = invocation.positionalArguments[0];
    }else {
      return _setup[ new Symbol( str ) ];
    }
  }
}
```

```

    }
}

```

Similar to the Ruby example, the code above is a class that defines a mechanism to capture and process undefined methods. In Dart it is written as `noSuchMethod`. The class `Node` contains an internal `Map` collection to store and retrieve data made available only at runtime. The collection is referenced by keys as symbols. The data represented by the symbols was specified to be dynamic and covers from all primitive data types to other class instances.

The following code shows how the dynamic feature works.

```

var tmp = new Node( );
tmp.state = "disabled";

var node = new Node( );
node.padding = [ 10, 20, 30, 40 ];
node.box = { #x: 100, #y: 50, #width: 45, #height: 22 };
node.lineHeight = 25;
node.textAlign = "left";

node.child = tmp;

```

This populate the `_setup` map as:

```

{
  Symbol("padding"): [10, 20, 30, 40],
  Symbol("box"): {
    Symbol("x"): 100,
    Symbol("y"): 50,
    Symbol("width"): 45,
    Symbol("height"): 22
  },
  Symbol("lineHeight"): 25,
  Symbol("textAlign"): left,
  Symbol("child"): {
    Symbol("state"): disabled
  }
}

```

Dart's `noSuchMethod` is used within any class declaration to specify an unknown resolution will take place at run-time. An example of this is a class instance that invokes a method, property, variable or attribute that does not exist and instead of an error it attempts to resolve the issue. As

a result, any class using this feature would allow interceptions to attempt a second chance at a resolution. The `noSuchMethod` function takes in one parameter of type `Invocation` which contains three properties; `memberName`, `positionalArguments`, `namedArguments`. The dynamic addition of attributes and values above uses both the name and arguments. Position arguments are defined via an array matching every parameter and in this example to dynamically set a value the first position was used. Named parameters are parameters that contain a key referencing any data, this was not used in this example. The distinction between the two parameters is that one is a list (positional) while the other is a map (named).

2.5. Why Dart?

The motivation for using Dart is because it runs on modern web browsers and therefore inherits an API to render drawings into Canvas. It features object-oriented programming through class-based inheritance like Ruby and Scala. Similar to Ruby, it can also be tailored to be dynamic for run-time use while the Dart Editor is equipped with a type-checker that can be used to find errors early, help with code completion and communicate intent.

3. Design of DSL for Declarative Rendering in Dart

3.1. Project Requirements

The requirements for this project are to render diagrams as they appear in an entry level computer programming class using declarative layout for ease of specification and programmability for rendering repetitive tasks.

I use the Dart language with the Canvas drawing technology. To prove viability I declaratively render all diagrams found in Big Java[22].

3.2. Describing Figures with Code

In my approach to develop a DSL within Dart, I identified the domain to discover a reusable design pattern. My design idea was encapsulation for groups of diagram entities that share a common theme without specifying concrete classes for them. This has been defined in the abstract factory pattern which became the API for the library I strived for. In the diagrams I analyzed, consider the figure below from the textbook:

Figure 3-1: A sample figure from the book describing annotated code

Here I observed that this is one line of code with three annotated highlights. A DSL should express that in terms that make sense to an author or professor. I provide a note method that produces this visual effect. For example,

```
factory.note("Object", "System.out");
```

renders

Putting several of these together,

```
List data = [factory.note("Object","System.out"),
              ".",
              factory.note("Method","println"),
              "(",
              factory.note("Parameters","Hello, World!"),
              ")"];

factory.diagram( data ).draw();
```

produces

Figure 3-2: Analyses of figures with code produced this diagram with close resemblance to the original

As a second example, consider the more complex diagram from the textbook shown in Figure 3-3.

Figure 3-3: A complex drawing taken from the Big Java book

The diagram illustrates what happens during the for loop. The imperative approach I took for this illustration is as followed:

1. Define a progressive (iterative) data structure where each element is an iteration and each iteration contains a title
2. For each iteration, define a left-to-right rendering for two dynamic components
3. Define the first component as one variable (mapped to value) and the second as code
4. In the second component, apply color to specific text where inline html syntax exists

For the code to this complex diagram see Figure 3-5.

3.3. Programmability

When describing a domain-specific language it is desirable to reuse features of the host language. In Dart there are built-in constructs which facilitate the management of various collection types for data. The collections that exist are the Map, Set, List and Queue. The primary one I considered for this project was the list, which is identical to arrays and was used to represent the data. In Dart, lists can be heterogeneous; that is they can hold elements of different types. This is convenient when specifying drawing elements. In Dart, multi-dimensional arrays are lists of lists. The lists can be specified statically, such as in the preceding example, but they can also be generated programmatically for instance with the for-loop. This code below produced a sketch of a linked list:

```
<script type="application/dart">

  int fibonacci(int n) {
    if (n < 2) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
  }

  void main() {
    CanvasFactory factory = new CanvasFactory('drawable', {});
    List fibList = [];

    for( var x =0; x < 9; x++){
      fibList.add( fibonacci(x) );
    }

    factory.linkedlist( fibList )
      ..draw();
  }
</script>
```

Figure 3-4: Linked list computed programmatically

A new syntax feature is seen in the code above on the line that has `..draw()` which represents method cascading. This is similar to method chaining for invoking multiple method calls,

however with the cascading feature the method is always invoked on the same object, or the top parent. The semicolon in this mechanism is deferred until the last method and methods can exist on the same line with two periods(..₂) for separation, however it reads best with each method on its own line. This is convenient to use when multiple properties are dynamically attached to a class instance, such as in the method missing example which I rewrite as:

```
var node = new Node()  
  ..padding = [10, 20, 30, 40]  
  ..box = { #x:100, #y:50, #width: 45, #height: 22 }  
  ..lineHeight = 25  
  ..textAlign = "left"  
  ..child = tmp;
```

The second collection I used extensively were maps which, like lists, are able to represent diverse content. In the code from Figure 2-8, a Map is assigned dynamic values in the Node class and was used within the `noSuchMethod` handler to express dynamic properties for the elements that were drawn. In addition, I also used maps to represent key-value pairs for diagrams to illustrate a variable mapped to a value, as seen in Figure 3-3. For representing data the map keys were declared as strings. In the values I limited usage to strings, numbers, and dynamic pointers to nodes or containers.

3.3.1. Metaprogramming

For Dart's dynamic capability to work, I had to convert symbols into strings which required usage of the mirror library. Other than that the metaprogramming research I conducted in Dart was not used in my library since I did not find any real application in my library because the figures are represented statically and no dynamic evaluation was necessary during run-time. Furthermore, the mirror library was currently unstable [23] according to the API documentation from the website. As a result my focus was on the declarative rendering to reduce the imperative approach of having to calculate the positioning of elements.

In the `noSuchMethod` definition I used the `MirrorSystem` class to call the method `getName` which consumed a symbol via parameter to return a string equivalent. The method missing handler in

Dart has an Invocation parameter that allows the identification of the calling member, however it is represented as a symbol. Once this symbol is converted into a string I was then able to lookup the last character of the string and identify if it was a setter or getter call. The setter call has the same substring as the getter, the only difference is the included equals (=) at the end. Dart has an optional-typed system and now with dynamic properties reduced the need to predefine getters and setters for the computed layout implementation.

3.3.2. Object-Oriented Library

Without the need to use metaprogramming, instead I designed the library to follow an object-oriented approach. The imperative nature to reproduce all figures from the Big Java book was abstracted through the use of factory methods. The names of those methods are semantic towards the different types of figures found in the book. The following list of figures were identified:

- Array
- Code
- Linked List
- Note
- Object
- Queue
- Progressive, or Steps
- Stack
- Table
- Tree

All of these figure were managed through extensive usage of dynamic lists found in Dart. Each of these groups of figures is managed by a configuration file that facilitates rendering direction and positioning in addition to color schemes to use. The following illustrates the complex figure looked at earlier in the chapter. It uses a Progressive theme with two components, Stack and Code.

a) Main App

```

<script type="application/dart">
  Map canvas = {'coord_display': false,
                'bgColor':WHITE};

  CanvasFactory factory = new CanvasFactory(CANVAS_NAME, canvas);

  var page_title = ["Initialize counter", "Check condition",
                    "Execute loop body", "Update counter",
                    "Check condition again"];
  var i = [1,1,1,2,2];
  var code_sentences = [
    ["for (<span style='color: red;'>int i = 1</span>; i <= numberOfYears; i++)",
     "{",
     "  double interest = balance * rate / 100;",
     "  balance = balance + interest;",
     "}"],
    ["for (int i = 1; <span style='color: red;'>i <= numberOfYears</span>; i++)" ,
     "{",
     "  double interest = balance * rate / 100;",
     "  balance = balance + interest;",
     "}"],
    ["for (int i = 1; i <= numberOfYears; i++)" ,
     "{",
     "  <span style='color: red;'>double interest = balance * rate / 100;</span>",
     "  <span style='color: red;'>balance = balance + interest;</span>",
     "}"],
    ["for (int i = 1; i <= numberOfYears; <span style='color: red;'>i++</span>)" ,
     "{",
     "  double interest = balance * rate / 100;",
     "  balance = balance + interest;",
     "}"],
    ["for (int i = 1; <span style='color: red;'>i <= numberOfYears</span>; i++)" ,
     "{",
     "  double interest = balance * rate / 100;",
     "  balance = balance + interest;",
     "}"] ];

  List pages = [];
  for( int x = 0; x < page_title.length; x++)
  {
    pages.add( {"title": page_title[x],
               "content":
                 [factory.stacks( [{"i": "    "+i[x].toString()+"    "} ] ),
                 factory.codes(code_sentences[x]) ]
               } );
  }

```

```

    }

    factory.steps(pages)
      ..draw();
</script>

```

Figure 3-5: Library usage example shows how multiple figures are bundled to produce one that is more complex.

The abstract factory pattern was used to produce many figures and in each one applied the configuration for rendering. In the example above, a list called pages has hash maps that contain the title and content for the steps. In the construction of the pages, the hash key content contains a list of two entities, stack and code. The abstract factory class produces all the figures which were pre programmed according to declaratively specified properties and attributes determined from the Big Java book. Up next we look at how these declarative properties work.

3.4. Layout Algorithms

By definining the characteristics that make the figures in the book, the declarative approach is used to define the configuration and the data. Predefined configurations are resused via factory creation methods and external declarative maps. Once the draw method is called on any figure the layout algorithms underneath handle the logistics of figuring out how to produce the sketch, similar to GraphViz's algorithmic advantage. The configuration properties for each group of diagrams sharing the same theme dictates special requests for handling the drawing. An example for the Object drawing is seen below:

a) Object Configuration

```

Map object = {
  "type": "object",
  "presentation": {

```



```

"container": {
  "titleAlign": "center",
  "font": {
    "family": "Arial",
    "size": 15,
    "color": "black"
  },
  "lineHeight": 25,
  "direction": "vertical"
},
"nodes": {
  "padding": [ 5, 20, 5, 20],
  "margin": [10, 10, 10, 10],
  "textAlign": "left",
  "textVerti": "middle",
  "background": "#FD6769",
  "spacer": " = ",
  "borderRadius": [0, 0, 0, 0],
  "valueBackground": "white",
  "equalWidths": "true",
  "equalKeyValueWidths": "true",
  "textValueAlign": "center",
  "minValueWidth": 100
}
}
};

```

b) Output

Rectangle

x	=	5
y	=	10
width	=	30
height	=	30

Figure 3-6: The default configuration is shows for the Object figure. The configuration is applied to the container and each individual element.

The algorithm's role relied heavily on the configuration that was put in place for each type of drawing. An optional default configuration exists when the configuration file does not exist. The changing requirements for different figures prompted the addition of new entries in the configuration. In the beginning only a few properties were discovered and embedded within the library. Then as the number of properties grew for the project I had to apply separation of concern and decouple the declarative layout and presentation aspect of into individual files for each group.

In this project I used only three methods for rendering diagrams. The first was a vertical approach where each entity is drawn and the next one follows beneath. The next technique is the horizontal layout which similarly draws the next element where the previous one finished. The third rendering scheme was a complex algorithm applied to the tree diagrams to have them render in their natural form, as tree data structures. For the first two methods the margin property provided the only separation specified. For trees I used the margin property to provide proper spacing for the deepest level where the number of nodes for that level had exponentially grown.

In order for the layout algorithm to work a number of properties had to be defined to keep track of the position of the elements. These properties include:

- Container
 - X-Position
 - Y-Position
 - Width
 - Height
- Node
 - X-Position
 - Y-Position
 - Width
 - Height

The width of the container spans across all nodes drawn horizontally, or inherits from the node which has the maximum width in the collection. The same inverse principle applied to the height of the container. For each node the width and height were determined through measurements of the data after the setup of font size and font family was set. Canvas provided a method to set the

font definitions and then measure the total pixel width any string would encompass. With number types the same method can be used however it needed to be transformed into a string before Canvas would read its width. For data that contains a new collection, or container, this required a recursive iteration into the new collections to determine its boundaries before the node can establish its own. When the layout parser entered a new collection through a node a parent-child relationship was formed and linkage between the child and parent allowed surrounding awareness for the calculation of starting points, spanned widths and heights.

4. API Specification & Manual

The abstract factory pattern, as shown below, is the library which allows for many figures found in the Big Java book to be rendered. The data parameter for every builder method is a list. When the method is called to create a figure the final step to render to the screen is realized via the draw method. The CanvasFactory has only one dependant. This is the Canvas class which is a wrapper around the Canvas HTML element and ensures the drawing is visible on screen. With Canvas defined I was able to extract the CanvasRenderingContext2D which contained the methods to draw.

```
class CanvasFactory extends Canvas {
    List<DataStructure> layers;
    CanvasFactory( CanvasElement element ) : super( element ) {
        layers = new List<DataStructure>();
    }

    noSuchMethod( Invocation invocation ){
        //determines what needs to be drawn, imports configuration, and created DataStructure
        //saves drawing into layers list
        //returns the newly created DataStructure
    }

    void draw() {
        //loops through layers and issues draw
    }
}
```

The API above described the dynamic capability for the factory class to generate diagrams. Each diagram is represented by the DataStructure class which contains a list of nodes and has been configured to a particular digram. For nodes there are options for a tree in which case this class extends the Node class, or the base class can be used. Each node entity represent one unit to be drawn and within it other diagrams may exists. The DataStructure class at some point contained the methods necessary to add, edit, update and delete nodes, however it was discontinued in order to focusing on declarative rendering static data.

Figure 4-1: UML Diagram describes the abstract factory pattern

4.1. Factory Pattern

The requirement to build many diagrams from a book leverages a pattern which deferred the diagram creation until run time by looking up the invoked dynamic method to see if it had properties associated to it. This class is called the CanvasFactory and it realized the configurations to individual drawings by providing properties and characteristics to a new instance of the DataStructure class. This class is preconfigured with default properties in the event a full specification is not available for a dynamic method.

The DataStructure class is used to emulate a wrapper to a collection where methods can be defined to add and remove elements. Initially, it was used to facilitate working with data structures which are found in computer programming but later was exclusively used to contain static data and parse it to determine positioning of its elements. The collection class contains within a list of type Node or TreeNode. These nodes represents one unit, or element, to be layed out and drawn on the browser. The data allowed on these nodes are one array of mapped key-value pairs, numbers or strings, or also the primitive data types integer, string, or mapped key-value pairs.

4.2. Factory Elements

The following list of base figures can be drawn with the library.

Name: Array

Name: Code

```
for (int i = 1; i <= numberOfYears; i++)  
{  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

Name: Linked List

Name: Note

Object Method Parameters

```
System.out.println("Hello, World!")
```

Name: Objects

Name: Progressive

1 width = 10

2 width = 20

Name: Queue

D H R T

Name: Stack

luckyNumber = 13

Name: Table

Product_Code	Description	Price
116-064	Toaster	24.95
257-535	Hair Dryer	29.95
643-119	Car Vacuum	19.99

Name: Tree

5. Results and Conclusion

5.1. Big Java

To validate the implementation, I attempt to render all diagrams in Big Java.

With the CanvasFactory the following list of figure types was drawn as seen below. For each category type there is a total count found where a close relationship exists between at the high level, although the view can represent differently. Along with the list is also a survey of the chapters in Big Java where each of the different figures was identified from.

Type	Count
Array	5
Binary Search Tree	7
Diagram & Note	1
Hash Table	2
Linked List	4
Linked List & Progressive	4
List	3
Object	7
Object & Note	2
Other	29
Progressive & Stack	3
Progressive, Stack & Array	2
Progressive, Stack & Code	1
Progressive, Stack & Object	2
Progressive, Stack, Array & Note	2
Progressive, Stack, Code & Note	1
Regular Expression	1
Stack	2
Stack & Array	4
Stack & Note	1

Stack & Object	12
Stack, Array & Object	1
Stack, Object & Note	1
Table	7
Tree	9
Tree & Array	1
Progressive & Tree	3

Chapter	Page	Figure	Type	Rendered?
1	6	5	Other	No
1	13	7	Diagram & Note	Yes
1	16	10	Other	No
1	23	12	Other	No
2	39	1	Progressive & Stack	Yes
2	39	2	Stack & Note	Yes
2	40	3	Progressive & Stack	Partially
2	41	4	Object	No
2	42	5	Object	Partially
2	43	6	Object	No
2	44	7	Object	Partially
2	44	8	Object	No
2	45	9	Object	Partially
2	46	10	Other	No
2	46	11	Object	Yes
2	48	12	Other	No
2	54	17	Stack & Object	Yes
2	55	18	Stack & Object	Partially
2	55	19	Stack	Yes
2	55	20	Progressive & Stack	Partially
2	56	21	Progressive, Stack & Object	Yes
3	83	2	Stack, Object & Note	Partially

3	102	6	Stack & Object	Partially
4	138	1	Stack	No
4	151	3	Array	No
4	151	4	Array	No
5	173	1	Other	No
5	173	2	Other	No
5	179	3	Array	No
5	181	4	Stack & Object	No
5	197	6	Other	No
6	219	1	Progressive, Stack, Code & Note	Yes
6	221	2	Other	No
6	228	2	Other	No
6	229	4	Progressive, Stack & Code	Yes
7	277	1	Stack & Array	Yes
7	277	2	Stack & Array	Yes
7	281	3	Stack & Array	Partially
7	281	4	Stack, Array & Object	Partially
7	284	5	Progressive, Stack, Array & Note	Partially
7	285	6	Progressive, Stack, Array & Note	Partially
7	290	7	Stack & Object	Yes
7	293	8	Stack & Array	Partially
7	297	9	Hash Table	No
7	297	10	Hash Table	No
7	298	11	Hash Table	No
7	298	12	Hash Table	No
7	299	13	Progressive, Stack & Array	Partially
7	300	14	Progressive, Stack & Array	Partially
8	330	1	Other	No
8	331	2	Other	No
8	335	3	Progressive, Stack & Object	Partially

8	345	4	Stack & Object	Partially
9	376	2	Other	No
9	378	3	Stack & Object	Partially
9	380	4	Stack & Object	Partially
9	383	5	Other	No
10	421	2	Other	No
10	422	3	Other	No
10	423	4	Object & Note	Partially
10	426	5	Object & Note	Partially
10	433	6	Stack & Object	Partially
10	445	7	Other	No
10	447	8	Stack & Object	Yes
10	447	9	Stack & Object	Yes
10	448	10	Stack & Object	Partially
12	507	1	Other	No
12	515	6	Other	No
12	522	7	Other	No
12	533	8	Other	No
12	534	11	Other	No
13	570	2	Tree	No
13	579	3	Other	No
13	580	4	Tree	No
15	630	1	Linked List	Partially
15	632	2	Linked List	Partially
15	633	3	List	Partially
15	637	4	Linked List & Progressive	Partially
15	638	5	Linked List & Progressive	Partially
15	640	6	Linked List & Progressive	No
15	642	7	Linked List & Progressive	No
15	647	8	Linked List	No
15	647	9	List	Partially
15	648	10	Linked List	No

15	648	11	List	Yes
16	667	2	Other	No
16	671	4	Other	No
16	675	5	Hash Table	No
16	676	6	Hash Table	No
16	686	7	Binary Search Tree	Partially
16	687	8	Binary Search Tree	Partially
16	688	9	Binary Search Tree	Partially
16	689	10	Binary Search Tree	Partially
16	690	11	Binary Search Tree	Partially
16	691	12	Binary Search Tree	Partially
16	692	13	Binary Search Tree	Partially
16	698	14	Tree	Partially
16	699	15	Tree	Partially
16	700	16	Tree	Yes
16	700	17	Tree & Progressive	Partially
16	702	18	Tree & Progressive	Partially
16	704	19	Tree & Array	Partially
16	710	20	Tree & Progressive	Partially
18	743	2	Other	No
18	751	10	Other	No
19	778	1	Other	No
19	780	2	Array	No
19	785	3	Array	No
20	811	1	Other	No
22	866	1	Table	Yes
22	868	2	Table	No
22	868	3	Table	No
22	869	4	Table	No
22	870	5	Table	No
22	870	6	Table	No
23	916	4	Tree	No

23	921	5	Tree	No
23	922	6	Tree	No
23	931	8	Regular Expression	No
24	967	11	Table	No

Figure 5-1: Table of figures in Big Java. The total count of each type (a) and the book's entire list of figures found in each chapter (b). (61 no draw, 42 partially, 17 drawn)

5.2. Challenges

The major challenges faced in this project were first on the instability of Dart when it first came out. The language was still actively being developed and on several occasions the semantics would change and require refactoring. Also, not everything was known about the new language at first thus resources were hard to find. Dart became stable on November 2013 when a production release was announced.

Outside of Dart, there were figures in the Big Java book that were difficult to analyze and describe. In circumstances when a pattern was not detected it became increasingly difficult to program for declarative rendering and automatic layouts. The parsing of the various diagram I defined also needed to shared common methods to facilitate smaller code and this was not feasible when special drawings were attempted.

The idea of using declarative specification to draw the figures is one thing, but the actual implementation or workflow relied heavily on imperative programming to describe how and where a figure is to be drawn according to the configuration file.

Additional challenges existed in the computation and layering of components within each other. For most of the drawings it was not possible to define its position and dimensions and then move on, therefore the process of programming for graphical applications is an increasingly difficult field as the domain grows.

5.3. Conclusion

Dart's ability to host a domain-specific language is reflected on a library designed to encapsulate domain knowledge about various figures. The metaprogramming functionality was tested and seemed useful at first but then it was determined that no applications exists with sketches like they do with database access in Ruby and Scala. The library features faster rendering onto Canvas while being simple and user-friendly. The factory pattern in the library allows for a centralized API to produce many combinations of drawings without the need for imperative declaration to realize the same result. The figures from the book were analyzed and clustered into groups. Then the creation of base-elements was delegated to the factory class methods which pre-built the collection of entities to be drawn. The separate declarative configuration files for the library provided the set of rules for configuring, calculating layout, and rendering the sketches into the browser.

Out of the 120 diagrams surveyed from Big Java, my library was able to render correctly 15% , partially rendered were 35% and 50% were not rendered. The partially rendered diagrams had the correct visual representation when compared to one another, however were not identical. Some figures that were drawn had very limited resemblance due to the difficulty that it was to analyze and describe patterns found within. In my comments from the catalog section I reviewed every element to determine if the layouts were accurately implement when a pattern was found. For figures that did not have a pattern the only approach was imperative programming.

Dart's capability to be a domain-specific language was able to provide the ability to build a library on it. The dynamic programmability with method missing was used heavily and allowed for reduced boiler plate code with dynamic getters and setters. This project allowed me to discover computer graphics from an imperative approach to produce a mini-language through a library and achieve automated rendering.

REFERENCES

- [1] Emden Gansner, Eleftherios Koutsofios and Stephen North. [Drawing graphs with dot](#). Nov 2, 2010. <http://www.graphviz.org>
- [2] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. [Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools](#). AT&T Labs - Research.
- [3] [Node, Edge and Graph Attributes](#) <http://www.graphviz.org>
- [4] Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (Morgan Kaufmann Series in Computer Graphics)
- [5] Marjan Mernik, Jan Heering, and Anthony M. Sloane. [When and How to Develop Domain-Specific Languages](#) ACM Computing Surveys, Vol 37, No. 4, December 2005, pp. 316-344.
- [6] Martin Fowler and Rebecca Parsons. Domain Specific Languages. Addison-Wesley Professional, 2010
- [7] Brown, M.H., Sedgewick, R., A System for Algorithm Animation, Computer Graphics, July 1984, 177-186.
- [8] Robert Sedgewick and Kevin Wayne. Algorithms. Pearson Education. 2011
- [9] Baecker, R., & Sherman, D. (1981). Sorting Out Sorting [16 mm color sound film]. SIGGRAPH '81, Los Altos, CA, 1981
- [10] Justin Cappos, Patrick Homer, DsCats: Animating Data Structures for CS 2 and CS 3 Courses. University of Arizona
- [11] Jon L. Bentley and Brian W. Kernigham. A System for Algorithm Animation (Tutorial and User Manual). AT&T Bell Labs. 1991
- [12] Duane J. Jarc and Michael B. Feldman. A Empirical Study of Web-based Algorithm Animation Courseware in the Ada Data Structure Course. SIGAda '98. ACM

- [13] Sami Khuri. Designing Effective Algorithm Visualizations. Proceedings of the First Program Visualization Workshop, Porvoo, Finland, 2001, pp. 1-12.
- [14] David Scot Taylor, Andrei F. Lurie, Cay S. Horstmann, Menko B. Johnson, Sean K. Sharma and Edward C. Yin. Predictive vs. Passive Animation Learning Tools. ACM, SIGCSE 2009
- [15] Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects. OOPSLA '04, Oct 24-28 2004, Vancouver, British Columbia, Canada. 2004 ACM
- [16] David Flanagan and Yukihiro Matsumoto. The Ruby Programming Language. Jan 2008, Oreilly
- [17] [JavaFXScript](http://www2.hawaii.edu/~janst/465/lecture/JavaFXScript.htm) <http://www2.hawaii.edu/~janst/465/lecture/JavaFXScript.htm>
- [18] Martin Odersky et.al. [An overview of the scala programming language](#). Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 2006.
- [19] Cay Horstmann. Scala for the Impatient. Addison-Wesley 2012
- [20] <https://www.dartlang.org>. Dart.
- [21] Gilad Bracha and David Ungar. Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. OOPSLA '04, Oct. 24-28 2004, Vancouver, British Columbia, Canada. 2004 ACM
- [22] Cay Horstmann. Big Java. John Wiley & Sons, Inc. 2010
- [23] <https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart-mirrors> Accessed May 2014

Appendix I: Catalog of Figures Drawn

The figures from the Big Java book are surveyed. Code that was used to attempt to render such figures is shown, then a comparison from the book's figure to my library is briefly commented.

Chapter 1 - Figure 7

Code

```
List data = [factory.note("Object", "System.out"),
             ".",
             factory.note("Method", "println"),
             "(",
             factory.note("Parameters", "'Hello, World!'"),
             ")"];
Map config = {"nodes":{"padding":[5, 0],
                      "roundedCorner":3,
                      "font-family":"consolas"},
             "datastructure":{"margin":0}};

factory.diagram( data, config )
    ..draw();
```

Book

Library

Comments

This figure is a close approximation however the annotations were not rendered in bubbles. Instead they are rectangles with round edges. The map configuration was originally used to specify properties. This was later decoupled and made external.

Chapter 2 - Figure 1

Code

```
factory.steps([
    {"content": [new Stackframe( [{"width": "    10    "} ] ] },
    {"content": [new Stackframe( [{"width": "    20    "} ] ] },
    {"datastructure": {"orderedStyle": 2}}
])
..draw();
```

Book

Library

Comments

This was a precise layout. Minimal code accomplished proper layout. Only issue here was not specifying minimum width for the values therefore the code is seen with extra spaces for the width value.

Chapter 2 - Figure 2

Code

```
factory.stacks( [{
    "height": factory.note("No value has been assigned.", "    ")
} ])
..draw();
```

<p>Book</p>	<p>Library</p>
<p>Comments</p> <p>The annotation was reused from a previous example. No minimum width was specified for the value of height which is empty therefore needed to be emulated with spaces.</p>	

Chapter 2 - Figure 3

Code

```
factory.steps(
  [{"title": "Compute the value of the right-hand side",
    "content": [ new Stackframe([{"width":30}],
      new Note.withUnderlined( "width + 10;", "40",
        factory) ]}],
  [{"title": "Store the value in the variable",
    "content": [ new Stackframe([{"width":40}],
      new Note.with_("40", "", factory) ]}]
])
..draw();
```

Book

- 1 Compute the value of the right-hand side

- 2 Store the value in the variable

Library

- 1 Compute the value of the right-hand side

- 2 Store the value in the variable

Comments

An annotation on code through an underline is represented. Declarative syntax was not used to position the annotation in its proper place.

Chapter 2 - Figure 5

Code

```
var cls = factory.object("String",
    [{"data":"H e l l o ..."},
    {"length":"X"},
    {"toUpperCase":"X"}]);
var cls2= factory.object("String",
    [{"data":"M i s s i ..."},
    {"length":"X"},
    {"toUpperCase":"X"}]);

factory.diagram( [cls, cls2] )
    ..draw();
```

<p>Book</p>	<p>Library</p>
<p>Comments</p> <p>The very first attempt at rendering the object is seen here. The more recent and accurate approximation is found in Figure 3-6. The cogs inside the example can be added through a png without having to draw. I was able to write minimal amount of code to produce these figures although not exactly showing a fair approximation at the theme.</p>	

<p>Chapter 2 - Figure 7</p> <p>Code</p> <pre>factory.object("String", ["H e l l o ...", {"length":""}, {"toUpperCase":""}]) ..draw();</pre>	
<p>Book</p>	<p>Library</p>
<p>Comments</p> <p>Same as Chapter 2 - Figure 5.</p>	

Chapter 2 - Figure 9

Code

```
factory.object("String",
    ["M i s s i ...",
    {"length":""},
    {"toUpperCase":""},
    {"replace":""}])
..draw();
```

Book

Library

Comments

Same as Chapter 2 - Figure 5.

Chapter 2 - Figure 11

Code

```
var tmp1 = factory.object("Rectangle", [{ "x":5}, {"y":10},
    {"width":20}, {"height":30}]);
var tmp2 = factory.object("Rectangle", [{ "x":35}, {"y":30},
    {"width":20}, {"height":20}]);
var tmp3 = factory.object("Rectangle", [{ "x":45}, {"y":0},
    {"width":30}, {"height":20}]);

factory.diagram([ tmp1, tmp2, tmp3])
..draw();
```


Book

Library

Comments

This diagram initially did not come out with the best matching characteristics. It was later improved for the objects to match exactly as was presented in the book. Declarative code is smaller and rendering occurs automatically inside draw method.

Chapter 2 - Figure 17

Code

```
var tmp = factory.object("Rectangle",
    [{"x":5},
    {"y":10},
    {"width":20},
    {"height":30}]);

factory.diagram( [factory.stacks([{"box":tmp}]), tmp] )
    .draw();
```

Chapter 2 - Figure 18

Code

```

var cls = factory.object("Rectangle",
    [{"x":5},
    {"y":10},
    {"width":20},
    {"height":30}]);

factory.diagram( [factory.stacks([{"box":cls}, {"box2":cls}]), cls] )
    ..draw();

```

<p>Book</p>	<p>Library</p>
<p>Comments</p> <p>This figure is similar to the previous one in the limited width for the pointer, or value position. Also the objects are not representing complete properties and alignments.</p>	

<p>Chapter 2 - Figure 19</p> <p>Code</p> <pre>factory.stacks([{"luckyNumber":13}]) ..draw();</pre>	
<p>Book</p>	<p>Library</p>
<p>Comments</p> <p>This was a close approximation however did not specify the minimum width for the value. The code is very small and clear.</p>	

Chapter 2 - Figure 20

Code

```
Progressive steps = factory.steps( [
    {"content":[new Stackframe( [{"luckyNumber":13}] )] },
    {"content":[new Stackframe( [{"luckyNumber":13},
                                {"luckyNumber2":13}])] },
    {"content":[new Stackframe( [{"luckyNumber":13},
                                {"luckyNumber2":12}])] }],
    {"datastructure":{"orderedStyle":2}});

Stackframe stk = factory.stacks( [
    factory.codes(["int luckyNumber = 13;"]),
    factory.codes(["int luckyNumber2 = luckyNumber;"]),
    factory.codes(["luckyNumber2 = 12;"]) ],
    {"datastructure":{"ordered":true,
    "orderedStyle":2,
    "margin":0},
    "nodes":{"bgcolor":[255,255,255], "roundedCorner":0}});

factory.diagram( [stk, steps],
    {"nodes":{"bgcolor":"none", 'padding':0}})
    ..draw();
```

Book

```
int luckyNumber = 13; ①
int luckyNumber2 = luckyNumber; ②
luckyNumber2 = 12; ③
```

Library

```
① int luckyNumber = 13;
② int luckyNumber2 = luckyNumber;
③ luckyNumber2 = 12;
```

Comments

Very close approximation but did not have the pointer to show transition from one variable into the next. Code is not very clean, required additional inline configuration before it was decoupled into separate configuration files.

Chapter 2 - Figure 21

Code

```
ClassObject cls = factory.object("Rectangle",
    [{"x":5}, {"y":10}, {"width":20}, {"height":30}]);
ClassObject cls2 = new ClassObject.from( cls );
ClassObject cls3 = factory.object("Rectangle",
    [{"x":20}, {"y":35}, {"width":20}, {"height":30}]);

Progressive steps = factory.steps([
    {"content":[factory.stacks( [{"box":cls}] ), cls] },
    {"content":[factory.stacks( [{"box":cls2}, {"box2":cls2}] ), cls2 ] },
    {"content":[factory.stacks( [{"box":cls3}, {"box2":cls3}] ), cls3 ] },
    {"datastructure":{"orderedStyle":2}}]);

Diagram dgm = factory.diagram( [ steps],
    {"nodes":{"bgcolor":"none", 'padding':0}});
dgm.draw();
```

Book

Library

Comments

Context can be understood however does not have a strong illustration with the pointer mechanism. The code uses a class called `ClassObject` with its overloaded constructor to make an identical copy of the object.

Chapter 3 - Figure 2

Code

```
ClassObject ctr = factory.object("Counter",
    [{"value":"        "}] );
ClassObject ctr2 = new ClassObject.from( ctr );

Diagram dgm = factory.diagram(
    [factory.stacks([{"concertCounter":ctr}]), ctr]);
Diagram dgm2= factory.diagram(
    [factory.stacks([{"boardingCounter":ctr2}]), ctr2]);

Diagram wrapper = factory.diagram( [dgm, dgm2],
    {"datastructure":{"vertical":true}});
wrapper.draw();
```

Book

Library

Comments

The default flow for the diagram implementation is to render left-to-right but in this example it was overwritten to render vertically. The annotation was not implemented for the object.

Chapter 3 - Figure 6

Code

```
ClassObject obj = factory.object("Bank Account",
    [{"balance":1000}]);
```

```
Diagram dgm = factory.diagram(
    [factory.stacks([{"momsSavings":obj},
                    {"this":obj},
                    {"amount":300}], obj)];
dgm.draw();
```

Book

Library

Comments

Close approximation for the layout. The divisor was not configured for the declarative code.

Chapter 6 - Figure 1

Code

```
var page_title = ["Initialize counter", "Check condition",
                  "Execute loop body", "Update counter",
                  "Check condition again"];
var i = [1,1,1,2,2];
var code_sentences = [
    ["for (<span style='color: red;'>int i = 1</span> i <= numberOfYears; i++)",
     "{",
     "  double interest = balance * rate / 100;",
     "  balance = balance + interest;",
     "}"],
    ["for (int i = 1; <span style='color: red;'>i <= numberOfYears</span>; i++)",
     "{",
     "  double interest = balance * rate / 100;",
     "  balance = balance + interest;",
     "}"],
    ["for (int i = 1; i <= numberOfYears; i++)",
     "{",
```



```

    " <span style='color: red;'>double interest = balance * rate / 100;</span>",
    " <span style='color: red;'>balance = balance + interest;</span>",
    "}],
["for (int i = 1; i <= numberOfYears; <span style='color: red;'>i++</span>)" ,
"{",
"  double interest = balance * rate / 100;",
"  balance = balance + interest;",
"}"],
["for (int i = 1; <span style='color: red;'>i <= numberOfYears</span>; i++)" ,
"{",
"  double interest = balance * rate / 100;",
"  balance = balance + interest;",
"}"]];

List pages = [];
for( int x = 0; x < page_title.length; x++)
{
    pages.add( {"title": page_title[x],
                "content":
                [factory.stacks( [{"i": "    "+i[x].toString()+"    "}] ),
                factory.codes(code_sentences[x]) ]
    });
}
var steps = factory.steps(pages);
steps.draw();

```

Book

Library

Comments

Close approximation for the layout and design. The three dots separating the diagram was not specified in the declarative code and inline text highlighting is applied for the code component on the right side.

Chapter 6 - Figure 4

Code

```
var code = [];
code.add( [
    "for (<span style='color:red;'>int i = 1</span>; i <= numberOfYears; i++)" ,
    "{",
    "    double interest = balance * rate / 100;",
    "    balance = balance + interest;",
    "}" ] );
```

```

code.add( [
  "for (int i = 1; <span style='color:red;'>i <= numberOfYears</span>; i++)" ,
  "{",
  "  double interest = balance * rate / 100;",
  "  balance = balance + interest;",
  "}" ] );
code.add( [
  "for (int i = 1; i <= numberOfYears; i++)" ,
  "{",
  "  <span style='color:red;'>double interest = balance * rate / 100;</span>",
  "  <span style='color:red;'>balance = balance + interest;</span>",
  "}" ] );
code.add( [
  "for (int i = 1; i <= numberOfYears; <span style='color:red;'>i++</span>)" ,
  "{",
  "  double interest = balance * rate / 100;",
  "  balance = balance + interest;",
  "}" ] );
code.add( [
  "for (int i = 1; <span style='color:red;'>i <= numberOfYears</span>; i++)" ,
  "{",
  "  double interest = balance * rate / 100;",
  "  balance = balance + interest;",
  "}" ] );

var iterationSteps = [
  {"title": "Initialize counter",
   "content": [
     factory.stacks( [{"i":1}] ),
     factory.codes( code[0] ) ]},
  {"title": "Check condition", "content": [
     factory.stacks( [{"i":1}] ),
     factory.codes( code[1] ) ]},
  {"title": "Execute loop body", "content": [
     factory.stacks( [{"i":1}] ),
     factory.codes( code[2] ) ]},
  {"title": "Update counter", "content": [
     factory.stacks( [{"i":2}] ),
     factory.codes( code[3] ) ]},
  {"title": "Check condition again", "content": [
     factory.stacks( [{"i":2}] ),
     factory.codes( code[4] ) ]},

  ];

factory.steps( iterationSteps )

```

```
..draw();
```

Book

1 Initialize counter	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>
2 Check condition	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>
3 Execute loop body	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>
4 Update counter	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>
5 Check condition again	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>

Library

1 Initialize counter	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>
2 Check condition	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>
3 Execute loop body	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>
4 Update counter	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>
5 Check condition again	<pre>for (int i = 1; i <= numberOfYears; i++) { double interest = balance * rate / 100; balance = balance + interest; }</pre>

Comments

Close approximation for the layout and view. The color automatically increased gradient and resulted in a significant mismatch.

Chapter 7 - Figure 1

Code

```
List zeros = new List(10);  
for(var x=0; x < zeros.length; x++ )  
    zeros[x] = " 0  ";
```

```
ClassObject array = factory.object("double[]", zeros);  
  
factory.diagram([factory.stacks( [{"values":array}]), array])  
  ..draw();
```

Book

Library

Comments

Close approximation for the layout. The node properties for the key-value entry did not have a minimum width value and the values inside the objects did not specify full width span.

Chapter 7 - Figure 2

Code

```
List arrayData = new List(10);
for(var x=0; x < arrayData.length; x++ )
{
    if( x == 2 )
        arrayData[x] = {"[${x}]" : "29.95"};
    else
        arrayData[x] = {"[${x}]" : "0"};
}
ClassObject array = factory.object("double[]", arrayData);

factory.diagram([factory.stacks( [{"values":array}]), array])
    ..draw();
```

Book

Library

Comments

Close approximation for the layout. The key-value separator was later removed to be offered optionally.

Chapter 7 - Figure 3

Code

```
ClassObject obj1 = factory.object("int[ ]", [
    "    ",
    "    ",
    "    ",
    "    "]);
ClassObject obj2 = factory.object("int[ ]", [
    "    ",
    "    ",
    "    ",
    "    "]);

factory.diagram([factory.diagram([
    factory.stacks([{"accountNumbers":obj1}],
    obj1),
    factory.diagram([
    factory.stacks([{"balances":obj2}],
    obj2),
    ]).draw();
```

Book

Library

Comments

The diagram feels positioned properly. The important part missing here is the rectangle comparing two object properties.

Chapter 7 - Figure 4

Code

```
ClassObject bank2 = factory.object("BankAccount", [
    {"accountNumber":"    "},
    {"balance":"    "});

List arrayData = new List(10);
for(var x=0; x < arrayData.length; x++ )
{
    if( x == 3 )
        arrayData[x] = {"_ptr":bank2};
    else
        arrayData[x] = "    ";
}
ClassObject bank = factory.object("BankAccount[]", arrayData);

Diagram page281 = factory.diagram([factory.stacks( [{"accounts":bank}]),
    bank, bank2]).draw();
```

Book

Library

Comments

Close approximation for the layout. The middle object's values were not specific to one-hundred percent width therefore the arrow overlaps through it.

Chapter 7 - Figure 5

Code

```
ClassObject arrayList = factory.object( "ArrayList<String>",
                                         ['"Emily"', '"Bob"']);
ClassObject arrayList2 = factory.object( "ArrayList<String>",
                                         ['"Emily"', '"Bob"', '"Cindy"']);
Progressive page284 = factory.steps(
    [
        {"title":"Before add", "content": [
            factory.stacks(["names": arrayList ], ), arrayList]
        },
        {"title":"After add", "content": [
            factory.stacks(["names": arrayList2 ], ), arrayList2]
        }
    ], {"datastructure":{"vertical":false, "equalWidths":false}} )
    ..draw();
```

Book

Library

Comments

Close approximation for the layout. Properties not implemented for minimum width of values and neither were the annotations included.

Chapter 7 - Figure 6

Code

```
ClassObject arrayList = factory.object( "ArrayList<String>", ["\"Emily\\",
                                                                "\"Bob\\", "\"Carolyn\\""]);
ClassObject arrayList2 = factory.object( "ArrayList<String>", ["\"Emily\\",
                                                                "<span style='color:red;'>\"Ann\\"</span>",
```

```

        "\Bob\"", "\Carolyn\"]);
ClassObject arrayList3 = factory.object( "ArrayList<String>", ["\Emily\"",
        "\Bob\"", "\Carolyn\"]);
factory.steps(
    [
        {"title":"Before add", "content": [
            new Stackframe(["names": arrayList ], ), arrayList]
        },
        {"title":"Afternames.add(1, \Ann\)", "content": [
            new Stackframe(["names": arrayList2 ], ), arrayList2]
        },
        {"title":"After names.remove(1)", "content": [
            new Stackframe(["names": arrayList3 ], ), arrayList3]
        }
    ]);
..draw();

```

Book

Library

Comments

Close approximation for the layout. Missing some node properties, centered text, and the annotated highlights.

Chapter 7 - Figure 7

Code

```
ClassObject obj = factory.object( "Double", [{"value":"29.95"}]);
```

```
factory.diagram(  
  [  
    factory.stack(["d:obj"]), obj  
  ] )  
  ..draw();
```

Comments

Close approximation for the layout. Node is missing minimum height attribute.

Chapter 7 - Figure 8

Code

```
ClassObject obj1 = factory.object("double[ ]", [ "          ", "          ",
            "          ", "          ", "          ", "          ",
            "          ", "          "]);
factory.diagram([factory.diagram([factory.stacks([{"values":obj1},
            {"valueSize":6}]), obj1)]) )
..draw();
```

Chapter 7 - Figure 13

Code

```
ClassObject double1 = factory.object( "double[]", ["32", "54",
                                                    "67.5", "29",
                                                    "35", "47.5"]);

ClassObject double2 = new ClassObject.from( double1);
ClassObject double3 = new ClassObject.from( double1);

factory.steps(
[
    {"title":"After the assignment prices = values", "content": [
        factory.stacks([{"values":double1}, {"prices": double1}], double1
    ]},
    {"title":"After calling Arrays.copyOf", "content": [
        factory.diagram([factory.diagram([
            factory.stacks([{"values":double2}], double2)],
        factory.diagram([
            factory.stacks([{"prices":double3}], double3)]),
```

```

        {"datastructure":{"vertical":true}})
    ]}
    ], {"datastructure": {"vertical":false, "equalWidths":false}} )
    ..draw();

```

Book

Library

Comments

Close approximation for the layout. Node properties missing.

Chapter 7 - Figure 14

Code

```

ClassObject double1 = factory.object( "double[]",
                                     [ " ", " ", " ", " ", " ", " ", " ", " ", " " ]);
ClassObject double2 = new ClassObject.from( double1);
ClassObject double3 = new ClassObject.from( double1);
ClassObject double4 = new ClassObject.from( double1);

factory.steps(
    [
        {"title":"Move elements to a larger array", "content": [
            factory.diagram([factory.diagram([
                factory.stacks([{"values":double1}], double1)),
                factory.diagram(double4)], {"datastructure":{"vertical":true}})

```

```

    }},
    {"title":"Store the reference to the larger array in values",
     "content": [
       factory.diagram([factory.diagram([
         factory.stacks([{"values":double3}]), double2]),
         factory.diagram([double3]), {"datastructure":{"vertical":true}})
     ]}
  ], {"datastructure": {"vertical":false, "equalWidths":false}} )
  ..draw();

```

Book

Library

Comments

Layout is not accurate. Missing node properties to align components to the right.

Chapter 8 - Figure 3

Code

```

ClassObject row1 = factory.object("BankAccount", [{"balance":2500}]);
ClassObject row2 = new ClassObject.from( row1 );
ClassObject row3 = factory.object("BankAccount", [{"balance":2000}]);
ClassObject row4 = new ClassObject.from( row3 );

factory.steps(
  [
    {"title":"Before method call", "content": [
      factory.stacks([{"harrysChecking": row1,
        {"savingsBalance":1000}], row1]

```

```

    },
    {"title":"Initializing method parameters", "content": [
        factory.stacks([{"harrysChecking": row2},
            {"savingsBalance":1000},
            {"this": row2},
            {"amount":500},
            {"otherBalance":1000}]), row2 ]
    },
    {"title":"About to return to the caller", "content": [
        factory.stacks([{"harrysChecking": row3},
            {"savingsBalance":1000},
            {"this": row3},
            {"amount":500},
            {"otherBalance":1000}]), row3 ]
    },
    {"title":"After method call", "content": [
        factory.stacks([{"harrysChecking": row4},
            {"savingsBalance":1000}]), row4 ]
    }
] )
..draw();

```


Book

Library

Comments

Close approximation for the layout. Missing horizontal spacer and annotated highlight.

Chapter 8 - Page 338

Code

```

ClassObject obj1 = factory.object("BankAccount", [{"balance":2500}]);
ClassObject obj2 = factory.object("BankAccount", [{"balance":1000}]);
ClassObject obj3 = factory.object("BankAccount", [{"balance":1500}]);
Stackframe tmp = factory.stacks( [obj1, obj2, obj3]);
factory.diagram([factory.stacks([{"harrysChecking":obj1},
                                {"savingsAccount":obj2},
                                {"this":obj1},
                                {"amount":500},
                                {"otherAccount":obj3}]), tmp ] )

..draw();

```

Book

Library

Comments

Close approximation for the layout. Missing some node properties, horizontal spacer and the annotated highlight.

Chapter 8 - Figure 4

Code

```

ClassObject obj1 = factory.object("BankAccount",
                                   [{"balance":1000},
                                   {"accountNemer":1001}]);
ClassObject obj2 = factory.object("BankAccount",
                                   [{"balance":8000},
                                   {"accountNemer":1002}]);

```

```

ClassObject obj3 = factory.object("BankAccount",
                                [{"balance":0},
                                {"accountNumber":1003}]);
factory.diagram([factory.diagram([
    factory.stacks([
        {"collegeFund":obj1}]), obj1]),
    factory.diagram([
        factory.stacks([
            {"momsSavings":obj2}]), obj2]),
    factory.diagram([
        factory.stacks([
            {"harrysChecking":obj3}]), obj3]),
    factory.diagram([
        factory.stacks([
            {"BankAccount.lastAssignedNumber":1003}])])
    ], {"datastructure":{"vertical":true}} );

..draw();

```

Book

Library

Comments

Close approximation for the layout. Missing some node properties and the annotated highlights.

Chapter 9 - Figure 3

Code

```
ClassObject obj2 = factory.object("BankAccount", [{"balance":10000}]);
factory.diagram([factory.diagram([
    factory.stacks([{"account":obj2}]), obj2]),
    factory.diagram([factory.stacks([{"meas":obj2}]), obj2]),
    {"datastructure":{"vertical":true}} )
    ..draw();
```

Book

Library

Comments

Close approximation for the layout. Missing some node properties and the annotated highlights.

Chapter 9 - Figure 4

Code

```
ClassObject obj1 = factory.object("?", [{" " " }]);
factory.diagram([factory.diagram([
    factory.stacks([{"meas":obj1}]), obj1]),
    {"datastructure":{"vertical":true}} )
    ..draw();
```

Book <p>The diagram shows a light blue rounded rectangle representing the 'Book' class. Inside, there is a yellow box labeled 'meas =' with an arrow pointing to a red box containing a question mark. A speech bubble next to the yellow box says 'Has a getMeasure method'.</p>	Library <p>The diagram shows a light red rounded rectangle representing the 'Library' class. Inside, there is a yellow box labeled 'meas =' with an arrow pointing to a red box containing a question mark.</p>
Comments <p>Close approximation for the layout. Missing some node properties and the annotated highlight.</p>	

Chapter 10 - Figure 4 Code <pre>factory . object (title: "SavingsAccount", data: [{"balance":10000}, {"interestRate ":10}]) ..draw();</pre>	
Book <p>The diagram shows a light red rounded rectangle representing the 'Book' class. Inside, there is a red box labeled 'SavingsAccount' with a dashed line. Below the dashed line, there are two fields: 'balance =' with value '10000' and 'interestRate =' with value '10'. A speech bubble next to the box says 'BankAccount portion'.</p>	Library <p>The diagram shows a light red rounded rectangle representing the 'Library' class. Inside, there is a red box labeled 'SavingsAccount' with a dashed line. Below the dashed line, there are two fields: 'balance =' with value '10000' and 'interestRate =' with value '10'.</p>
Comments <p>Close approximation for the layout. Redesigned object shows added properties but still missing dashed line and the annotated highlight. Color was changed but can easily switch to match the book design by changing the configuration file.</p>	

Chapter 10 - Figure 5

Code

```
factory . object ( title: "SavingsAccount",  
                  data: [ {"balance":10000},  
                        {"interestRate ":10},  
                        {"balance":500}])  
  
..draw();
```

Book

Library

Comments

Improved object renders closer to actual drawing. Missing the annotated highlight and a second horizontal line.

Chapter 10 - Figure 6

Code

```
ClassObject obj1 = factory.object("SavingsAccount", [  
    {"balance":10000},  
    {"interestRate": 10}]);  
factory.diagram([factory.stacks([{"collegeFund":obj1},  
    {"anAccount":obj1},  
    {"anObject":obj1}]), tmp ] )  
  
..draw();
```

Chapter 10 - Figure 8

Code

```

ClassObject obj1 = factory.object("Coin", [{"value": ".25"},
                                           {"name": "quarter"}]);
ClassObject obj2 = factory.object("Coin", [{"value": ".25"},
                                           {"name": "quarter"}]);

factory.diagram([factory.diagram([
    factory.stacks([{"coin1": obj1}], obj1),
    factory.diagram([factory.stacks([{"coin2": obj2}],
    obj2)]),
    {"datastructure": {"vertical": true}} )
    ..draw();

```

Comments

Close approximation for the layout. Missing some node properties and the annotated highlights.

Chapter 10 - Figure 10

Code

```
ClassObject obj1 = factory.object("BankAccount", [{"balance":10000}];
ClassObject obj2 = factory.object("BankAccount", [{"balance":10000}];
factory.diagram([factory.diagram([factory.stacks([{"account":obj1}], obj1)),
                factory.stacks([{"clonedAccount":obj2}],
                                obj2)]) )

..draw();
```

Book

Library

Comments

Close approximation for the layout. Missing some node properties and the annotated arrow.

Chapter 15 - Figure 1

Code

```
String nde = "Node<String>";
ClassObject num5 = factory.object(nde, ["Tom","null"]);
ClassObject num4 = factory.object(nde, ["Romeo", {"_ptr":num5}]);
ClassObject num3 = factory.object(nde, ["Harry", {"_ptr":num4}]);
ClassObject num2 = factory.object(nde, ["Diana", {"_ptr":num3}]);
ClassObject num1 = factory.object("LinkedList<String>", [{"_ptr":num2}]);

factory.diagram([num1, num2, num3, num4, num5] )
  ..draw();
```

Book

Library

Comments

Close approximation for the general layout. The node in the second row was not automatically positioned. The lack of node properties to span the values to the full width caused the linking lines to render over the nodes.

Chapter 15 - Figure 2

Code

```
String nde = "Node";
ClassObject num5 = factory.object(nde, ["Tom","null"]);
ClassObject num4 = factory.object(nde, ["Romeo", {"_ptr":num5}]);
ClassObject num3 = factory.object(nde, ["Harry", {"_ptr":num4}]);
ClassObject num2 = factory.object(nde, ["Diana", {"_ptr":num3}]);
ClassObject num1 = factory.object("LinkedList", [{"_ptr":num2}]);

factory.diagram([num1, num2, num3, num4, num5] )
  ..draw();
```

```

String nde = "Node<String>";
ClassObject node = factory.object("Node", [{"data":"Diana"}, {"next":"" }]);
ClassObject node2 = factory.object("Node", [{"data":"Amy"}, {"next":"" }]);

ClassObject node3 = new ClassObject.from(node);
ClassObject node4 = factory.object("Node", [{"data":"Amy"}, {"next":node3}]);

Diagram row1 = factory.diagram([factory.object("LinkedList<String>",
        [{"first":node}]), node]);
Diagram row2 = factory.diagram([new Stackframe([{"newNode":node2}]),
        node2]);

Diagram row3 = factory.diagram([factory.object("LinkedList<String>",
        [{"first":node4}]), node3]);
Diagram row4 = factory.diagram([factory.stacks([{"newNode":node4}]), node4]);

factory.steps([{"title":"Before insertion", "content":[
        factory.diagram([row1, row2],
            {"datastructure":{"vertical":true}}
        ]},
        {"title":"After insertion", "content":[
        factory.diagram([row3, row4],
            {"datastructure":{"vertical":true}}
        ]}], {"datastructure":{"ordered":false}})

    ..draw();

```

Book

Library

Comments

The outer containers are positioned correctly. Within each of them it was difficult to automatically render correctly. Missing node properties as well.

Chapter 15 - Figure 5

Code

```
factory . linkedlist ( data: [ " ", "Amy", "Diana" ] )  
  ..draw();
```

Comments

Close approximation for the layout. Simpler nodes used to provide high-level view of linked list. Attempt was taken at improving the layout from the earlier figures in this chapter. Progressive container missing also.

Chapter 15 - Figure 6

Code

```
factory . linkedlist ( data: [ " ", "Diana", "Harry", "Romeo"] )
  ..draw();

factory . linkedlist ( data: [ " ", "Diana", "Romeo"] )
  ..draw();
```

Chapter 15 - Figure 7

Code

```
factory . linkedlist ( data: [ " ", "Diana", "Harry", "Romeo"] )
  ..draw();

factory . linkedlist ( data: [ " ", "Diana", "Harry", "Julian", "Romeo"] )
  ..draw();
```

Comments

This diagram is a linked but does not have a constant layout. Unable to automatically render.

Chapter 15 - Figure 9

Code

```
factory . queue ( data: [ " ", " ", " ", " " ] )  
  ..draw();
```

Book

Library

Comments

Exact approximation for the layout. Node properties complete but no annotations included in the collection.

Chapter 15 - Figure 11

Code

```
factory . queue ( data: [ " ", " ", " ", " " ] )  
  ..draw();
```

Book	Library
Comments Diagram rendered identical, automated layout identified.	

Chapter 16 - Figure 6 Code <pre>factory . linkedlist (data: ["Juliet", "Katherine", "Tony"]) ..draw();</pre>	
Book	Library
Comments Automated rendering was possible for this diagram but unable to implement.	

Chapter 16 - Figure 7 Code

```
factory . tree( data: ["Juliet","Eve","Romeo","Adam","Harry","Romeo","Tom"])
  ..draw();
```

Book

Library

Comments

Binary tree rendered with subtle layout issues in vertical alignment. The nodes were made from simpler abstracts to focus on the layout for the tree. No annotated highlights added to the tree.

Chapter 16 - Figure 8

Code

```
factory . tree( data: ["Juliet","Eve","Romeo","Adam","Harry","Romeo","Tom"])
  ..draw();
```

Book

Library

Comments

Same as Chapter 16 - Figure 7.

Chapter 16 - Figure 9

Code

```
factory . tree( data: ["Juliet","Diana","Tom","Harry"])
  ..draw();
```

Book

Library

Comments

Simple tree representation does not have properties for the nodes. The numbered annotations are missing also. The linking mechanism can be changed and automatic declarative rendering is possible.

Chapter 16 - Figure 10

Code

```
factory . tree( data: ["Juliet","Diana","Tom","Harry","Romeo"])  
  ..draw();
```

Book

Library

Comments

Tree layout failed to render correctly. Algorithm had been designed to allow spacing between all nodes. Nodes are simple in my diagram, did not apply the properties to focus more on the rendering.

Chapter 16 - Figure 11

Code

```
factory . tree( data: [50,25,75,62,80,77])  
  ..draw();
```

Book

Library

Comments

Binary search tree implemented with text hidden to render silhouette representation. Very close automated rendering, color of node was not changed.

Chapter 16 - Figure 12

Code

```
factory . tree( data: [50,25,75,62,80,78,76,77, 85])  
  ..draw();
```

<p>Book</p>	<p>Library</p>
<p>Comments</p> <p>No pattern detected in this figure to render automatically. When attempted the tree's width doubled at every level.</p>	
<p>Chapter 16 - Figure 13</p> <p>Code</p> <pre>factory . tree(data: [50,25,75,12,37,62,80,6,15,35]) ..draw();</pre>	

<p>Book</p>	<p>Library</p>
<p>Comments</p> <p>Tree was able to auto scale as seen on the book diagram. The tree in my library had precise calculations to render it in such a way that all nodes would fit at their respective levels.</p>	

<p>Chapter 16 - Figure 14</p> <p>Code</p> <pre>factory . tree(data: [20,17,30,15,19,25,35,12,16,18]) ..draw();</pre>	
<p>Book</p>	<p>Library</p>

Comments

Tree representation for execution order was not implemented. The tree would have to be built manually however the rendering would be automatic and look properly aligned.

Chapter 16 - Figure 15

Code

```
factory . tree( data: [20,17,30,15,19,25,35,12,16,18])  
  ..draw();
```

Book

Library

Comments

Tree was calculated with initial data then during the draw phase the data was hidden via property. Annotations not included in tree design. Tree rendered perfectly and has space for missing nodes on fourth level which doesn't require it to re-size.

Chapter 16 - Figure 16

Code

```
factory . tree( data: [20,17,30,15,19,25,35,12,16,18])
  ..draw();
```

Book

Library

Comments

Automated rendering layout is exact. balanced binary tree was only tree represented in my library. To construct a regular binary tree would have required procedural insertions and I was going for automated rendering.

Chapter 16 - Figure 17

Code

```
factory . tree( data: [20,17,30,15,19,25,35,12,16,18])
  ..draw();
```

Book

1 Add vacant slot at end

Library

Comments

Container for binary search tree was not implemented. Would be possible by extending the progressive diagram to accept tree types.

Chapter 16 - Figure 18

Code

```
factory . tree( data: [20,17,30,15,19,25,35,12,16,18])  
  ..draw();
```

Book

1 Remove the minimum element from the root

2 Move the last element into the root

Library

Comments

Container for binary search tree was not implemented. Would be possible by extending the progressive diagram to accept tree types.

Chapter 16 - Figure 19

Code

```
factory . tree( data: [20,17,30,15,19,25,35,12,16,18])  
  ..draw();  
  
factory . array ( data: [ " ",20,30,15,19,25,35,12,16,18])  
  ..draw();
```

Book

Library

Comments

Annotations for binary search tree was not implemented. Would be possible by extending the node diagram render outside of tree container. Same with array.

Chapter 16 - Figure 20

Code

```
factory . tree( data: [21,17,30,15,19,25,35,12,16,18,20,23])  
  ..draw();
```

Book

Library

Comments

Container for binary search tree was not implemented. Would be possible by extending the progressive diagram to accept tree types. Color coding could have been achieved by specifying a different color for nodes not at the fourth level.

Chapter 22 - Figure 1

Code

```
Table tble = new Table( [ ["Product_Code", "Description", "Price"],  
  ["116-064", "Toaster", "24.95"],  
  ["257-535", "Hair Dryer", "29.95"],  
  ["643-119", "Car Vacuum", "19.99"]],  
  factory,  
  {"nodes":{"bgcolor":[255, 241, 191]} });  
  ..draw();
```

<