Snapdown: A Text-Based Snapshot Diagram Language for Programming Education

by

Daniel Whatley

Submitted to the Department of Electrical Engineering and Computer Science on May 20, 2021, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Snapshot diagrams, which visualize in-memory program state, are frequently used in programming education to demonstrate new concepts and help students develop a better understanding of program functionality. This thesis introduces Snapdown, a textual language for drawing snapshot diagrams, designed for use by both students and instructors of programming courses. Snapdown is designed with an emphasis on learnability and simplicity: both to be picked up by students in a classroom setting in a matter of minutes, and to enable creation and maintenance of diagrams in instructional content with minimal overhead. I introduce several use cases of Snapdown and describe the design and features of its textual language. I also describe a deployment of Snapdown during two semesters of emergency remote teaching in MIT software engineering course 6.031 Software Construction, in which students used it to complete pre-class reading exercises and in-class collaborative exercises. Finally, I demonstrate that Snapdown is generally applicable by using it to replicate over 100 diagrams from introductory- and intermediate-level courses at a variety of institutions.

Thesis Supervisor: Max Goldman
Title: Principal Lecturer
Acknowledgments

This thesis would not have been possible without the guidance and support of many wonderful people.

Thank you to my parents Bill and Jingda, and my brother Alex, for dealing with all the ups and downs in my personal life. Thank you for always believing in me and always pushing me forward, no matter how hard things might seem.

Thank you to Max Goldman, my thesis advisor, for teaching me many valuable lessons about research and software development, and always helping me get unstuck from everything, from the smallest bug to the biggest design challenge. Thank you to Rob Miller for always providing valuable and helpful feedback at every step of my project. Thank you both for your constant support, as both mentors and instructors, even as classes and research went remote. Thank you to my fellow UP group members for many fun conversations, always reminding me that I’m not alone and that research truly can be a team effort.

Thank you to all the friends I’ve made over the years, and especially to the MIT Syncopasian family, and the not-so-official (yet) MIT osu! club.

Finally, thank you to all the staff members of 6.031 for making a huge positive impact on my experience at MIT. Special thanks to everyone who helped me move 6.031 forward by providing feedback on this work.
# Contents

1 Introduction ................................................................. 13

2 Related Work ................................................................. 19
   2.1 Diagramming languages .............................................. 19
   2.2 Diagram layout ......................................................... 20
   2.3 Drawing diagrams in classroom settings ....................... 21
   2.4 Generating diagrams from code ................................... 22
   2.5 Drawing diagrams with code ....................................... 23

3 Language Design ............................................................ 25
   3.1 Primitives and variables ............................................. 25
   3.2 Objects, fields, and arrays ......................................... 27
   3.3 Immutability ............................................................ 28
   3.4 Reassignments ......................................................... 28
   3.5 Name-binding and resolution ..................................... 28
   3.6 Stack frames ........................................................... 30
   3.7 Multi-step diagrams .................................................. 30

4 Implementation .............................................................. 33
   4.1 User interface ......................................................... 33
   4.2 Rendering diagrams .................................................. 35
      4.2.1 Drawing the heap .............................................. 36
      4.2.2 Drawing stack frames ....................................... 36
List of Figures

| Figure | Description |
|--------|-------------|
| 1-1    | A snapshot diagram that shows a list of three String objects. Left: Snapdown syntax for this diagram, right: corresponding diagram. |
| 1-2    | A snapshot diagram equivalent to that in Figure 1-1, but with a different level of detail desired by the creator. |
| 1-3    | Multi-step diagram demonstrating reassignment and mutation of variables. Example Java code follows. Step 1: String t = "a"; List<String> tarr = List.of(t); Step 2: String s = t; List<String> arr = tarr; Step 3: s = "ac"; Step 4: tarr.add("d"); |
| 1-4    | Example figure from Wellesley CS 111’s lecture notes on lists and memory diagrams |
| 1-5    | Snapdown representation of diagram from Fig. 1-4 |
| 3-1    | Primitives and variables |
| 3-2    | Objects, fields, and arrays |
| 3-3    | Immutability |
| 3-4    | Reassignments |
| 3-5    | Name-binding and resolution |
| 3-6    | Stack frames |
| 3-7    | Multi-step diagram demonstrating the need to keep objects in place |
| 4-1    | A portion of the Snapdown help sidebar. The sections are Primitives, Objects, Fields, Stack frames, Literals, Using names as references, Unnamed references, and Reassignments. |
| 4-2    | Completed Snapdown reading exercise |
10

4-3 Individual diagrams corresponding to the multi-step diagram in Figure 3-7

4-4 Combined diagram corresponding to the multi-step diagram in Figure 3-7

5-1 Example sequence of events for a single student, for a single exercise, to which the correct answer is a -> 25

5-2 Fall 2020, Reading 15 Snapdown exercise analysis. Showing students who answered the exercise correctly, possibly after clicking Explain. * = Students who spent more than 5 minutes on the exercise.

5-3 Fall 2020, Reading 27 Snapdown exercise analysis. Showing students who answered the exercise correctly, possibly after clicking Explain.

5-4 manyNotes exercise analysis - Spring 2020 vs. Fall 2020

5-5 Spring 2021, Reading 4 Snapdown exercise analysis. * = Students who spent more than 5 minutes on the exercise; includes both students who clicked Explain and those who didn’t.

5-6 Spring 2021, Reading 8 Snapdown exercise analysis

5-7 Spring 2021, Reading 15 Snapdown exercise analysis

5-8 Spring 2021, Reading 27 Snapdown exercise analysis

5-9 Starting Stroke diagram

5-10 Correct student-drawn Stroke diagram

5-11 Incorrect student-drawn Stroke diagram

5-12 Example figure from "OOP in Java" at UCSD

5-13 Snapdown representation of diagram from Fig. 5-12

5-14 Example snapshot diagram in EECS 280, University of Michigan

5-15 Example snapshot diagram in 6.009, MIT
5.1 Analysis of diagrams from other courses. A diagram is successfully replicated if the structure and all components are preserved, possibly with visual differences in shape or layout. A diagram is nearly replicated if it can be completely replicated with one additional Snapdown feature.
Chapter 1

Introduction

Understanding the evolution and state of variables in the lifecycle of a program is an important capability and a source of difficulty for students in introductory programming courses. Diagrams that show program state, commonly termed *memory diagrams*, are used as a teaching aid to help students overcome this difficulty [7, 28, 15]. To assist with programming education, instructors often embed these diagrams into their lecture notes, digital textbooks, online exercises, and other online course content. Students are often asked to create these diagrams to aid their mental visualization of programs and to help fulfill learning objectives. These diagrams are designed to be simple and flexible, only showing particular aspects of the program state that are relevant to the discussion at hand. I refer to these diagrams as *snapshot diagrams*, as they represent transient snapshots of memory, and depict both objects on the heap and frames on the call stack. Figure 1-1 is an example of a snapshot diagram presented early in 6.031: Elements of Software Construction, an undergraduate-level software engineering course taught at MIT, consisting mainly of sophomores and juniors. Figures 1-1 and 1-2 show two possible ways to draw a snapshot diagram illustrating the same program state. In Figure 1-2, for example, the particular values of the *String* objects, or the inner representation of the *List*, are details that can be included in or omitted from the diagram as desired.

Generating many snapshot diagrams for online course content is not an easy task. Instructors must create, update, and maintain snapshot diagrams as course content
Figure 1-1: A snapshot diagram that shows a list of three String objects. Left: Snapdown syntax for this diagram, right: corresponding diagram.

cities -> (  
  List<String>  
  0 -> (String "Boston"),  
  1 -> (String "Bogotá"),  
  2 -> (String "Barcelona")  
)

Figure 1-2: A snapshot diagram equivalent to that in Figure 1-1, but with a different level of detail desired by the creator.

cities -> (  
  List (String) (String) (String)  
)

Figure 1-3: Multi-step diagram demonstrating reassignment and mutation of variables. Example Java code follows. Step 1: String t = "a"; List<String> tarr = List.of(t); Step 2: String s = t; List<String> arr = tarr; Step 3: s = "ac"; Step 4: tarr.add("d");
Figure 1-4: Example figure from Wellesley CS 111’s lecture notes on lists and memory diagrams

```python
def myList -> [98.6, 17, 23, True, #subList, #circle]
def list2 -> myList
  def #subList -> ['I', 'was', 'not', 'Adams']
  def #circle -> (Circle
    radius -> 50
    location -> (Point
      x -> 200
      y -> 100
    )
  )
```

Figure 1-5: Snapdown representation of diagram from Fig. 1-4
changes and as students need more examples. Diagrams can be drawn by hand, constructed with an image editor like Graphviz [12] or OmniGraffle [31], or with a low-level diagramming language like DOT [9]. However, these methods are not designed to support continuous changes in diagram content and structure. In addition, some of these methods have steep learning curves, and are not feasible for students to use in completing coursework or fulfill learning objectives that involve drawing diagrams. Programming courses typically contain tens or hundreds of code snippets, examples, and exercises. In a typical semester of 6.031, for example, students are presented with approximately 150–250 diagrams, across all online reading material and in-class sessions. Several challenges would immediately surface if instructors and students were to use these methods to draw and maintain such quantities of diagrams.

In this thesis, I address these challenges with Snapdown, a text-based language for drawing snapshot diagrams, intended for use by both instructors and students. Inspired by Markdown [17], Snapdown aims to represent graphical objects using text syntax that resembles the diagram as closely as possible. Snapdown also takes advantage of the fact that a primary purpose of these diagrams is to show associations between names and values: the same names that appear as labels in the diagram can be used to refer to objects in the text source, making many diagrams more straightforward to draw. Finally, Snapdown gives users the ability to draw multi-step diagrams that show the evolution of the state of a program. Figures 1-1 and 1-2 each show Snapdown syntax on the left for the corresponding diagram on the right, and Figure 1-3 contains an example four-step diagram drawn with Snapdown.

I evaluated the learnability of Snapdown by asking students and instructors of 6.031 to draw diagrams with it. Snapdown proved to be an advantage during the COVID-19 pandemic: students could complete diagram exercises online using Snapdown without a drawing tablet or other means. In addition, 6.031 incorporates pair programming exercises into its class sessions. Using Snapdown, students collaboratively edited diagrams in their web browsers. Overall, approximately 500 students used Snapdown across two semesters, Fall 2020 and Spring 2021, of emergency remote teaching in 6.031. Instructors of 6.031 created approximately 10–15 new exercises and
20 new diagrams using Snapdown for course material.

I also evaluated the generalizability of Snapdown. Using Snapdown, I recreated over 100 diagrams from approximately 15 programming courses at a variety of institutions. For example, Figure 1-4 shows a snapshot diagram presented in the lecture notes of CS 111 at Wellesley College, and Figure 1-5 shows the equivalent representation in Snapdown.

This thesis makes the following contributions:

• *Snapdown*, a textual language for drawing snapshot diagrams, designed to be learnable by students in a matter of minutes, with a syntax that resembles the shape of diagram elements, and that uses the same syntax for drawing names in the diagram and binding names in text.

• A syntax and mechanism for rendering *multi-step diagrams* that is easy to write and maintain as examples grow or improve, and that preserves visual consistency between steps.

• An implementation of Snapdown that generates SVG diagrams with an intermediate representation in JSON. This implementation allows Snapdown to be embedded in any web application and facilitates collaborative drawing of diagrams using an existing collaborative text editor.

• An analysis of how students and instructors in a large introductory software engineering course used Snapdown over the course of two semesters, and an examination of Snapdown’s ability to replicate diagrams from other courses.

Chapter 2 reviews related work in diagram layout engines, and diagrams and visualizations used in programming education. Chapter 3 introduces Snapdown and explains its design and syntax. Chapter 4 details outlines the situations in which Snapdown has been used, Snapdown’s technical layout routine, and its approach to both single-step and multi-step diagrams. Chapter 5 explains my evaluations of the learnability and effectiveness of Snapdown, both in its deployment in 6.031 and its ability to replicate diagrams found in courses at other institutions. In Chapter 6,
I conclude this thesis, and discuss limitations of my approach and potential future work.
Chapter 2

Related Work

2.1 Diagramming languages

Several diagramming languages are well-known and commonly used to draw diagrams for a variety of purposes, including ones related to programming education. Systems such as Penrose [22], DOT [9], Mermaid [18], and PlantUML [23] are examples. DOT is a low-level language that is capable of drawing arbitrary diagrams. Penrose, Mermaid and PlantUML all share Snapdown’s approach of providing an easily accessible, high-level language, but they output different types of diagrams than Snapdown does. Penrose focuses on 3-D geometry diagrams, while Mermaid and PlantUML are centered on UML diagrams. Object diagrams in UML are similar to snapshot diagrams, but are much more focused on class and object hierarchy, not on the state of variables in a program.

Overall, these systems solve different problems and focus on different use cases. For example, UML object diagrams do not contain compound nodes, while most snapshot diagrams do. (In Figure 1-3, the Array<"string"> objects are all examples of compound nodes: they have internal fields such as 0 and 1 which point to other nodes, such as the string objects.) Drawing a visually appealing snapshot diagram in a system designed for UML diagrams would require careful manual positioning or substantial syntax additions. This added work presents a complexity barrier and distracts the user from their primary goal of drawing a diagram.
Of these systems, I believe that PlantUML is most similar to Snapdown in terms of approach and types of diagrams supported. PlantUML, like Snapdown, puts emphasis on drawing diagrams instead of generating them from existing code. Its intended audience, however, appears to differ from that of Snapdown: many example diagrams on the PlantUML webpage demonstrate its use in a workplace setting. Parts of its FAQ section address concerns specific to corporate needs, such as the confidentiality of diagrams and the security of its system. This suggests that the design of PlantUML, as well as the types of diagrams it generates, are not immediately applicable to Snapdown’s intended audience of students and instructors. In addition, many features of snapshot diagrams are difficult to express in PlantUML syntax, and some features—such as those that require compound nodes—are not supported by some PlantUML diagram types, including the PlantUML Class diagram. Other diagram types, such as the PlantUML Component diagram, support a wider range of diagram layouts but are not designed to visualize program state.

2.2 Diagram layout

Automatic diagram layout is not a focus of Snapdown’s approach or the problem it solves. In this thesis, however, I built on innovations and systems that past researchers have explored and used successfully. Ruegg et al. have published several papers and technical reports demonstrating diagram compaction and layout algorithms [27, 25, 26]. Their work was done using the Eclipse Layout Kernel (ELK) [8]. In a general sense, the authors took advantage of the fact that ELK is open-source, and contributed their work to the project. These results, among those by other researchers in other contexts also achieved with ELK as the main layout library, indicate that ELK is a good choice for my purposes of Snapdown implementation.
2.3 Drawing diagrams in classroom settings

Research has shown that drawing diagrams is difficult yet beneficial for students in introductory programming classes. Sajaniemi et al. [28] performed studies with students in an introductory course centered on object-oriented programming, collected many student-drawn diagrams to different types of exercises, and analyzed them to check for common mistakes and misconceptions. Based on the diagrams, a small number of key concepts, such as objects and constructors, caused the most confusion. A study by Holliday and Luginbuhl [15] involved giving two different versions of a quiz to students, one with multiple-choice questions about Java code, and another asking students to draw snapshot diagrams. Students scored significantly lower on the snapshot diagram questions, leading the authors to conclude that drawing correct diagrams requires a deeper understanding of course material. In these studies, students drew diagrams limited to visualizations of objects on the heap at a single point in the execution of a program. With Snapdown, I aim to design a language that can not only visualize the heap, but also the call stack, and that can accurately capture the evolution of both the heap and stack through multi-step diagrams.

Researchers have also performed experiments to show the effectiveness of visualizations in different settings. The Holliday and Luginbuhl study, for example, also demonstrated that a student’s ability to draw memory diagrams is positively correlated with an assessment of their understanding. A study by Hendrix et al. [14] divided participants into two groups, one presented only with a question about a code snippet, and the second presented with the same question and code snippet but also an accompanying diagram. The group presented with the diagram performed significantly better on the question than the group without. A study by Baltes and Diehl [1] demonstrated the effectiveness of visualizations in a workplace setting, and concluded that visualizations can be a helpful way to enhance out-of-date documentation.

Snapshot diagrams currently presented in 6.031 and other courses I surveyed were drawn in many different ways. During class, instructors mostly drew them by hand to explain concepts. Instructors also drew diagrams presented in online reading mater-
rial using software such as OmniGraffle [31], SVG-Edit [30], and sometimes Microsoft Paint. One in-class exercise in 6.031 asked students to fill in a blank snapshot diagram by drawing arrows and ovals in Google Drawings [11]. These systems all have a low learning overhead and offer great flexibility in customizing every part of the diagram: users, especially students, do not need to learn HTML or a new language to draw diagrams using them. However, because so many systems are used for drawing, there is no longer a sense of consistency in diagrams. In addition, many of these drawing tools are based on manual drawing or drag-and-drop operations using a mouse or tablet. Maintenance of tens or hundreds of diagrams can get very cumbersome, especially as course material changes and is revised from semester to semester. The Snapdown language makes this maintenance much easier, often reducing large updates to a single find-and-replace operation. Also, while Snapdown is not as highly customizable as many of these systems, it creates diagrams with a consistent format and style, making Snapdown diagrams a viable tool to communicate program state [7].

Peer instruction and collaborative learning have been proven to achieve beneficial results in classroom settings [13] [4] [19]. Snapdown’s learnable syntax and user interface allows for students and instructors to use it in a large variety of settings, including in-class exercises, pre-class readings, and quizzes and exams. By enabling students to collaboratively edit diagrams, which I have done during the COVID-19 pandemic, I build off previous research in education and create opportunities to improve student learning.

2.4 Generating diagrams from code

The problem of visualizing executable code with diagrams has been explored extensively. Python Tutor [24] is a well-known example of a system that generates snapshot diagrams, with both stack and heap, from executable code in several programming languages. GitUML [10] is an example of a system that can generate UML diagrams from GitHub repositories containing object-oriented executable code. Systems built
into certain IDEs such as PyCharm for IntelliJ IDEA [16] and ObjectAid for Eclipse [20], can generate UML diagrams from code in existing projects. A paper by Dalton and Kreahling [5] took a step toward automatically generating snapshot diagrams from executable code in an instructional setting, with a focus on generating incorrect diagrams for teaching purposes. Incorrect diagrams were generated through MDL, a custom language specified by the authors, that resembles executable code.

Compared to these systems, Snapdown has different requirements and makes different tradeoffs. For example, diagrams drawn directly from executable code can become very large and contain parts of the heap or stack that are beyond the scope of course material. I believe my approach better serves students and instructors who only wish to visualize parts of the heap or stack relevant to their specific needs. In addition, many code snippets presented in course material are incomplete and do not compile and run out-of-the-box. With Snapdown, users can both create and maintain snapshot diagrams illustrating these code snippets, without needing to write working code or refactoring the code every time they wish to make substantial changes to how the examples are presented.

2.5 Drawing diagrams with code

Constrain [3] is an example of a system designed to solve the same problem as Snapdown, but with a different approach. Constrain is used in classes such as CS 2112 at Cornell to demonstrate evolving states of programs. To draw simple diagrams in Constrain, users write approximately 20 lines of JavaScript code—complex diagrams can require much more. Such code, while possible for instructors to write and use to draw a large range of diagrams, would be infeasible for students to write for their coursework. Diagramcodes [6] and Structurizr [29] are two other examples of systems that draw diagrams from a representation written in code, and each have a similar complexity overhead to that of Constrain. In comparison with these systems, Snapdown’s language is much smaller and constrained and has much less flexibility. The benefit is that Snapdown does not require students to learn a new API in a new
programming language, which greatly smoothens its learning curve.
Chapter 3

Language Design

Snapdown allows users to draw snapshot diagrams with plain text. This section describe the features that Snapdown supports. The examples presented in this section were chosen with the Java programming language in mind, but Snapdown is language-independent. When users type Snapdown syntax, they can include single-line comments starting with //, similar to languages such as Java and C.

3.1 Primitives and variables

Figure 3-1 shows examples of Snapdown syntax for primitive data types and variables, as well as the corresponding diagram. Arrows between variable names and their values are represented with a textual arrow, ->.

Variable names can be a single alphanumeric word, similar to those in most programming languages. For the purpose of snapshot diagrams, however, we may want...
f -> (MyFloat 5.0)
arr -> [ (String), (String)]
map -> ( HashMap
  "key": "value"
 )
lst -> ( ArrayList
  0 -> 1000
  1 -> 2000
  2 -> 3000
  length -> 3
 )

Figure 3-2: Objects, fields, and arrays

a => ( MyClass
  b -> (MyImmutableClass))

Figure 3-3: Immutability

a -x> 1
a -> 2
b -> a

Figure 3-4: Reassignments

26
Figure 3-5: Name-binding and resolution

to include the types of variables as well. Placing backticks around literals, such as `int i`, allows for any string literal to be interpreted as a variable name or a value.

Note how all arrows point downward, and how variables are introduced left-to-right as they are written in Snapdown. This design was based on an evaluation of diagrams drawn on paper and collected in the course of 6.031, and drawn by course staff during an initial exploratory study. Students and staff both had a strong preference to draw arrows top-to-bottom and introduce variables left-to-right.

### 3.2 Objects, fields, and arrays

Figure 3-2 shows an example of objects, fields, and arrays, and the corresponding Snapdown syntax. Parentheses are used around class names and fields to resemble the round shape of an object in the diagram. The `MyFloat` object contains only its inherent value of 5.0, and the array contains two `String` objects. The `HashMap` object contains a single key-value pairing, represented with the syntax "key": "value". The third object goes into more detail on the inner structure of the `ArrayList`. It has four fields, 0, 1, 2 and `length`, which may or may not actually exist in the representation of
the object in a programming language. Objects can be represented in different ways depending on the desired level of detail. Objects can also be nested within others, as demonstrated in Figures 1-1 and 1-2 from the introduction to this thesis. In a single diagram, adding, removing, or updating names of objects or their fields can be done by simply editing the text or using a find-and-replace operation.

### 3.3 Immutability

Keywords such as `final` and `const` are commonly used in programming languages to indicate immutability of references, or variables that cannot be reassigned. In addition, some types of objects are immutable if they have no mutator operations. Immutability is a concept commonly presented in introductory programming courses, as it helps prevent many bugs involving aliasing and unintentional mutation. In Snapdown, users have the option to indicate that certain references and objects in their diagrams are immutable using double-lined arrows and double-layered parentheses, respectively, as demonstrated in Figure 3-3.

### 3.4 Reassignments

Variables in a program are typically assigned new values over time. To show the evolving nature of programs, Snapdown allows users to draw both old and new values of variables using reassignments. Figure 3-4 shows a diagram illustrating a code segment such as `int a = 1; a = 2; int b = a;`. Crossed-out arrows are drawn with the syntax `-x>`, as it resembles what is drawn in the diagram as closely as possible.

### 3.5 Name-binding and resolution

Values in a program are typically associated with names. Snapdown takes an approach to name-binding that allows the user to easily use the names they have associated with parts of a diagram. In addition to a value or object, the right-hand side of an
arrow can also be a name. In the case that the right-hand side is a name, Snapdown infers that the name must be looked up and found elsewhere in the diagram.

Figure 3-5 shows an example of such a diagram with names for both variables (such as cs1 and cs2) and fields (such as semester, season, and year). The names cs1 and cs2 are used in the definition of the Set<Course> object, and Snapdown looks them up. The first definitions of each cs1 and cs2 in the Snapdown text correspond to each of the two Course objects, which are thus where the arrows from the Set<Course> object point. If Snapdown is unable to lookup a name the user provides on the right-hand side of any arrow, it reports an error stating which name could not be looked up. Many programs also have duplicate names and objects with identical field names. If a variable name is defined multiple times, the first definition of that name in the Snapdown text is always used.

Some values also may not be associated with a name. To give a reference to these unnamed values, and to help disambiguate between identical names, users can include a # in the names they write. Here, the semester fields of both Course objects are named the same, so to disambiguate, we can use the notation semester#1 and semester#2. #-disambiguation is helpful to indicate that semester#2 is reassigned. If #-disambiguation was not used here, then the first instance of semester would have been the field in the Course object on the left. In addition, the name #sem is associated with the Semester object representing Fall 2020 in the Snapdown snippet. Snapdown still binds the name #sem to this Semester object, but does not draw the name #sem itself.

Finally, the example in Figure 3-4 demonstrates a important feature of name-binding in Snapdown that applies specifically when a variable is reassigned. Here, when the user types b -> a, Snapdown infers that b should point to the reassigned value of 2.
3.6 Stack frames

Figure 3-6 shows an example of Snapdown syntax for stack frames and the corresponding diagram. This diagram represents a call stack of three methods: `topOfStack()`, `lowerInStack()`, and `main()`, each of which can contain local variable definitions. The method `topOfStack()` is likely within the class `MyClass`, and the `this` reference points at the `MyClass` object.

3.7 Multi-step diagrams

In many cases, a single diagram containing reassignments is not enough to communicate the evolving nature of a program. In Snapdown, users can draw multi-step diagrams by including dashed lines, `---`, between steps that illustrate consecutive code fragments in a program which build off each other. In Figure 3-7, `#sam` is used both to represent the initial value of `a` in Step 1, and element 0 of the list in Step
2. The variable \( a \) is reassigned between Steps 1 and 2. Drawing an arrow from \( a \) in Step 2 clears out all previous arrows from \( a \) and draws a new arrow from \( a \). Figure 1-3 in the introduction also demonstrates a feature used to demonstrate mutation of objects. The syntax \texttt{tarr add 1 -> (string "d")} adds a new field, 1, to the object—in this case a \texttt{List<String>}—bound to the name \texttt{tarr}.

One method to draw multi-step diagrams, used for example by Constrain, would be to animate transitions between snapshots. I take a different approach that does not rely on transitions to communicate the changes between each snapshot, and allows users to scroll between steps of a diagram at an arbitrary speed. My method requires that if the same object is present in multiple steps of a multi-step diagram, \textit{it should not change position} physically within the diagram. In the first frame of the diagram presented in Figure 3-7, the arrow is intentionally drawn much longer than usual to ensure a fixed position for this object.
Chapter 4

Implementation

4.1 User interface

Snapdown’s user interface is its textual language, which can be embedded and integrated into many different situations and use cases. Here, I describe some scenarios in which Snapdown has been used and tested by students and instructors.

**Snapdown web app:** Snapdown is embedded in a simple, minimal web app that can serve as a sandbox for users to try out its features. This web app contains 1) a textbox in which the user can type Snapdown syntax, and 2) a diagram that live-updates to reflect the user’s input. Users are also presented with a help sidebar that documents common Snapdown features, a portion of which is shown in Figure 4.1. This help sidebar contains an explanation of one representative example for each core feature, the Snapdown text for that example, and the corresponding diagram. These examples can serve as templates for more complex diagrams users wish to draw: users can copy-paste the canned syntax examples and modify them as necessary. After the user has drawn a diagram, they are given options to export the diagram to an SVG file.

**Pre-class readings:** Snapdown is used in 6.031’s pre-class required reading material. When preparing reading material with embedded snapshot diagrams, instructors can type Snapdown syntax within embedded HTML `<script>` tags, and include the Snapdown library JavaScript file. Snapdown converts these HTML elements to SVG
Snapdown is a language for drawing snapshot diagrams. Click on the links below to see examples of Snapdown in action! In each example, you’ll see a snippet of Snapdown syntax and the corresponding diagram side-by-side.

**Primitives** (click to hide)
In this example, we have two variables: \( i \) points to the value 5, and \( s \) points to the value "abc".

\[
i \rightarrow 5 \\
s \rightarrow "abc"
\]

**Objects** (click to expand)

**Fields** (click to expand)

---

**Figure 4-1:** A portion of the Snapdown help sidebar. The sections are Primitives, Objects, Fields, Stack frames, Literals, Using names as references, Unnamed references, and Reassignments.

---

**Figure 4-2:** Completed Snapdown reading exercise.

diagrams and embeds them within the final version of the reading that is released to students.

**Pre-class reading exercises:** Readings in 6.031 contain embedded exercises, usually
multiple-choice or fill-in-the-blank. Snapdown was used by students to complete longer fill-in-the-blank reading exercises on the 6.031 website. Students can type Snapdown in a setting similar to that of the web app: students type in a text box and a diagram is generated to the right of it. Clicking the “Check” button provides feedback on their diagram. A screenshot of a completed reading exercise involving Snapdown is shown in Figure 4-2. The link to open the Snapdown help sidebar calls a Snapdown library function to open the sidebar.

**Collaborative in-class exercises:** Snapdown was used by students to complete collaborative in-class exercises that involve drawing snapshot diagrams. Using Constellation [2], a plug-in for the Eclipse IDE that also provides a web interface for pair editing, students formed pairs and edited Snapdown syntax to complete snapshot diagrams that illustrate code segments. After the exercise was over, TAs gave feedback on diagrams drawn by each pair.

**Usage by instructors:** Snapdown was used by 6.031 instructors during class to explain concepts, such as mutability and aliasing, best explained with snapshot diagrams. Instructors had the option of using Snapdown for explaining these concepts, thus achieving cleaner and more consistent diagrams, as opposed to drawing on a canvas software. Instructors also developed the in-class and reading exercises that involve Snapdown. Writing these exercises required instructors to iterate on different versions of starting and completed diagrams, and Snapdown made this iterative process much easier.

### 4.2 Rendering diagrams

Snapdown is implemented\(^1\) in JavaScript and uses ELK [8] as its layout engine. Since I found that students and staff prefer arrows to be drawn downward, I chose ELK *Layered*, a layout algorithm that can support the requirement for directionality.

Snapdown has an intermediate JSON representation to decouple parsing of Snapdown syntax from diagram layout and rendering. Snapdown syntax is parsed, eval-

\(^1\)Implementation hosted at [http://github.com/uid/snapdown/](http://github.com/uid/snapdown/)
uated, and converted into this intermediate JSON representation. This JSON representation is fed into a layout engine, and finally the laid-out diagram is drawn as a scalable vector graphics (SVG) image in the browser. Because Snapdown outputs an SVG, any web application with Snapdown embedded has great flexibility in post-processing or applying a custom CSS stylesheet to Snapdown diagrams.

I now describe how Snapdown visualizes the heap, how it draws stack frames, and how it handles multi-step diagrams.

### 4.2.1 Drawing the heap

Snapdown’s implementation parses heap elements and performs name resolution on every named object, value, and field in the diagram. Through this name resolution as described in Section 3.4, each object, variable name, and value is assigned a unique ID. Each arrow is also assigned a specific source ID and destination ID. ELK requires an input representation consisting of explicit nodes and edges, and this assignment of IDs gives ELK the information it needs to lay out the diagram. Assigning IDs is also crucial to the layout of multi-step diagrams, as explained later.

### 4.2.2 Drawing stack frames

For diagrams with stack frames, my approach is more complicated, since these diagrams have arrows that point in more directions than just downward. Currently, I apply a global ELK setting that specifies that the same layout direction should be used for all nodes when ELK runs its Layered algorithm. At a high level, this setting does not allow edges to be drawn in two different directions. Other settings for diagram directionality are available, but they substantially alter the layout of the heap, and in particular do not support hierarchy-crossing edges. To work around this limitation, I use PathFinding.js [21], an external pathfinding library, to lay out edges between the heap and the stack.

My approach treats the stack diagram and the heap diagram as separate, and runs Snapdown’s full layout routine twice, with a different ELK directionality setting on
each diagram. In Figure 3-6, for example, the three rectangular frames are considered one diagram, and the 5 and MyClass object are considered a separate diagram. Then I develop a set of constraints according to the nodes and edges that already exist in the diagram, and this pathfinding library lays out edges between the stack and heap. Finally, the two diagrams are combined and the required edges drawn.

### 4.2.3 Multi-step diagrams

Multi-step diagrams are the biggest challenge, since diagram elements are expected to remain in the same positions across steps.

Each frame in a multi-step diagram represents a diff: for example, objects that are added or variables that are reassigned. In Figure 1-3, for instance, between Steps 0 and 1, the introduction of the two references s and arr is part of the diff between
those two steps. Starting with the initial diagram (Step 0), each diff is applied to
to obtain a sequence of individual diagrams. Individual diagrams contain all the elements
necessary for each step, but do not yet keep those elements in stable positions. Figure
3-7 shows an example of the individual diagrams corresponding to each frame of Figure
3-7. These individual diagrams do not yet account for updates from previous or future
steps. While constructing individual diagrams, Snapdown preserves the IDs assigned
to each diagram element. An object included in two distinct animation frames must
have the same ID in both frames.

Next, Snapdown constructs a combined diagram. Every object and arrow ever
drawn in the diagram will be included in the combined diagram. It is not sufficient
to just make the last frame in sequence the combined diagram, since some arrows
and objects may have been deleted prior to the last animation step. As an example,
Figure 4-4 shows the combined diagram corresponding to the multi-step diagram in
Figure 3-7. For example, multiple arrows are drawn from a, and only one is included
in each frame, since a is reassigned between Steps 1 and 2.

In addition, the IDs assigned to objects in the combined diagram must exactly
correspond with those assigned when constructing individual diagrams. This requires
information to be retained after individual diagram construction; this information is
passed over to the combined diagram construction routine. The final set of diagrams
(as depicted in Figures 1-3 and 3-7) are a result of deleting elements from the combined
diagram until the remaining set of elements exactly match those in each individual
diagram.

4.3 Implementation timeline

To give context to my evaluation, I briefly describe the timeline of Snapdown imple-
mentation.

Aaron Zeng, a past TA for 6.031, developed an existing ELK-based web prototype
of Snapdown, which was the basis of my implementation work. Over the summer and
fall of 2020, I added the bulk of the functionality necessary for drawing the heap, in-
cluding name-binding, #-disambiguation, and a rudimentary version of reassignments. By the time Snapdown was deployed in the Fall 2020 semester of 6.031, Snapdown supported all features contained in Sections 3.1 through 3.5. In January 2021, I added the ability to draw stack frames. Snapdown also supported stack frames by its deployment in Spring 2021. Over the spring of 2021, I added the ability to draw multi-step diagrams.
Chapter 5

Evaluation

We evaluated Snapdown by having both students and instructors use it in the course of taking or teaching 6.031. I discuss the process of deploying Snapdown in 6.031 over the Fall 2020 and Spring 2021 semesters, and describe my evaluation of its learnability. To evaluate the generalizability of Snapdown, I also used it to replicate diagrams from courses at other institutions, and I describe my results.

5.1 Deployment in readings and in class

Snapdown was presented in a total of six pre-class readings and in four class sessions of 6.031. In sections 5.1.1 and 5.1.2, I present my results from including Snapdown in 6.031 readings in the Fall 2020 and Spring 2021 semesters, respectively. Each exercise contained a text input field into which students could type Snapdown syntax. Students were given the source for an initial starting diagram, and were expected to complete the rest of the diagram according to the provided instructions. Within the exercise, students could also click a link to access a Snapdown help sidebar. After every sequence of Snapdown exercises in a reading was a feedback box in which students could comment on their experience. Each exercise had an Explain button that students can click if they wish to give up and see the answer before figuring it out on their own, as shown in [4-2]. Student submissions were checked for correctness using a regular expression on the Snapdown source.
Figure 5-1: Example sequence of events for a single student, for a single exercise, to which the correct answer is a -> 25.

I recorded the time students spent on each exercise. Every update a student made to the exercise text field was recorded through a Google form, with anonymous information about the student’s browser session, the exercise name, and the Snapdown source currently in the text field at the time. For Fall 2020 exercises, I define the time a student spent as the time between their first recorded typing event and the time at which their Snapdown source matches the regex for the exercise. In Fall 2020, I did not track when students clicked the Explain button, but in Spring 2021, I did. For Spring 2021 exercises, the clock stopped ticking as soon as the student clicked Explain. See Figure 5-1 for a visualization of a possible sequence of events for a single student. In Fall 2020, the Explain button event is skipped, and this student is considered to have spent 10 seconds on this exercise. In Spring 2021, the student is considered to have spent 5 seconds.

5.1.1 Reading exercises: Fall 2020 semester

In the Fall 2020 semester, I was still in the process of implementing Snapdown, and developed reading exercises as more features became ready. During this semester, with the help of 6.031 instructors, I developed Snapdown exercises for two 6.031 readings.

Reading 15, Equality: In previous semesters of 6.031 (i.e. Spring 2020 and before), Reading 15 had an exercise, named circles, which asked students to draw a diagram on paper first and then select the number of HashMap objects they drew. A correct diagram illustrates the state of a small code snippet, shown in Listing 5.1. As my
first test run of Snapdown in reading exercises, I deployed a version of the *circles* exercise that asked for a diagram drawn using Snapdown.

Listing 5.1: Circles exercise code snippet

```java
Map<String, Integer> a = new HashMap<>(), b = new HashMap<>();
String c = "c";
a.put(c, 130); // put ints into the maps
b.put(c, 130);
```

Students were given a starting diagram illustrating the features of Snapdown necessary to draw the entire diagram, but with some arrows, objects, and fields left out. Before the reading was released to students, 6.031 staff members playtested this exercise and left generally positive feedback. Some staff found the exercise confusing initially, but the help sidebar cleared things up.

Students who got this exercise correct took an average of 81.0 seconds to do so. Figure 5-2 shows an analysis of student responses to this exercise. Approximately 209 students attempted this exercise. This analysis filters out approximately 25 students whose time spent was recorded as zero. I hypothesize that these 25 students clicked Explain and copy-pasted the correct answer into the box, and then clicked Check, therefore only triggering one recorded event. It is also possible that more students who answered correctly made some unsuccessful attempts at first, then clicked Explain, and copy-pasted the correct answer, thus having a nonzero recorded time. In the implementations of Snapdown reading exercises in Spring 2021, I made sure to track when students clicked Explain, so that I could differentiate between students who made a true attempt at the exercise and those who gave up. All graphs presented in this chapter also filter out an insignificant number of students who triggered at least one recorded event, but did not end up getting the exercise correct. I believe these students either were disengaged, or may have refreshed the page, thus incurring a new session ID.

For this exercise, some students found Snapdown very intuitive, and commented that it should be used across more reading exercises. Some found Snapdown diagrams
Figure 5-2: Fall 2020, Reading 15 Snapdown exercise analysis. Showing students who answered the exercise correctly, possibly after clicking Explain. * = Students who spent more than 5 minutes on the exercise.

easy to edit (by changing bits and pieces of the textual representation) but hard to create from scratch. Others found it learnable, getting it correct within a small number of tries, but commented on the large number of parentheses required to draw all the necessary objects. The initial diagram did not have line breaks between each field, possibly leading to students not realizing that each field could be started on a new line. The help sidebar did use line breaks between fields, but I hypothesize that students took an approach of editing the initial diagram rather than creating their own.

During the class period for which this reading was assigned, the 6.031 staff ran an in-class exercise where students were given the option to use Snapdown. The exercise involved drawing a single object with four fields, and the fields were a mix of primitives and other objects. Students who used Snapdown for this exercise had to do it in a shorter amount of time, and I noticed that they tended to copy-paste from the help sidebar examples before trying to draw actual elements of the necessary diagram. As a result, more students realized that they could use line breaks to separate fields of a single object.

One issue that came up frequently, and that was commented on both by 6.031
students and staff, was the lack of specific error messages when Snapdown could not parse the input given by the user. When the user entered invalid syntax, the diagram would disappear, replaced by a message that says "Unable to parse Snapdown input." This was partially a limitation of Snapdown at the time, but is also partially a design issue with the web app: displaying the old diagram, along with an option to revert back to the old diagram (possibly showing the changes made to get from the old diagram to the broken diagram), would have helped.

*Reading 27, Little Languages I:* I deployed two more reading exercises involving Snapdown in Reading 27 on little languages. This reading presented a large code example based on a music language which contains classes like `Note`, `Rest`, and `Concat`, designed to represent pieces of music. These classes are all subtypes of a `Music` interface. For example, the first two notes of the song "Row, row, row your boat" could be represented recursively as `Concat(Concat(Note("C"), Note("C")), Rest(0))`, which uses five `Music` objects. Before Fall 2020, this reading included an exercise—which I call `manyNotes`—asking students to select how many `Music` objects it would take to represent all 27 notes of "Row, row, row your boat." This exercise was multiple-choice: the choices were 27, 28, 29, 54, 55, 56, and "more than 56". The correct answer was 55.

From looking at data collected in the Spring 2020 version of Reading 27, I noticed that students had a particularly hard time with this exercise because the music piece was large, and the snapshot diagram they had to imagine would have been deeply recursive and nested. I hypothesized that asking students to draw smaller snapshot diagram using these code examples would help improve student performance on this exercise. To this end, I deployed two new exercises, which I call `row` and `twoNotes`, that involve drawing smaller diagrams with Snapdown, leading up to the `manyNotes` exercise. The `row` exercise asked for a snapshot diagram illustrating just the first note of "Row, row, row your boat", and `twoNotes` asked for a snapshot diagram illustrating the first two notes. Upon completing these two exercises, students would see correct diagrams showing that one and two notes takes $1 \times 2 + 1 = 3$ and $2 \times 2 + 1 = 5$ `Music` objects, respectively. I included, verbatim, the same `manyNotes` exercise asking for
Figure 5-3: Fall 2020, Reading 27 Snapdown exercise analysis. Showing students who answered the exercise correctly, possibly after clicking Explain.

how many Music objects it would take to represent 27 notes, which would have been $27 \times 2 + 1 = 55$.

Figure 5-3 shows my analysis of student responses to the row and twoNotes exercises. Students who got these exercises correctly took, on average, 147.5 and 102.3 seconds, respectively. For the manyNotes exercise, I compared the number of correct answers with the number in the previous semester, Spring 2020. Figure 5-4 shows the number of attempts made by students in Spring 2020 and Fall 2020: the $x$-axis represents the number of students, the $y$-axis represents the number of attempts. There was not a statistically significant difference (ANOVA, $p = 0.692$) in the number of attempts needed to complete this exercise. I hypothesize that some students, especially those who took 2 or 3 attempts, may have chosen the wrong option on their first attempt, clicked the "Explain" button as their second, and either did not finish the exercise or chose the correct option as their third. Once again, I was limited by not knowing when students clicked the Explain button.

5.1.2 Reading exercises: Spring 2021 semester

In the Spring 2021 semester, I had developed enough language features of Snapdown to use it at the beginning of the semester. I was able to include Snapdown reading exercises much earlier than I did in Fall 2020.

Reading 4, Code Review: Figure 5-5 shows an analysis of student responses to
Figure 5-4: manyNotes exercise analysis - Spring 2020 vs. Fall 2020

Figure 5-5: Spring 2021, Reading 4 Snapdown exercise analysis. * = Students who spent more than 5 minutes on the exercise; includes both students who clicked Explain and those who didn’t.
a sequence of two exercises, named heap and stack, in Reading 4 on code review. During these two exercises, students in this semester were seeing Snapdown for the first time. Approximately 240 students were actively participating in the course at the time, and 226 students completed at least one of these two exercises.

For each exercise, students who did not click the Explain button took an average of about 2 minutes to get the exercise correct (heap: 127.8 seconds, stack: 145.2 seconds). Approximately 40 students clicked the Explain button almost immediately for both exercises. For students who spent more than 120 seconds on either exercise, I analyzed their sequences of answers. For both exercises, I believe these students were simply slow to adopt Snapdown: it was a new system, and these students clearly showed a tendency to slow down and think carefully. The help sidebar also presented a large number of features, which may have been daunting to some students, especially to those who clicked the Explain button. In the text of the exercise, it was specified which features were relevant, but it may have been better to only present the features the exercise text said would be useful.

The stack exercise asked students to both draw a Payment object, and represent a static variable in Java named Payment.taxRate outside of the Payment circle. The text of the exercise directed students to use the notation Payment.taxRate in their diagram to refer to this variable. However, many students got stuck here—for example, some tried to represent taxRate as a field of Payment, and others only used taxRate and not Payment.taxRate—leading them to click the Explain button. Before clicking Explain, some students clicked in the help sidebar, thinking their issue may be resolved there. In the future, I believe I could put the instructions to use Payment.taxRate in the starting text for the diagram, or simply accept taxRate as an alternative correct answer.

Reading 8, Mutability & Immutability: In previous versions of this reading (Fall 2020 and before), students were presented with a sequence of five exercises asking to draw and update a snapshot diagram on paper, and answer questions such as “How many circles did you draw?” or “What variable name did you introduce?” I included 5 exercises, snap1 through snap5, that ask students to draw these diagrams with
Snapdown. These exercises asked students to draw a sequence of diagrams illustrating the state of variables in a method, `dropCourse6()`, shown in Listing 5.2. The implementation of this method intends to remove all course numbers starting with 6. in the `ArrayList` parameter, but in reality, it iterates over the list while mutating it, resulting in an incorrect result. Students were asked about the method call `dropCourse6(["6.045", "6.031", "6.036"]). The exercises asked students to draw a diagram corresponding to each of the following places in the code:

1. snap1: Right at the beginning of the `dropCourse6` method body
2. snap2: After the first line that introduces `iter`
3. snap3: After the `while` loop is entered
4. snap4: After one subject is removed
5. snap5: In the next iteration of the loop

See Figure 5-6 for an analysis of the time it took for students to answer each of these questions correctly. Students gave feedback that there was lots of scrolling needed between the place the code was presented in the reading and the exercise box. For each exercise, approximately 30-35% of the class clicked Explain within the first 30 seconds. Approximately 28% of the class clicked Explain within the first 30 seconds for all five exercises.

In the graph for snap4, almost all students completed the exercise within the first 30 seconds. For this exercise, the automatic checker for student-drawn diagrams was very lenient, and accepted many diagrams with slight inaccuracies. When 6.031 staff members playtested this exercise, they also gave comments on the leniency of the checker. The checker currently uses regular expressions to check student submissions. Future work includes implementing an automatic checker for Snapdown exercises.

There was also no required change between snap3 and snap4: copy-pasting the correct answer from snap3 and clicking Check on snap4 would report the answer as correct. Multi-step diagrams were not yet ready at this point in the semester. For
future iterations of this exercise, students can be asked to draw a single multi-step diagram, thus making the process of checking and answering the exercises easier.

Listing 5.2: Exercise code snippet for snap1 through snap5

```java
public static void dropCourse6(ArrayList<String> subjects) {
    MyIterator iter = new MyIterator(subjects);
    while (iter.hasNext()) {
        String subject = iter.next();
        if (subject.startsWith("6.")) {
            subjects.remove(subject);
        }
    }
}
```

Reading 15, Equality: In this reading, I deployed the same reading exercise as I did in the Fall 2020 version. Figure 5-7 shows an analysis of student responses to this exercise. More than 80 students clicked Explain within the first 30 seconds. Students who did not click Explain got the exercise correct in an average of 35.9 seconds. Approximately 70% of students clicked Explain while working on this exercise. From these results, I hypothesize that a similar fraction of students may also have clicked Explain during the Fall 2020 semester. With the new information gained by tracking the Explain button events, I can test this hypothesis in two ways in the future. One way is to run the exercise in future semesters and comparing student response trends. Another way is to look back at student sequences of submissions to the Fall 2020 exercise, and look for large differences in the sequences of Snapdown text submissions. A large difference could indicate that a student clicked the Explain button and copy-pasted the correct answer to replace an incorrect answer.

Reading 27, Little Languages I: I included the same two reading exercises here as I did in Fall 2020. Figure 5-8 shows an analysis of student responses to this exercise. More than 50% clicked Explain right away for row, and more than 35% did so for twoNotes. Students who did not click Explain got the exercises right in an average
Figure 5-6: Spring 2021, Reading 8 Snapdown exercise analysis
of 156.9 and 85.9 seconds, respectively.

To answer these questions, students had to read and understand a large code example. This code example was presented in sections in the reading itself, and the entire code was shown on a separate page. Similar to the exercises in Reading 8, students commented that this exercise involved a large amount of scrolling, and sometimes switching between pages. This led to many students clicking Explain almost immediately for the row exercise. The fact that many students clicked Explain is new information that I did not have for the Fall 2020 versions, and thus would be helpful for future iterations of this exercise.
5.2 In-class exercises

In the Spring 2021 semester, students in 6.031 completed collaborative exercises using Snapdown in pairs during class. Two such exercises were conducted: one in Class 5 on version control, and one in Class 15 on equality. The Class 15 exercise asked pairs of students to use Snapdown to draw a representation of a \texttt{Stroke} object in Java given its source code. Students were provided with the starting diagram shown in Figure 5-9; they were instructed to draw a variable \texttt{seg} pointing to a \texttt{Stroke} object, and to include \texttt{Point} objects in their diagrams, and to use the syntax (\texttt{Color `BLACK`}) somewhere. Class 15 occurred after students had already seen Snapdown in both the Reading 4 and Reading 8 exercises, and after they used it in Class 5.

Figure 5-10 shows an example of a correct diagram drawn by a pair of students. A correct diagram in this case has objects recursively nested within others: the \texttt{Stroke} object has fields pointing at \texttt{Point} objects, which themselves have fields. Such recur-
sively nested objects were presented along with corresponding Snapdown syntax in
previous reading exercises, and examples were given in the help sidebar. All syntax
examples, however, involved #-disambiguation—no examples were given that directly
included the nested objects inside their parent objects.

A total of 107 pairs of students worked on this exercise for twelve minutes. Ap-
proximately 56% of pairs drew correct diagrams. The pair shown in Figure 14 used
#-disambiguation to achieve the object nesting; 6% of pairs did so in total. The other
50% of pairs put the Point objects directly inside the Stroke objects.

Figure 5-11 shows an example of an incomplete student-drawn diagram. This
specific pair of students may have been unsure of how to achieve the desired nesting of
objects, and directly included startx and starty as fields of Stroke. Approximately
14% of pairs of students had this issue. Another 4% of pairs of students drew diagrams
without arrows between field names and objects, like start and the corresponding
Point. Introducing students to previous examples of nested objects, either through
previous exercises or in the help sidebar, may have resolved these issues.

The remaining 26% of pairs drew diagrams that demonstrated issues with concep-
tual understanding. Some pairs did not draw the names of fields such as x or y. In
general, I consider diagrams without any field names (such as x or y) or values (such
as 15 or 20) to indicate a conceptual issue.

On a final note, an instructor of 6.031 also presented Snapdown in class by using
the main Snapdown web application. The instructors gave positive feedback, noting
that compared with a tablet or whiteboard application, drawing with Snapdown

Figure 5-11: Incorrect student-drawn Stroke diagram.
Table 5.1: Analysis of diagrams from other courses. A diagram is successfully replicated if the structure and all components are preserved, possibly with visual differences in shape or layout. A diagram is nearly replicated if it can be completely replicated with one additional Snapdown feature.

| Institution - Course Name | Diagrams Attempted | Completely Replicated | Nearly Replicated |
|---------------------------|-------------------|-----------------------|-------------------|
| Wellesley College - CS 111| 35                | 35                    | 0                 |
| Cornell University - CS 2112| 13                | 10                    | 3                 |
| UCSD - OOP in Java        | 14                | 13                    | 1                 |
| Texas A&M University - CS 121| 5                 | 5                     | 0                 |
| University of Washington - CSE 143| 2             | 2                     | 0                 |
| UC Berkeley - CS 61A      | 8                 | 5                     | 3                 |
| UT Austin - CS 307        | 2                 | 2                     | 0                 |
| University of Michigan - EECS 280| 21                | 0                     | 21                |
| University of Waterloo - CS 116| 12               | 12                    | 0                 |
| MIT - 6.009               | 8                 | 4                     | 4                 |
| TOTAL                     | 120               | 88                    | 32                |

resulted in much cleaner diagrams, and took approximately the same amount of time.

5.3 Replicating diagrams from other courses

Snapshot diagrams are used at a variety of institutions, in a variety of courses, to explain key programming concepts. I evaluated Snapdown’s ability to draw diagrams presented in many courses that use diagrams throughout their lecture notes and lab assignments. I surveyed introductory- and intermediate-level programming courses from a variety of institutions. Table 5.1 contains a summary of the courses I surveyed, including the number of diagrams I replicated from each. See Appendix A for links to the sources of these diagrams. I now describe my evaluation of Snapdown on diagrams from a representative sample of these courses.

CS 2112, Cornell: I found 13 snapshot diagrams across lecture notes, 10 of which were readily expressible in Snapdown. One of the diagrams includes text written at arbitrary positions on the diagram, which Snapdown currently does not handle: the output SVG must be edited in order to achieve this affect. The other two diagrams
involved drawing a multidimensional array using a grid, and drawing a representation of memory. However, the lecture notes acknowledge that these representations are usually better visualized through object diagrams, and immediately follow them with corresponding object diagrams, which are handled well by Snapdown.

“OOP in Java”, UCSD (Coursera): I found 14 snapshot diagrams across eight lecture videos in this course. I was able to replicate 13 of them using Snapdown. The only diagram I was unable to replicate was an invalid diagram, in which a variable name had an arrow pointing at another variable name as opposed to a value. I was also able to express two sequences of three diagrams each using Snapdown’s multi-step diagram feature. I noticed that this course, among many others, prefer primitive values to be drawn directly in boxes: see Figure 5-12 for an example of this. Figure 5-13 shows how I more closely replicated Figure 5-12 in Snapdown after I added a new syntax, =, to represent assignment of primitive values to variables. Due to the modular design of Snapdown, where parsing and interpretation are decoupled from layout and drawing, I found this syntax very easy to add. This addition brought Snapdown much closer to visually replicating many diagrams not only from this course, but from others in which this visual syntax is preferred.

CS 111, Wellesley: I found 35 snapshot diagrams across two sets of lecture slides. All of them were readily expressible in Snapdown individually. CS 111 also includes a lab assignment instructing students to draw a sequence of nine snapshot diagrams that depict the state of a program at various points in its execution. I was able to draw all nine diagrams with Snapdown individually. I also tried expressing this
sequence as a multi-step diagram in Snapdown. Snapdown has syntax for appending list elements, but does not yet have syntax for more advanced control or customization of list elements. Future work includes adding syntax to support such mutations of lists and other objects.

EECS 280, University of Michigan: I did not manage to completely replicate any diagrams from EECS 280. In particular, all diagrams from this course display memory addresses for every value and every variable in the diagram. Snapdown does not support this feature, but I was able to replicate the rest of the structure of the diagram, and `Triangle::scale` and `main` as two stack frames, and the `Triangle` object on the heap. I classified all 21 diagrams from this course as diagrams I managed to nearly replicate, since Snapdown had one key missing feature.

6.009, MIT: Figure 5-15 shows an example of a snapshot diagram drawn in 6.009, the prerequisite to 6.031. Contrary to 6.031 and the previous two courses, 6.009 does not have diagrams in online readings or a textbook—these diagrams are all drawn live by instructors during class sessions. Diagrams drawn in 6.009 contain stack frames and objects similar to those in 6.031, but with considerable visual differences. 6.009 uses Python, and being an introductory course, its diagrams focus heavily on the inner workings of Python and how code executes. The diagram in figure 5-15 contains arrows pointing to stack frames, from other stack frames as well as functions on the heap. Snapdown cannot yet draw these arrows, but it can draw all other components in the diagram, modulo visual differences.
Figure 5-14: Example snapshot diagram in EECS 280, University of Michigan

Figure 5-15: Example snapshot diagram in 6.009, MIT
Chapter 6

Conclusion

While Snapdown was developed and tested within the context of 6.031, it is generalizable to a wide range of situations. The textual nature of the Snapdown language promotes usability in a wide range of user interfaces, both in the classroom and in web applications. When used in readings or course material, updating a diagram—or many diagrams at once—becomes as easy as a find-and-replace operation. During the COVID-19 pandemic, the act of physically drawing a diagram as part of a class was no longer easily possible for students. Yet with Snapdown, students were able to complete collaborative diagram-drawing exercises, and instructors were able to use it to explain concepts during class. Snapdown’s implementation produces diagrams with a consistent format and structure, thus making them easier to understand and use as a tool for communicating ideas in introductory programming courses. Finally, Snapdown’s ability to handle multi-step diagrams gives it a compelling use case in course materials and in exercises given to students. I believe Snapdown makes snapshot diagrams easier to create and maintain, and will benefit courses in which drawing diagrams is a learning objective for students.

6.1 Limitations and future work

ELK provides a number of layout algorithms—as described earlier, the ELK Layered algorithm best serves my needs. One limitation of this algorithm, however, is the lack
of support for hierarchy-crossing edges. In other words, if arrows need to point in two
directions (i.e., down for arrows between heap objects, and right for stack-to-heap
arrows), ELK Layered performs automatic layout only on arrows pointing in one of
the two directions. I worked around this limitation by using an external pathfinding
library. My approach works because diagrams with stack frames, both in 6.031 and in
other courses surveyed, are not very complex. For example, no diagram I encountered
had more than four stack frames. Future work includes exploring other algorithms or
solutions to this problem, especially if the need arises to draw more complex diagrams
involving stack frames.

Snapdown can also be extended to include many more features. As described
in Section 5.3, Snapdown currently does not have syntax to illustrate mutations of
objects across diagram frames. In addition, some courses such as 6.009 place much
more emphasis on stack frames, and have arrows pointing to stack frames to demon-
strate the inner workings of a specific programming language. Currently my design
for Snapdown is fully language-independent, but in the future, it may be benefi-
cial for Snapdown to support customized modes for specific programming languages.
Future work also includes expanding the syntax for multi-step diagrams to include
customized mutations of objects, additions and deletions of fields, and additions and
removals of stack frames. Finally, my usage of PathFinding.js is currently limited,
and can be customized further to specify heuristics, especially for edge-crossing.

I also identified issues students had with using Snapdown. For example, some
students had difficulties with drawing recursively nested diagrams, and I hypothesize
that adding more examples to the help sidebar, or more exercises illustrating this
feature of the Snapdown language, can resolve this issue. Future work includes testing
these hypotheses with targeted experiments and exercises.
Appendix A

Snapshot Diagram Sources

Below are links to websites containing diagrams from many introductory- and intermediate-level programming courses from which I chose diagrams to try to replicate. I provide one representative link per course; more diagrams can be found via adjacent or related links. I also provide a canonical course homepage for future access.

- Wellesley College, CS 111: [http://cs111.wellesley.edu/~cs111/archive/cs111_spring16/public_html/notes/lectures/08_lists1_complete.pdf](http://cs111.wellesley.edu/~cs111/archive/cs111_spring16/public_html/notes/lectures/08_lists1_complete.pdf)
  
  Course homepage: [http://cs111.wellesley.edu/](http://cs111.wellesley.edu/)

- Cornell University, CS 2112: [https://www.cs.cornell.edu/courses/cs2112/2020fa/lectures/lecture.html?id=objects](https://www.cs.cornell.edu/courses/cs2112/2020fa/lectures/lecture.html?id=objects)
  
  Course homepage: [https://www.cs.cornell.edu/courses/cs2112/](https://www.cs.cornell.edu/courses/cs2112/)

- Texas A&M University, CSCE 121: [https://github.com/michaelrnowak/tamu/blob/master/csce-121/memory_diagrams-s.pdf](https://github.com/michaelrnowak/tamu/blob/master/csce-121/memory_diagrams-s.pdf)
  
  No canonical course homepage.

- CMU, 15-122: [https://www.cs.cmu.edu/~15122/handouts/03-arrays.pdf](https://www.cs.cmu.edu/~15122/handouts/03-arrays.pdf)
  
  Course homepage: [https://www.cs.cmu.edu/~15122/](https://www.cs.cmu.edu/~15122/)

- OOP in Java (Coursera), UCSD: [https://www.coursera.org/lecture/object-oriented-java/core-drawing-memory-models-with-objects-gOF1b](https://www.coursera.org/lecture/object-oriented-java/core-drawing-memory-models-with-objects-gOF1b)
Course homepage: https://www.coursera.org/learn/object-oriented-java

- University of Waterloo, CS 116: https://cs.uwaterloo.ca/~cbruni/CS116Resources/index.php
  Course homepage: https://student.cs.uwaterloo.ca/~cs116/

- University of Washington, CSE 143: https://courses.cs.washington.edu/courses/cse143/21sp/notes/notes06.html
  Course homepage: https://cs.washington.edu/cse143/

- UC Berkeley, CS 61A: https://drive.google.com/drive/folders/1tV7Y2RQ70aSKbt7DvPn4Rs2egp2azId
  Course homepage: https://cs61a.org/

- UT Austin, CS 307: https://www.cs.utexas.edu/~scottm/cs307/handouts/deepCopying.htm
  No canonical course homepage.

- University of Michigan, EECS 280: https://eecs280staff.github.io/notes/08_ADTS_in_C++.html#adts-in-c
  Course homepage: https://eecs280staff.github.io/eecs280.org/

- MIT, 6.009: https://py.mit.edu/spring21/video?video=lec01 (skip to 1:15:55 in the video)
  Course homepage: https://py.mit.edu/
Bibliography

[1] Sebastian Baltes and Stephan Diehl. “Sketches and Diagrams in Practice”. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 530–541. ISBN: 9781450330565. DOI: 10.1145/2635868.2635891 URL: https://doi.org/10.1145/2635868.2635891

[2] Constellation. URL: http://maxg.github.io/constellation/

[3] Constrain - a JS (ES6) library for responsive, animated figures, based on declarative constraint solving. URL: https://andrewcmyers.github.io/constrain/

[4] Catherine H. Crouch and Eric Mazur. “Peer Instruction: Ten years of experience and results”. In: American Journal of Physics 69.9 (2001), pp. 970–977. DOI: 10.1119/1.1374249 Eprint: https://doi.org/10.1119/1.1374249 URL: https://doi.org/10.1119/1.1374249

[5] Andrew R. Dalton and William Kreahling. “Automated Construction of Memory Diagrams for Program Comprehension”. In: Proceedings of the 48th Annual Southeast Regional Conference. ACM SE ’10. Oxford, Mississippi: Association for Computing Machinery, 2010. ISBN: 9781450300643. DOI: 10.1145/1900008.1900040 URL: https://doi.org/10.1145/1900008.1900040

[6] diagram.codes. URL: https://www.diagram.codes/

[7] Toby Dragon and Paul E. Dickson. “Memory Diagrams: A Consistant Approach Across Concepts and Languages”. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education. SIGCSE ’16. Memphis, Tennessee, USA: Association for Computing Machinery, 2016, pp. 546–551. ISBN: 9781450336857. DOI: 10.1145/2839509.2844607 URL: https://doi.org/10.1145/2839509.2844607

[8] Eclipse Layout Kernel. URL: https://www.eclipse.org/elk/

[9] Emden R Gansner, Eleftherios Koutsofios, and Stephen North. PDF.

[10] GitUML. URL: https://www.gituml.com/

[11] Google Drawings. URL: https://docs.google.com/drawings/

[12] Graph Visualization Software. URL: https://www.graphviz.org/
[13] Patricia Heller, Ronald Keith, and Scott Anderson. “Teaching problem solving through cooperative grouping. Part 1: Group versus individual problem solving”. In: American Journal of Physics 60.7 (July 1992), pp. 627–636. DOI: 10.1119/1.17117.

[14] T. Dean Hendrix et al. “Do Visualizations Improve Program Comprehensibility? Experiments with Control Structure Diagrams for Java”. In: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education. SIGCSE '00. Austin, Texas, USA: Association for Computing Machinery, 2000, pp. 382–386. ISBN: 1581132131. DOI: 10.1145/330908.331890. URL: https://doi.org/10.1145/330908.331890.

[15] Mark Holliday and David Luginbuhl. “Using Memory Diagrams When Teaching a Java-Based CS1”. In: (Feb. 2020). URL: https://dl.acm.org/doi/abs/10.1145/971300.971373.

[16] IntelliJ UML Class Diagrams. URL: https://www.jetbrains.com/help/idea/class-diagram.html.

[17] Markdown. URL: https://daringfireball.net/projects/markdown/.

[18] Mermaid. URL: https://mermaid-js.github.io/mermaid/.

[19] Larry Michaelsen, Arletta Knight, and L. Fink. Team-based learning: a transformative use of small groups. Jan. 2002. ISBN: 0-89789-863-X.

[20] ObjectAid for Eclipse. URL: https://objectaid.com/.

[21] PathFinding.js. URL: https://qiao.github.io/PathFinding.js/visual/.

[22] Penrose. URL: http://penrose.ink/.

[23] PlantUML. URL: https://plantuml.com/.

[24] Python Tutor. URL: http://pythontutor.com/.

[25] Ulf Rüegg and Reinhard von Hanxleden. Wrapping Layered Graphs. Technical Report 1803. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2018.

[26] Ulf Rüegg et al. Minimum-Width Graph Layering Revisited. Technical Report 1701. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2017.

[27] Ulf Rüegg et al. “Using One-Dimensional Compaction for Smaller Graph Drawings”. In: Diagrammatic Representation and Inference. Ed. by Mateja Jamnik, Yuri Uesaka, and Stephanie Elzer Schwartz. Cham: Springer International Publishing, 2016, pp. 212–218. ISBN: 978-3-319-42333-3.
[28] Jorma Sajaniemi, Marja Kuitinen, and Taina Tikansalo. “A Study of the Development of Students’ Visualizations of Program State during an Elementary Object-Oriented Programming Course”. In: Proceedings of the Third International Workshop on Computing Education Research. ICER ’07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 1–16. ISBN: 9781595938411. DOI: [10.1145/1288580.1288582](https://doi.org/10.1145/1288580.1288582).

[29] Structurizr. URL: [https://structurizr.com/help/code](https://structurizr.com/help/code).

[30] SVG-Edit. URL: [https://github.com/SVG-Edit/svgedit](https://github.com/SVG-Edit/svgedit).

[31] Visual Communication Software To Make Pro Diagrams - OmniGraffle for Mac. URL: [https://www.omnigroup.com/omnigraffle](https://www.omnigroup.com/omnigraffle).