PRINCETON UNIVERSITY

CCSC
CONSORTIUM FOR COMPUTING SCIENCES IN COLLEGES
Central Plains Region

codePost

Scaling Manual Code Review with **codePost**

April 8th, 2021

Jérémie Lumbroso, Princeton University
James Evans, codePost

Kd-Trees

Grade: 0 / 20

Submission Info

Students:
Anonymized reveal

Tests

Your instructor didn't define any tests for this assignment.

Files 3

- [⌘1] KdTree.java 16
- [⌘2] PointSET.java 5
- [⌘3] tests.txt 1

```
42          double yMax = 1.0;
43
44          if (treeRoot == null) {
45              treeRoot = new Node(p, 0, xMin, yMin, xMax, yMax);
46              nItems = nItems + 1;
47          }
48          else {
49              Node pointer = treeRoot;
50              Node insertPointer = null;
51              int insertLevel = 0;
52              boolean left = false;
53              boolean horizontal = false;
54              double pX = p.x();
55              double pY = p.y();
56
57              while (pointer != null) {
58                  insertLevel = insertLevel + 1;
59                  double pointScalar;
60                  double compScalar;
61                  if (insertLevel % 2 != 0) {
62                      pointScalar = pX;
63                      compScalar = pointer.getPoint().x();
64                      horizontal = true;
65                  }
66                  else {
67                      pointScalar = pY;
68                      compScalar = pointer.getPoint().y();
69                      horizontal = false;
70                  }
71                  if (pointScalar < compScalar) {
72                      left = true;
73                      if (horizontal)
74                          xMax = compScalar;
75                      else
76                          yMax = compScalar;
77
78                      insertPointer = pointer;
79                      pointer = pointer.getLeft();
80
81                  }
82                  else {
83                      left = false;
84                      if (horizontal)
```

Grading code could be like reading an essay

Line 44 0

Just like you did above with:

```
if (p == null)
    throw new IllegalArgumentException(
```

it is often a good idea to keep your `if-else` branches as small as possible. In fact, there are two rules of thumb to strive for:

1. Try to make sure that as little code is duplicated between the `if` and `else` branch (this is usually a sign that the conditional could be rewritten as just an if-statement without an `else`).
2. Try to make sure that neither one of the branches is "lopsided"—that is, in this case, there is much too much code in your `else` branch.

Here, I would write:

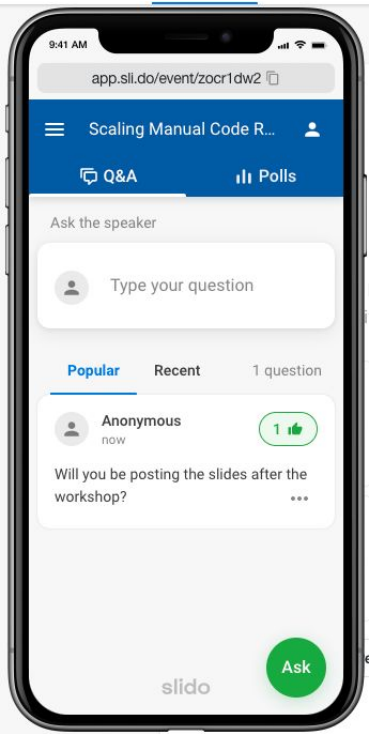
```
// base case
if (treeRoot == null) {
    treeRoot = new Node(p, 0, xMin, yMi
    nItems = nItems + 1;
    return;
}

// general case
Node pointer = treeRoot;
// ... and so on ...
```

Line 59 0

This is really excellent: You are trying to separate the specifics of the comparison from the orientation, by defining the generic variables `horizontal`, `pointScalar` and `compScalar`, and then

This workshop is **interactive**

- We want your questions!
  - You may raise your hand on Zoom
    - You will be unmuted and you will be able to ask a question
  - You may **ask your question** (possibly anonymously) on https://sli.do
    - Or **upvote** questions by others!
    - Event code is **#7481**
- We want this to be a wonderful experience for you, please speak up!



1.

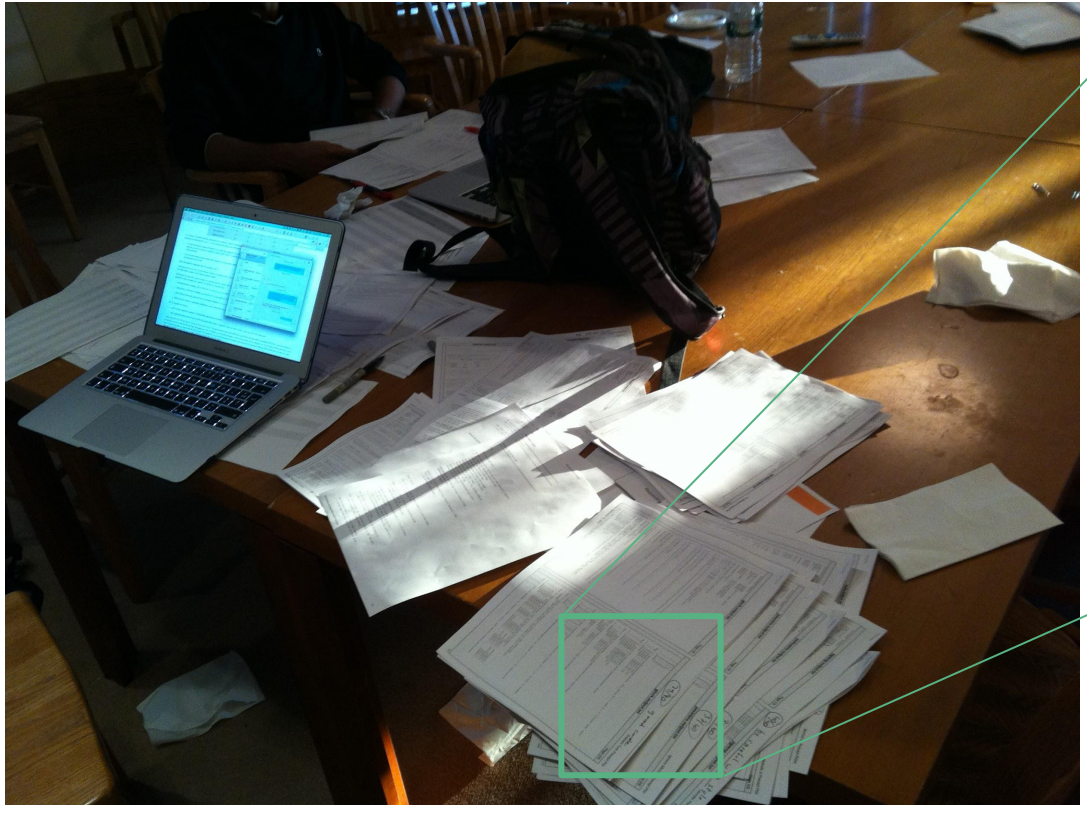
*"I don't have **the time**"*

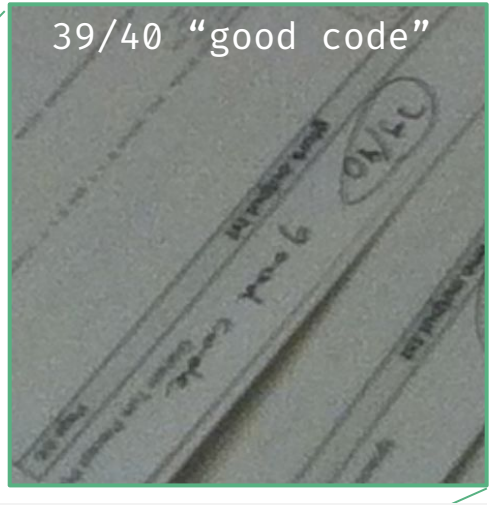
&

*"I don't have **the resources**"*

PRINCETON UNIVERSITY

CS2 grading at Princeton circa 2014 (1)





- weekly programming assignments
- assignments by Sedgewick & Wayne
- (same as Coursera Algorithms)
- ~130 students, 6 sections
- 1 instructor, 2 faculty section leaders, 3 grad TAs, 4–5 undergrad grading assistants
- expansive legacy autograders tests
  - **some exposed to students**
  - rest used for grading/diagnostic
- applied deduction, grade on 40pts
- no solution code (plagiarism!)

PRINCETON UNIVERSITY

CS2 grading at Princeton circa 2014 (2)

- Lots of paper
  - Time wasted printing
  - Tracking physical location of submission
  - Destroying old exams
- Grading
  - Applying complex rubric consistently
  - "Assessing worth of student"
  - No pedagogy, no feedback
- Many documents, tools, many authors, contradictory indications, grading as a logistical challenge

autograder output

```
...
*-----------------------------------------------------------
Running 8 total tests.

A point in an m-by-m grid means that it is of the
where i and j are integers between 0 and m

Test 1: insert n random points; check size() and
        (size may be less than n because of dupli
  * 5 random points in a 1-by-1 grid
  * 50 random points in a 8-by-8 grid
  * 100 random points in a 16-by-16 grid
  * 1000 random points in a 128-by-128 grid
  * 5000 random points in a 1024-by-1024 grid
  * 50000 random points in a 65536-by-65536 grid
==> passed

Test 2: insert n random points; check contains()
  * 1 random points in a 1-by-1 grid
  * 10 random points in a 4-by-4 grid
  * 20 random points in a 8-by-8 grid
  * 10000 random points in a 128-by-128 grid
  * 100000 random points in a 1024-by-1024 grid
```

Department of
Computer Science

Username: lumbroso

Assignment
Assignment 5

Due Date
Tuesday, March 24 2015 23:00:00

Required Files (Fixed name files marked as required by instructor)

Upload Required File Choose File No file c

PointST.java

KdTreeST.java

readme.txt

Check All Submitted Files

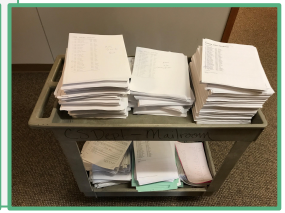
submission server

points

ctangles

rubric

```
...
* contains() / get() broken
  [ -5 get not implemented or hopelessly flawed ]
  [ -3 because of using reference equality instead of equals() ]
  [ -3 because of testing only x-coordinates, but not y-coordinates ]
  [ -3 because 2-way logic for (x < p.x) and (x > p.x) but no (x == p.x)
       common symptom = incorrect drawing for circle.txt ]
  [ -1 can't handle when root is null or other NullPointerException ]
  [ -1 not handling (xmin == xmax) ]
  [ -1 get works, but not contains ]

* range()
  [ -5 not implemented or hopelessly flawed ]
  [ -3 major flaws ]
  [ -1 if only fails when N = 0 or no points in range ]

* nearest()
  [ -8 not implemented or hopelessly flawed ]
  [ -1 if fails only when N = 0 ]
  [ -3 nearest only goes down insert path so
       answer but sure is fast! ]
  [ -3 nearest always tries left/bottom path
  [ -3 pruning is done incorrectly causing wr
  [ -2 if exception for corner case ]
...
```



"grading sheet"

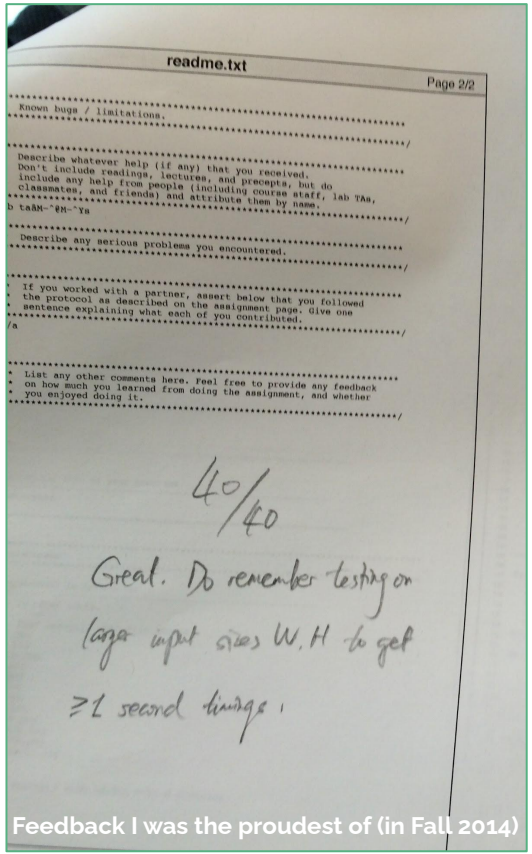
COS226 Assignment 1: Percolation Grade Report -903A

Precept: Grader:

login	Grade	style/header/login	Percolation	back	PercolationStats	readme
	40	names/whitespace/comments 5	15 API / performance	wash	10	10
	40	all good	too much logic, but it works (skynthjunker)	no	✓	✓
	40	all good	used the 2-WOF structure method	✓ but no mem	✓	✓ - rounded exponents(-1)
	35	no problems	pass all tests - no backwash	✗	very good	not enough diag points (-2)
	34	no problems	no runtime analysis / fighted corner case	✗	printed results	no runtime analysis - incorrect
	37	no problems	used int[][] (-0) - way too complex	✗		
	38	unused variable	good - failed collaboration	✗		parsed ex(-1) data structure(-1)
	38	good	good	✓	also had sticky	good small data(-1) - poor memory calculation

PRINCETON UNIVERSITY

CS2 grading at Princeton circa 2014 (3)


readme.txt Page 2/2

Known bugs / limitations.

Describe whatever help (if any) that you received. Don't include readings, lectures, and precepts, but do include any help from people (including course staff, lab TAs, classmates, and friends) and attribute them by name.

Describe any serious problems you encountered.

If you worked with a partner, assert below that you followed the protocol as described on the assignment page. Give one sentence explaining what each of you contributed.

List any other comments here. Feel free to provide any feedback on how much you learned from doing the assignment, and whether you enjoyed doing it.

40/40

Great. Do remember testing on large input sizes W, H to get ≥1 second timings!

Feedback I was the proudest of (in Fall 2014)

Problems for students

- **No/little feedback, and autograder output is laconic**
- Rubric/deductions appear arbitrary
- Since not given solution (plagiarism concerns), no improvement possible

Problems for instructors

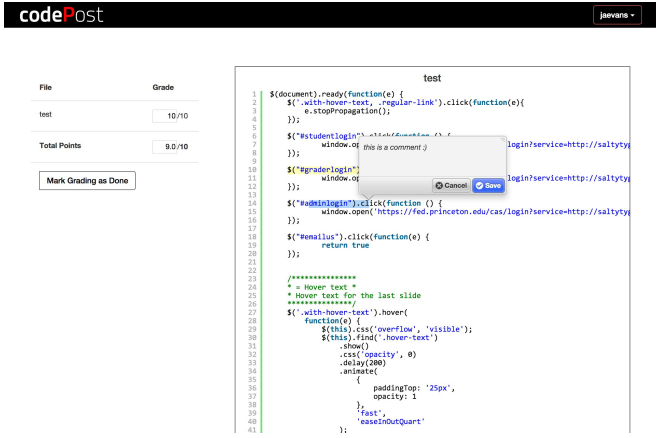
- **Bulk of time lost in logistics** (compiling, printing, assigning to graders, tracking submissions as they are graded, pregrading, entering grades in LMS, processing late submissions)
- Limited oversight of graders' work
- No/limited insights on students' work

Problem for graders (= possibly instructor themselves)

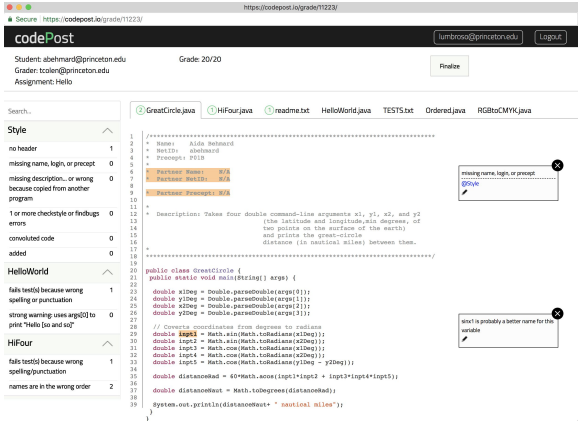
- **Bulk of time lost in repetitive work** (flipping through 5-page rubric, filling in grading sheet, adding points up, handwriting terse comments)
- Adversarial work: Find everything that is wrong with student's work
- No time to read code!!! Factory-line work
- Lots of different moving parts to master

PRINCETON UNIVERSITY

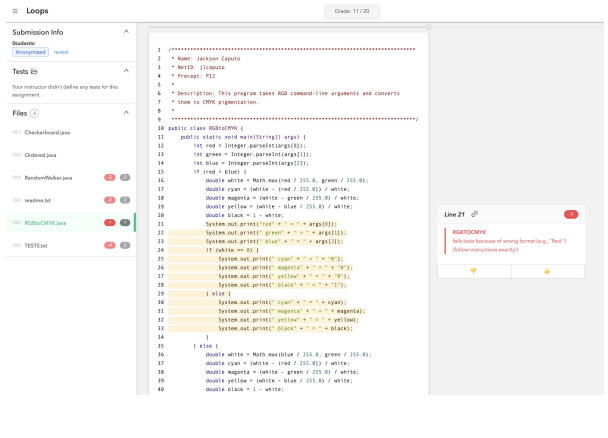
codePost



2014



2016



2019

“Resources haven’t changed but our tool and process have changed”

audience:	~120 students	**(Fall 2014)**	~300 students	**(Spring 2020)**
labor:	1 instructor, 2 co-lead faculty section leaders, 3 grad students, 4-5 undergrad grading assistants		1 lead faculty coordinator + 30-50 undergrad grading assistants	
breakdown:	5 hrs running autograder 10 hrs printing + stapling 2 hrs dispatching to graders 60 hrs grading (~6 hrs/person) 3 hrs collecting graded work 2 hrs redistributing		2 hrs preparing grading lesson 1 hrs teaching graders 30-70 hrs grading (~1-2 hrs/person) 10 hrs writing **explanations** (only once) 1-2 hrs auditing class-wide work	
total:	82 hours → ~40 min/student		35-85 hours → ~6-17 min/student	
summary:	output is a grade + handful of words time is spent moving paper around and looking through the rubric		output is appropriate assignment-targeted explanations + custom feedback on code time is spent reading code, honoring student and improving pedagogy	

Submission Info

Students:

Anonymized reveal

Tests

Category	Passed
HelloWorld	2/2
GreatCircle	6/6
RGBtoCMYK	5/5
HiFour	6/6
Ordered	10/10

Files 7


```java
public class RGBtoCMYK {
    public static void main(String[] args) {
        // Read the arguments into memory
        int r_int = Integer.parseInt(args[0]);
        int g_int = Integer.parseInt(args[1]);
        int b_int = Integer.parseInt(args[2]);
        // Print out the RGB values
        System.out.println("Red = " + r_int);
        System.out.println("Green = " + g_int);
        System.out.println("Blue = " + b_int);
        // Convert the RGB values to double form
        double r = Double.valueOf(r_int);
        double g = Double.valueOf(g_int);
        double b = Double.valueOf(b_int);

        // Calculate and print the CMYK values based on the given formulas
        double w = Math.max(Math.max(r, g), b) / 255;
        System.out.println("Cyan = " + (w-(r/255))/w);
        System.out.println("Magenta = " + (w-(g/255))/w);
        System.out.println("Yellow = " + (w-(b/255))/w);
        System.out.println("Black = " + (1-w));
    }
}
```

Line 8 0

RGBtoCMYK

While your code works it does not follow instructions.

In coding, following instructions exactly is important--it is how many coders throughout the world can collaborate on ambitious projects, the same way airplanes are assembled from parts imported from many different places that come together.

Output formats are very import in data science, computer science, coding. As you know data is incredibly valuable, it is often called the *new oil*. But data is only valuable if it is in a standardized, predictable form.

In this case, the assignment asked you to follow a precise output format:

```
red     = 75
green   = 0
blue    = 130
...
```

It is very important fo follow this format exactly. This type of situation will present itself again in the NBody assignment, and you will avoid deductions by following the output format exactly.

👎 👍

Line 12 0

A simpler way to cast integers to doubles would be:

```java
double r = (double) r_int;
```

Line 17 0

RGBtoCMYK

The term *magic number* refers to the bad programming practice of using numbers directly in your source code without explanation. In most cases this makes programs harder to read, understand, and maintain. Although most guides make an exception for the numbers zero and one, it is a good idea to define all other numbers in code as named **constants** (in Java, this is done using the `final` keyword).

This is preferable for several reasons, including:

- It is **easier to read and understand**, because the name of the constants provide information on the meaning of the value.
- It is **easier to alter the value**, as it is not redundantly duplicated across the source code, and is instead assigned to a constant in one location: So the change the value across the source code, it is only necessary to change the value assigned to the constant. Without the use of constants, changing the value of a magic number is error-prone, because the same value is often used several times in different places within a program.

In this case, you might have defined the following constant at the beginning of your program:

```java
double MAX_RGB = 255.0;
```

Submission Info

Students:

Grader:

Tests

Category	Passed
Correctness	23/23

Files 3

- [⌘1] NBody.java -0.2 5
- [⌘2] readme.txt 1
- [⌘3] TESTS.txt

Rubric

category:

Search rubric... (⌘ O)

GENERAL

off-by-one	-0.1
API violations	0
logic error	-0.2
typos	-0.1
not following instructions	-0.2

Passed	Failed	Not run	Summary
23	0	0	23/23

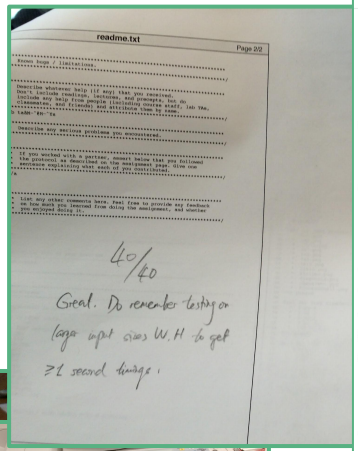
Correctness 23

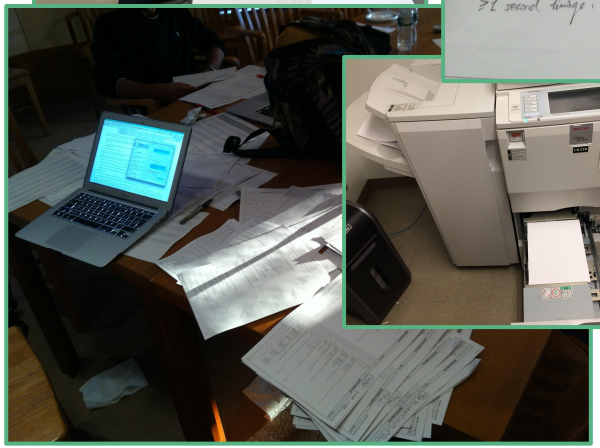
	Test Case	Explanation	Passed	Points
+	Test 1	Check that it reads data from standard input. Checks that the program reads data (necessary to take in data that will be processed). If this fails, the student is likely failing due to some other error (out of bounds or too many operations etc.). Check error messages being generated and future tests to find problem.	Passed	0
+	Test 2	Check number of calls to `stdin.readint()`, `stdin.readdouble()`, `stdin.readstring()`. Checks that the student is reading data in properly by type and amount (closely related to test 1). If this fails, most likely some other error (out of bounds etc.) is causing it to fail. Check other tests and error messages for hints.	Passed	0
+	Test 3	Check formatting of standard output. Checks the formatting of the output according to assignment specifications. If this fails, it may be due to some other error (see error messages), otherwise check the `for` loop and print statements at the bottom of the main method for improper formatting. If there are too many lines of code in the student solution, they may have placed their printing loop inside the main loop (printing each iteration of the loop).	Passed	0

21st century `code` grading toolbox

- Limit/eliminate "*manual transfer operations*" (students → submission server → autograder → printer, printer → graders, graders → …)
- **Autograder:**
  - Tries to ensure student code compiles
  - Help students avoid obvious problems; help weaker students make progress
  - Trade-off between time to write a test, and usefulness of test
- **Rubric:**
  - Provides direction to human graders
  - Helps ensure consistency of grading
- **"Explanations":** Instructor-authored paragraphs shown to students, provides bulk of quantitative feedback received—linked to rubric items
- **Custom-comments:** Individualized comments, left by graders, which both rewards students and helps address individual code problems

Rather be doing this…





API violation

Edit Preview

It seems your class has an API violation: That means maybe you made you forgot to make an instance variable `private` or that one of your helper methods, if you are using any, is not `private`.

An API, for *Application Programming Interface*, is a very important concept in programming, and has a few slightly different meanings. Here we are using it to talk about *interfaces*, which are contracts between the programmer who **creates** the class and the programmer who **uses** the class (these can or not be the same entity). Every student who implements this submission successfully creates a `MarkovModel` that could run with any other students' `TextGenerator`—because although many submissions might implement the solution differently, they all respect the same API and behave in a functionally similar way. Adding or expecting new public members breaks this interoperability.

Beyond that, the practical reason for you to respect the API provided in the assignments specifications, is that the auto-grading tests, which prepare the grading process before your submission is evaluated by a person, evaluate your submission by running your `MarkovModel` with a reference `TextGenerator`—this will help isolate issues that your `MarkovModel` might have—or running your `TextGenerator` with a reference `MarkovModel`—likewise to isolate issues to your `TextGenerator`. When your submission is running, errors in one class might snowball to another and affect your submission in many unpredictable ways—so this is why writing programs and testing them in this modular fashion allows to isolate errors that might be contained in one class but have effect on the whole program. If your submission's classes expect special methods or instance variables that you have made public but are not in our API, then your submission will not run properly on our servers, when combined with one of our classes.

… or writing this?

it is good practice to anticipate what your methods return

Edit Preview

Calling your methods in `main()` is good practice, because that way it gives a quick sanity check to make sure that your methods at least don't *crash* the Java interpreter with input parameters and a usage pattern of your choosing.

But making sure your class does not crash, is not enough! It is also good practice to do a bit more *testing*. And to do good testing (as in any good scientific experimentation), it is always important to *commit* to an outcome *before* running a test (or experiment) to make sure we are not biased in our observation of the outcome. That is, if we just accept any result from the calls to our method, we might not realize that we hold a believe that is not true.

For instance, if we have a function called `add` that prints 7, we might never find out that this function was called with parameters `add(2,2)` and its expected result was actually 4! We might also never realize that the function, in fact, always returns 7, if we don't try it with several different combinations of parameters.

Understanding how to design good test is an incredible valuable skill, because it means that you are anticipating the worst case scenarios; if you are learning how to anticipate these worst-case scenarios, that means you are preparing yourself for challenges and also that you are learning about the underlying functioning of Java and the computer.

Good testing saves (debugging) time, (avoidable) frustrations, and (preventable) deaths.

Next steps

- What is code review / code quality?
  - Why is autograding alone not sufficient?
  - Who does code review? Why is it essential?
- Preamble: Getting students to submit reviewable code
  - How to help students submit code that can be reviewed
  - What information can be extracted from a submission before human graders see it?
- Strategies for scaling code review
  - What are techniques when doing this alone (instructor alone)
  - ~~How to leverage (and quality-check) a larger staff (instructor + TAs)~~

- **Live** code**Post exercise for participants** [1 hour hands on]

2.

What is code **review** / code **quality**?

Why code review?

Some “correct” code

```
public static int dayOfYear(int month, int dayOfMonth, int year) {

    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }

    return dayOfMonth;
}
```

Two discussion questions:

- What is wrong with this code?
- What tests could you write to detect these problems?

Source: https://web.mit.edu/6.005/www/fa15/classes/04-code-review/

Why code review

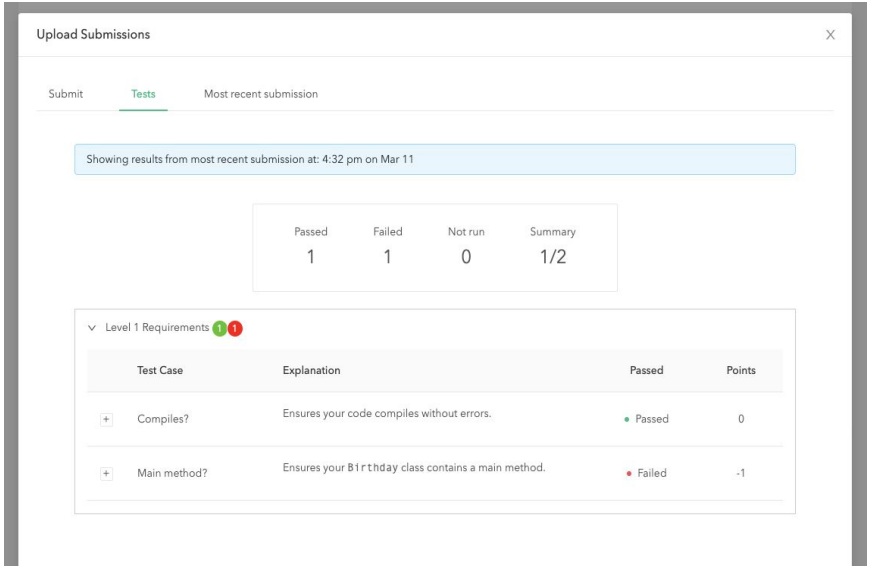
- Code review is ubiquitous in industry
  - Helps ensure code hygiene: maintainability, human-readability. Correct code != good production code
  - Allows for discussion and triage of correctness issues
- Case study:
  - At codePost, ~25% of development time is dedicated to code review
  - Important but rarely taught skills we focus on:
    - Assuming someone other than the original author will maintain the code you write
    - Writing specific, actionable comments about others' code
    - Reacting constructively, not defensively to suggestions about code, even correct code

What makes code especially hard to review?

- Code that doesn't compile or contains syntax errors
  - This code will fail all automated tests
  - Debugging this code (by finding the errors) can be extremely labor-intensive, crowding out more meaningful feedback
- Code that doesn't adhere to a specified API
  - Failed tests might not expose bugs
  - Harder to explore the code by stepping outside pattern recognition developed from other submissions
- Code with wacky style
  - Extra long lines, huge blocks of code, bad indentation, etc, make reading code tedious

Making code review easier

- One way to avoid this type of code: incentivize students to submit “reviewable” code
  - Feedback loop: Create automated tests to check for the above symptoms, and expose these tests to students at the point of submission
  - Gamify: Group these tests into a group called “Level 1 requirements” (or something to indicate that they represent the most basic requirements)
  - Incentivize: Attach point values to these tests



Level 1 requirements exposed to students in codePost

3.

Personalized feedback workflows

Personal Feedback Workflow: Disclaimer

This section will be concrete efficient personal feedback workflow:

- **techniques** for instructors alone
  - these readily transfer to a group/distributed setting
- ~~and how to leverage (and **quality-check**) a larger staff (instructor + TAs)~~

but all examples are based on my workflow in Princeton's CS1:

- 300 submissions
- I manage 30-70 undergraduate grader over a period of 1-3 hours
- the main advantage is parallelization and speed, but this could be done with a smaller number of full-time TAs

An important distinction

In codePost, there are two complementary notions for comments:

- **Rubric comments** belong to a rubric
  - instantiated by the graders
  - everything about them controlled centrally (and **retro-actively**) by instructor:
    - grader description,
    - student explanation,
    - point delta
  - they also contain a small part that is filled in by the grader (the customization of the comment)
- **Custom comments** are discretionary comments left by graders

Notions are important both for quality control and for scale efficiency

This is a rubric comment

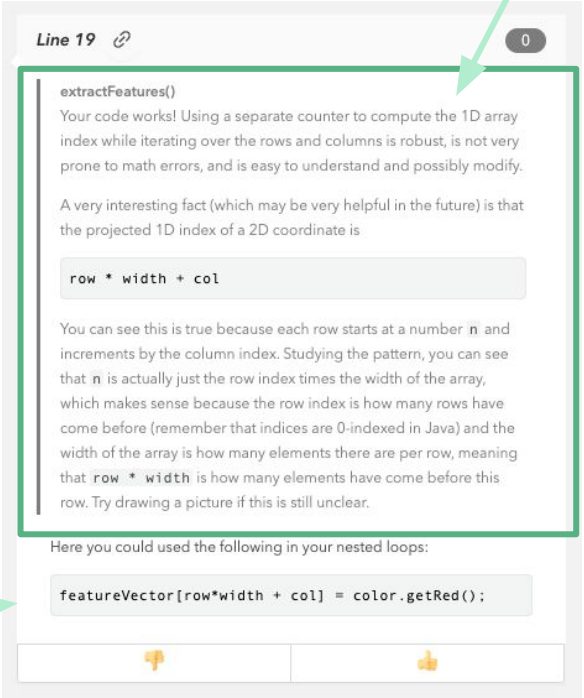
grader[-facing] caption (written once)

student[-facing] "explanation" (written once)

What the **grader** typically sees:

"customization" (written by grader, each time comment is applied)

What the **student** sees:



Individual Scenario:

Grading exam or new assignment

no existing rubric

single instructor doing the grading

Broad outline

To grade the assignments, you can follow three steps:

"*Tag First, Explain Later*"

- **Step 1:** Grade submissions, and create the rubric as you go using the in-line collaborative rubric feature (but alone)
- **Step 2:** Once you have *tagged* your submissions, your explore your data set, and use the combined examples to help you write an explanation for each rubric item.

"*Iterative Rubric Creation*"

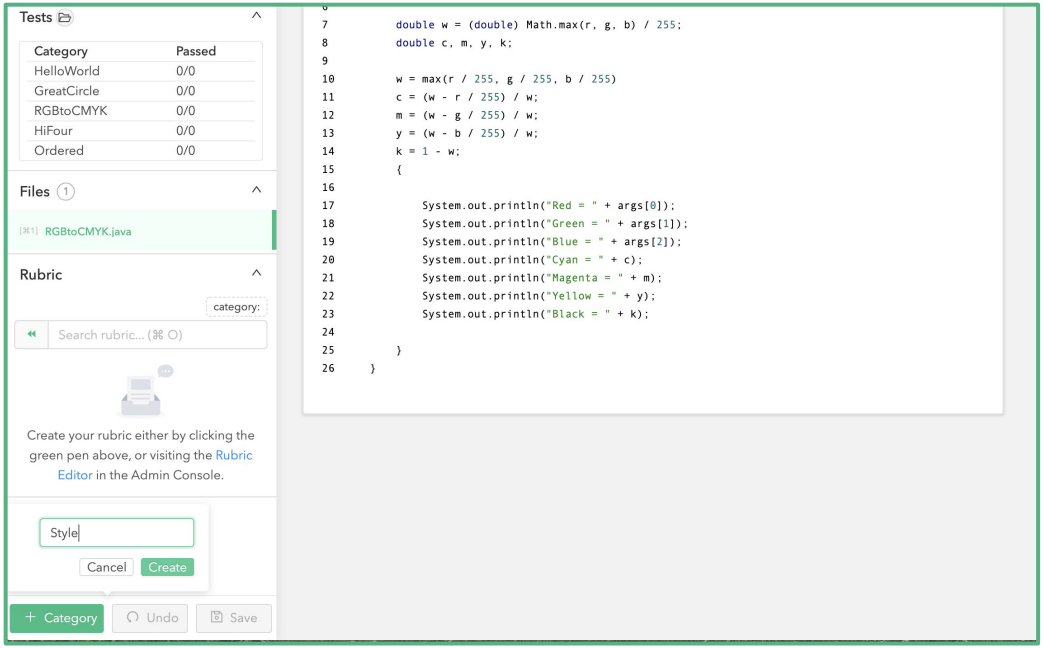
- **Step 3:** If you left custom comments in your submissions, you may audit them to see if you can merge some to become rubric comments

The rubric is the

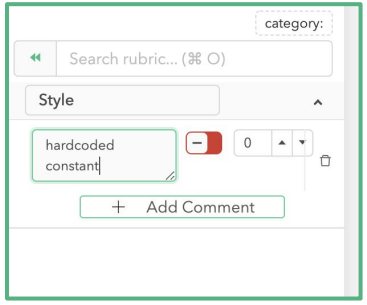
Step 1: create rubric

- As you go along, you can either
  - add custom comments (if you think comment is unique)
  - create a rubric comment as described here

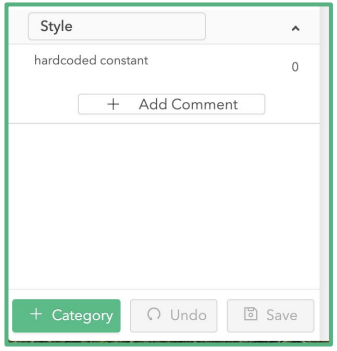
- This will build the rubric for your assignment and keep every submission linked to the corresponding rubric items



1.



2.

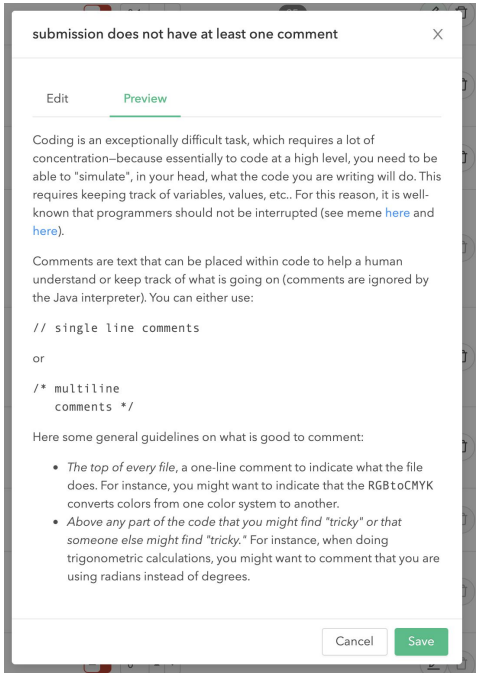


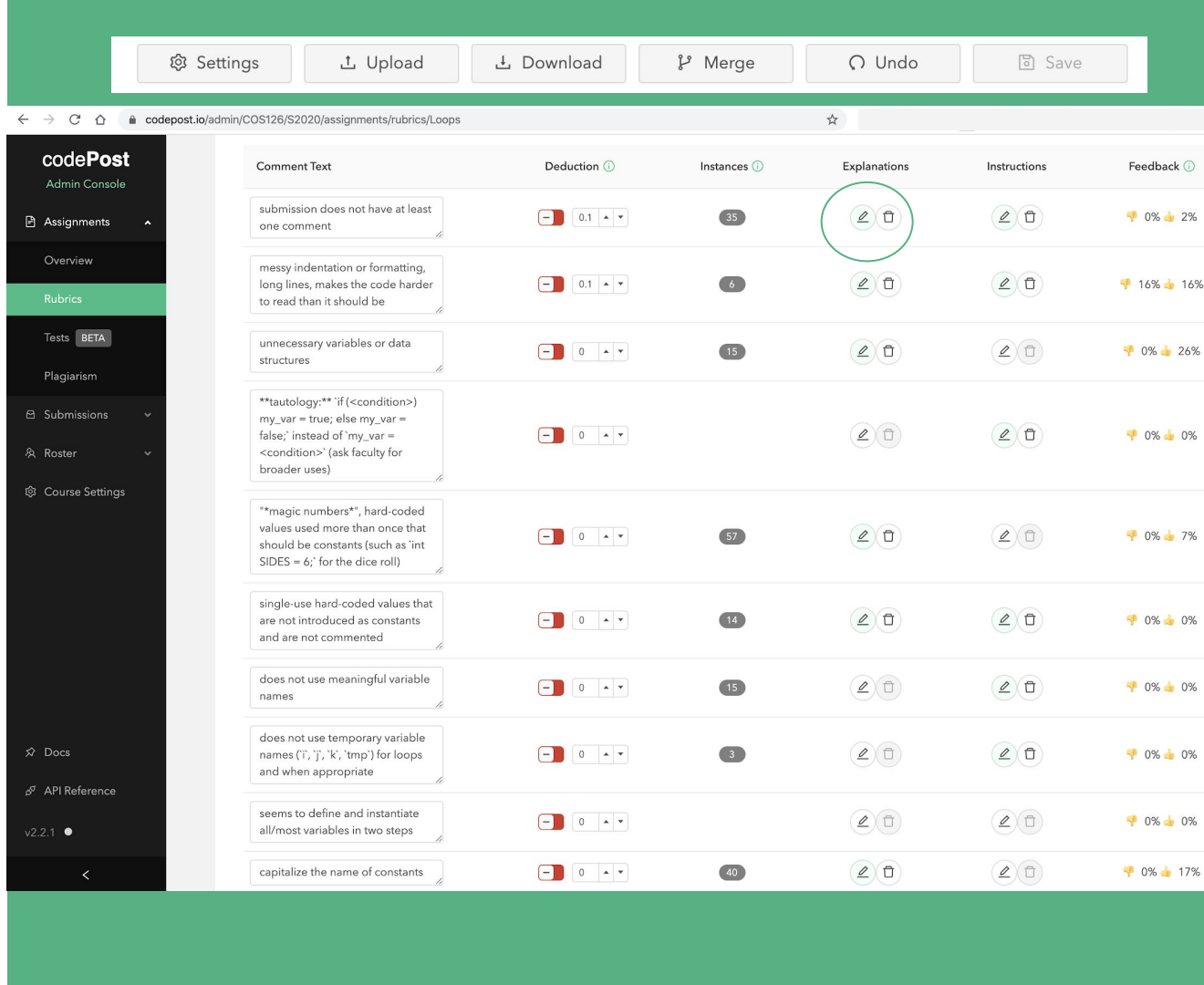
3.

Step 2: Explain!

Add explanations to rubric items; adjust deductions

Have fun and go crazy! You won't ever have to do it again

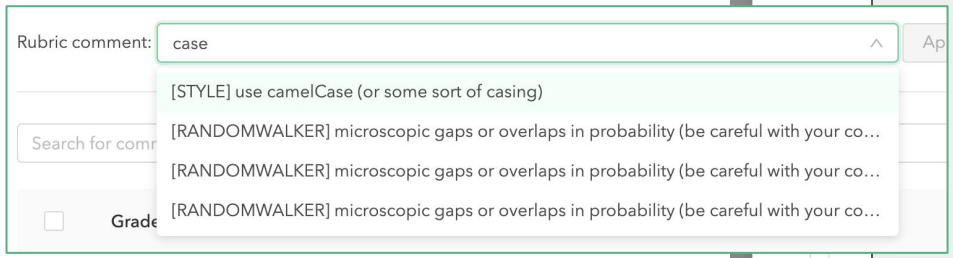


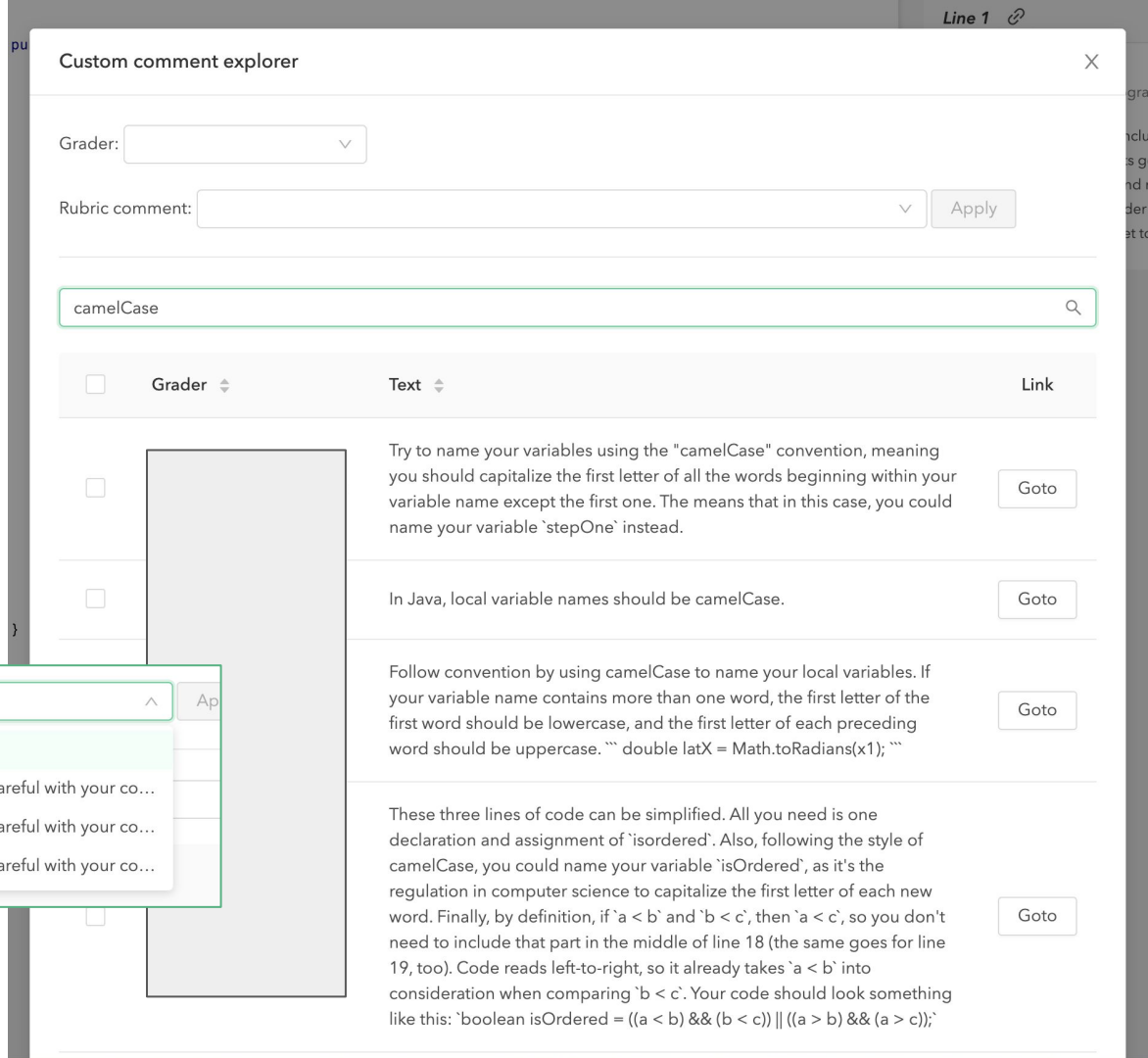


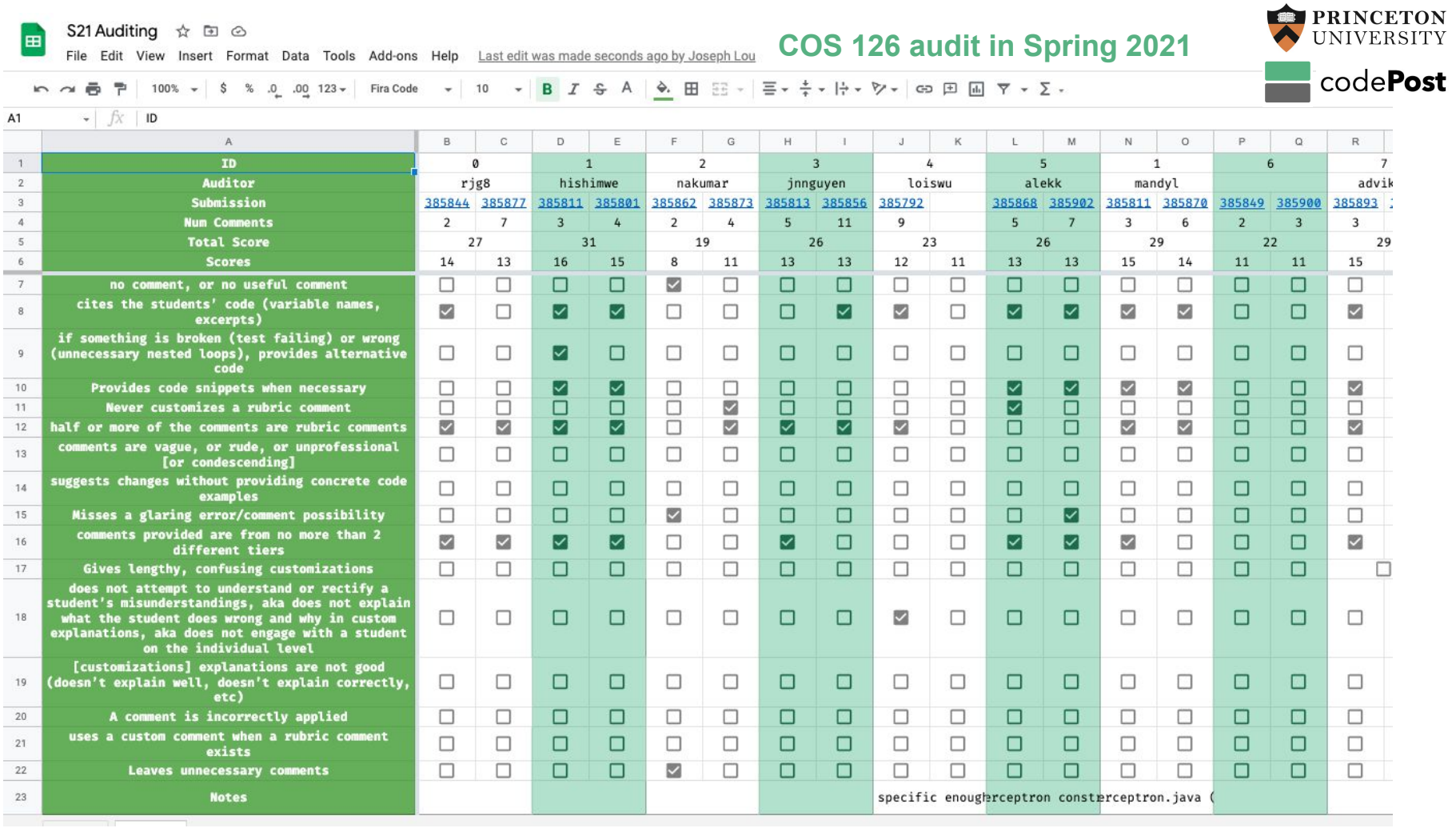
Step 3: Audit

You can audit the custom comments after grading

- to make sure some shouldn't be rubric comments instead (consistency)
- to see if there are similar custom comments that would suggest creating a rubric comment (efficiency)







	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	ID	0		1		2		3		4		5		1		6		7
2	Auditor	rjg8		hishimwe		nakumar		jnnguyen		loiswu		alekk		mandyl				advik
3	Submission	385844	385877	385811	385801	385862	385873	385813	385856	385792		385868	385902	385811	385870	385849	385900	385893
4	Num Comments	2	7	3	4	2	4	5	11	9		5	7	3	6	2	3	3
5	Total Score	27		31		19		26		23		26		29		22		29
6	Scores	14	13	16	15	8	11	13	13	12	11	13	13	15	14	11	11	15
7	no comment, or no useful comment	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
8	cites the students' code (variable names, excerpts)	☑	☐	☑	☑	☐	☐	☐	☑	☑	☐	☑	☑	☑	☑	☐	☐	☑
9	if something is broken (test failing) or wrong (unnecessary nested loops), provides alternative code	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
10	Provides code snippets when necessary	☐	☐	☑	☑	☐	☐	☐	☐	☐	☐	☑	☑	☑	☑	☐	☐	☑
11	Never customizes a rubric comment	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐
12	half or more of the comments are rubric comments	☑	☑	☑	☑	☐	☑	☑	☑	☑	☐	☐	☐	☑	☑	☐	☐	☑
13	comments are vague, or rude, or unprofessional [or condescending]	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
14	suggests changes without providing concrete code examples	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
15	Misses a glaring error/comment possibility	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐
16	comments provided are from no more than 2 different tiers	☑	☑	☑	☑	☐	☐	☑	☐	☐	☐	☑	☑	☑	☐	☐	☐	☑
17	Gives lengthy, confusing customizations	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
18	does not attempt to understand or rectify a student's misunderstandings, aka does not explain what the student does wrong and why in custom explanations, aka does not engage with a student on the individual level	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐
19	[customizations] explanations are not good (doesn't explain well, doesn't explain correctly, etc)	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
20	A comment is incorrectly applied	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
21	uses a custom comment when a rubric comment exists	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
22	Leaves unnecessary comments	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
23	Notes									specific enough		erceptron const		erceptron.java (				

Staff Scenario:

Grading existing assignment

pre-existing deductive rubric

instructor **with staff of TAs**

COS 126 ASSIGNMENT 0 -- **"HELLO, WORLD"** RUBRIC **Spring** 2020

GIVING FEEDBACK

- Grading is an opportunity for you to communicate with students, not just evaluate them. Deducting points is easy, giving *constructive feedback* is hard!
- When a student loses points, what they want to know is what, specifically, they should do differently next time. The deductions in codePost are a **starting point** to be augmented for each student. For errors that will pop up again, you can say *"Next time, ..."*. For specific bugs, you can point them out, e.g. for Ordered, you could say "*using < instead of <= here would catch these cases*". Aim for useful, actionable, specific.
- Before you write your feedback, think about *what kind of student* you're speaking to.
  - If they're failing all the tests, do not comment on style.
  - If they seem to be misunderstanding a concept, like booleans, teach it to them.
- Read their readmes. Respond to something, if you find it funny/interesting. ***If you find something concerning, contact a faculty member.***
- Look for opportunities to praise a student's work: we don't want students to receive just a 20/20 score with no feedback.

DEDUCTIONS & GRADE

- We will demo ***codePost*** and all its awesome features!
- Your deductions will be auto-capped, no more than the max points per file.
- We have several "-0" deductions, they mean we will deduct next time.
- The grade gets calculated automatically.

GENERAL

- Missing files, deduct 100% of the points on that part.
- Files that don't compile, confer with a faculty member.

STYLE

COMMENTS

-0: no comments at all on all the code

CONVOLUTED CODE

-0: code is exceptionally hard to read because of all-over-the-place indentation, confusing/misleading variable names, or too many wrapped long lines

HELLOWORLD (0.5 POINTS)

-.1: fails test(s) because wrong spelling or punctuation (follow instructions exactly!)
-0 strong warning: uses args[0] to print "Hello [so and so]" **(follow instructions!)**

HIFOUR (0.5 POINTS)

-.1: fails test(s) because wrong spelling/punctuation (follow instructions exactly!)
-.2: names are in the wrong order

ORDERED (0.5 POINTS)

-.2: only checks for ascending order or descending order, not both (logic error)
-.2: fails some/all tests when inputs are equal (logic error)
-.2: fails some/all tests when inputs are negative (logic error)
-.1: fails test because parses args as doubles, not as ints (follow instructions exactly!)
-.1: not storing result in a boolean (follow instructions exactly!)
-0: fails tests due to integer overflow, something like: x - y < 0 or (x - y) * (y - z) > 0
-0: uses & instead of && or | instead of || -- we'll learn about bitwise operators later!
-0: redundant checks or if (something) {x = true;} else {x = false;}

GREATCIRCLE (0.5 POINTS)

-.1: does not convert to radians before using Math.sin()/cos() (follow instructions!)
-.1: error in formula
-.1: prints only the numerical result but no units (follow instructions exactly)
-0: tries to print units, but spelling error or missing space, e.g., "nauticalmiles"

RGBTOCMYK (1 POINTS)

-.2: integer division error when calculating C, M, Y, or K
-.2: error in formula (e.g., order of operations issue, or **225** instead 255, etc.)
-.1: missing output for C, M, Y, or K (could be hiding a bug!)
-.1: order of output incorrect / extra / missing values for R, G, or B
-.1: fails tests because of wrong format (e.g., "Red:") (follow instructions exactly)
-.1: not using Integer.parseInt() to read in RGB values (follow instructions exactly)
-0: using both Integer.parseInt() and Double.parseDouble() on same arguments
  - warn students, they will need to understand casting!
  - implicit casting, also called promotion, for example R/255.0, is fine with us!
-0: comparing doubles with == (caution, doubles can be imprecise!)

README (1 POINTS)

-.5: missing all identifying info
-.5: missing all questions after identifying info
-0: does not list exam dates

Rubrics for COS126

Dan Leyzberg and course staff

- deductive
- on 4 pts
- (same normalization as exams)
- roughly correspond to certain learning objectives

assuming this can't be changed (time, hierarchy, legacy, etc.)

We will show how to apply and give feedback with team of TAs

Context

The rubric has been entered for the staff of graders to use:

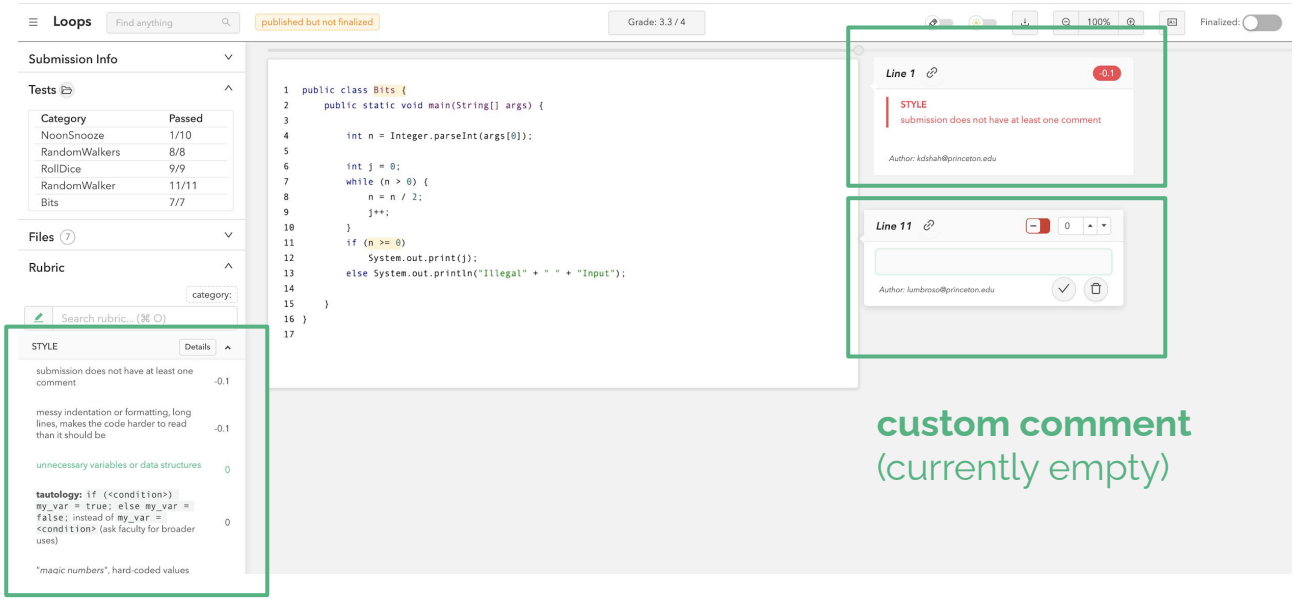
- They can apply the rubric comments, and optionally add their customization
- They are encouraged to provide personal feedback as custom comments

We have already shown how to audit custom comments, but rubric comments can also be checked

Step 1: Applying comments from the rubric

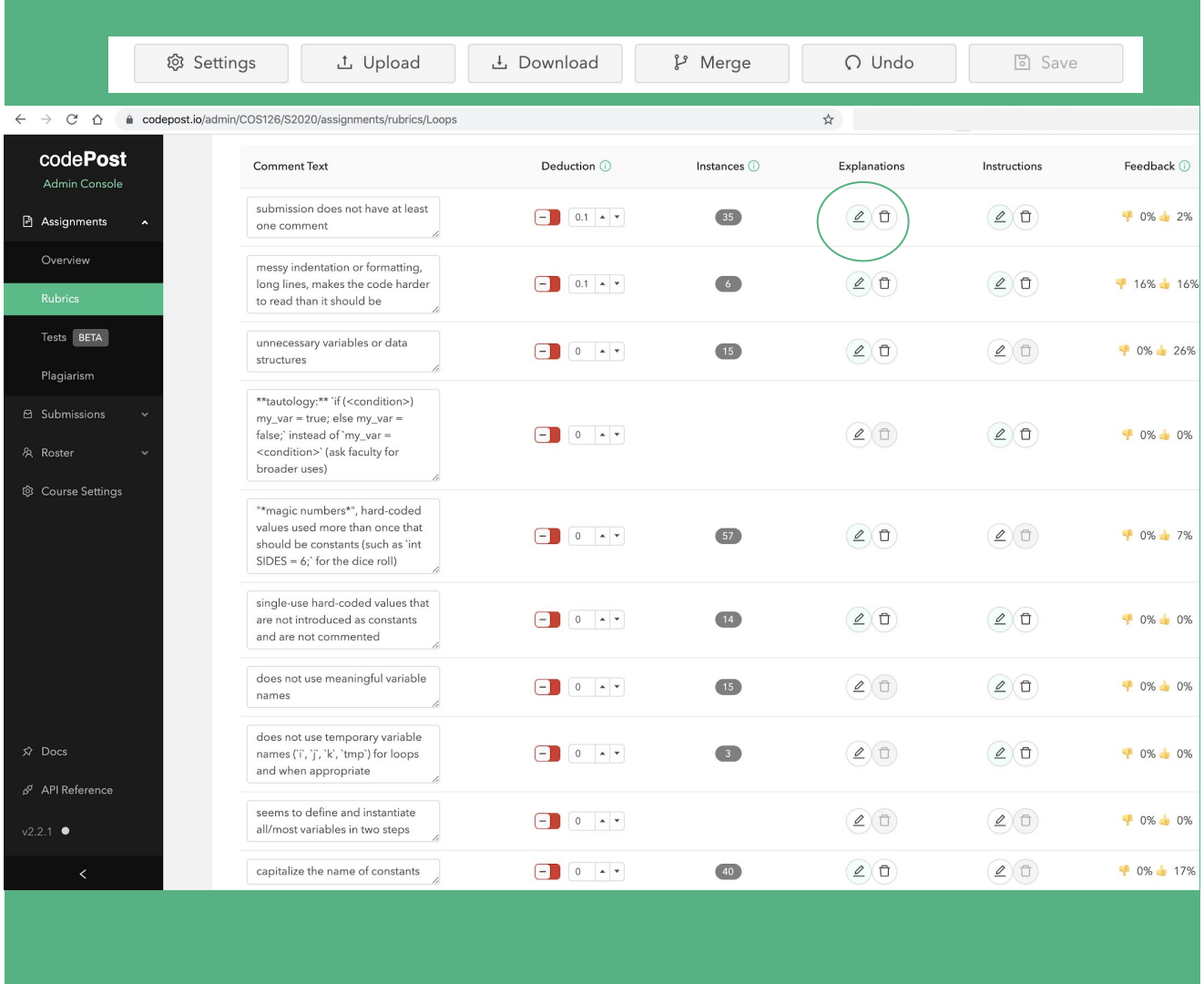
When you have a rubric predefined, it appears (with grader-specific captions if available) and is ready to be applied

rubric comment
(without customization)



rubric window

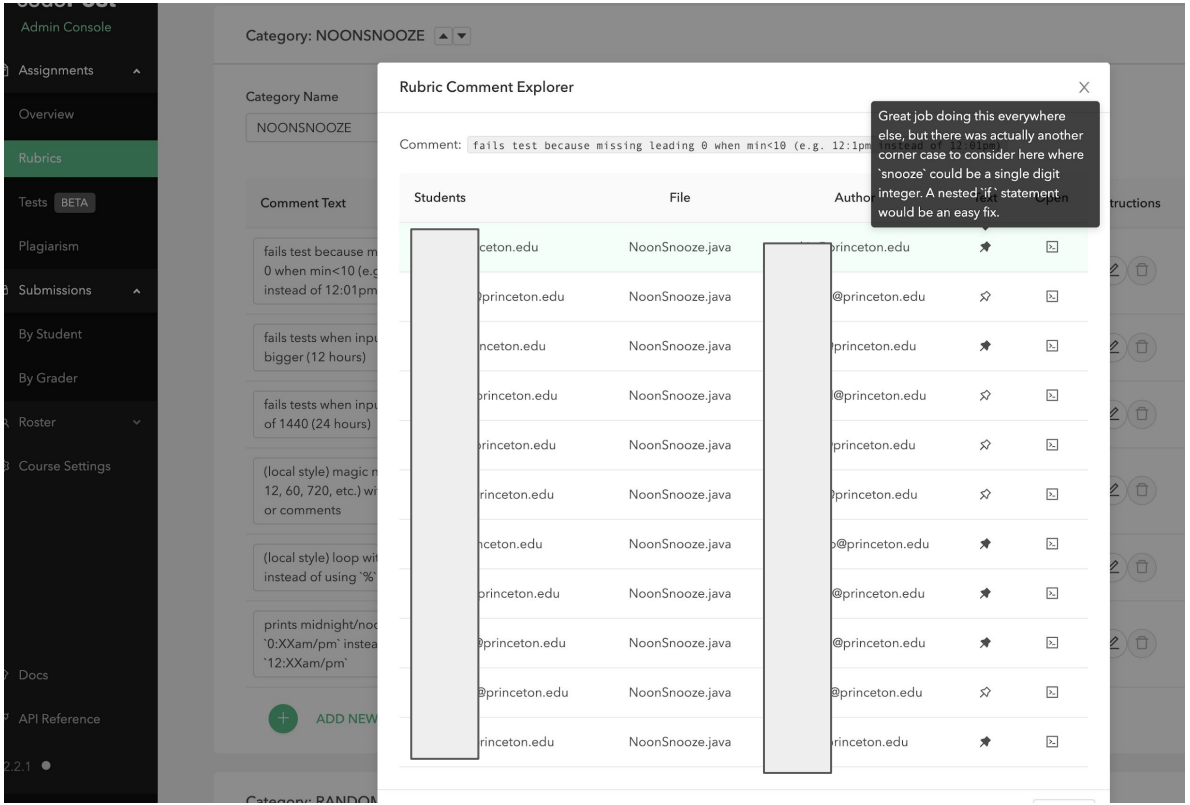
custom comment
(currently empty)



RUBRIC EXPLORER

Step 2: Exploring

- Explore every application of each rubric comment
- Able to look how this rubric item was applied
- Can be used to write explanations, and to audit graders



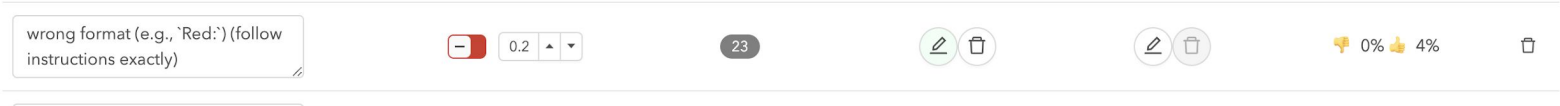
Bonus miscellaneous

Mining the Rubric Dataset

using the scale of
your class in your favor

Iteration via student feedback

- Improve your rubric by soliciting feedback from students
- Things to catch:
  - Unclear explanations
  - ....



- Bonus: use last year's data to improve this year's teaching
  - Distribution of rubric comments (combined with comprehension scores) can point to learning breakdowns => can tweak curriculum
  - Can leverage previous applications of rubric comments to train new staff (and students!)

Ensure fairness

- What does fairness mean for grading?
  - Avoid conflicts of interest
  - Consistent scoring
- Avoid conflicts of interest with anonymous grading mode
  - Added benefit of removing unconscious bias from grading process, besides explicit conflicts of interest
- Consistent scoring
  - Much easier to adjudicate if TAs are grading random submissions: otherwise, you may need to account for systematic deviations in submission quality by TA
  - Data to assess fairness across TAs:
    - Average score awarded
    - Average score awarded, normalized for automated test failures
    - Frequency of rubric comment usage

Ensure quality

- Hard problem: what makes a good code review?
  - Feedback quantity: lots of comments
  - Feedback quality: specific, actionable, reference student code, use rubrics
- How to enforce:
  - Rubric-only mode: in this mode, graders can't create custom comments, and are instead forced to use the rubric.
  - Instruction text: nudge graders to personalize rubric comments in specific ways.
- How to measure
  - {insert section on codePost API}

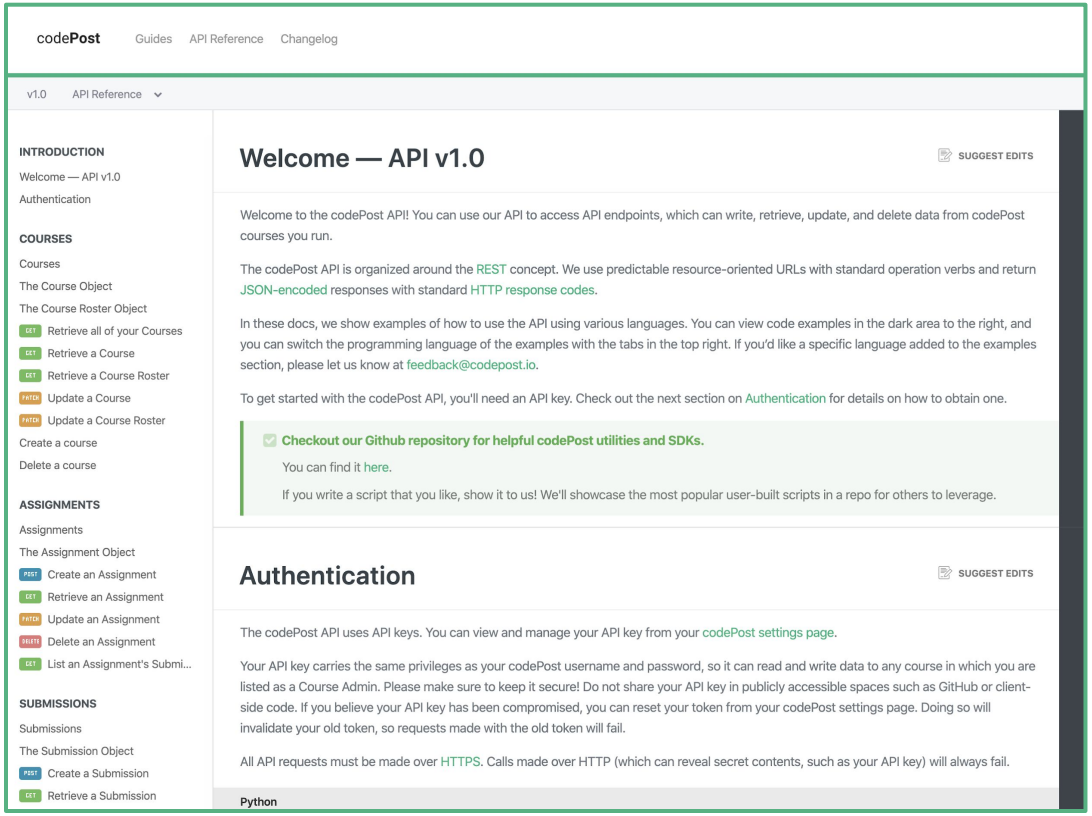
4.

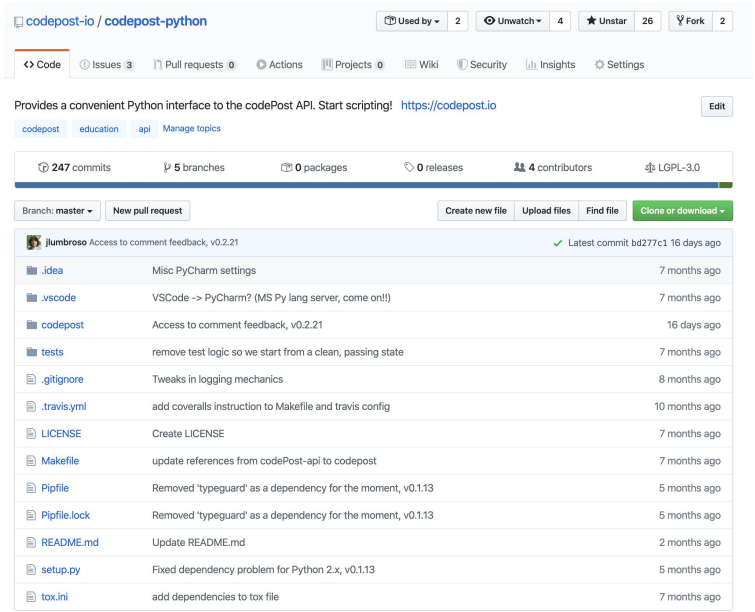
Live exercise for participants
facilitated by James Evans

5.

API, **SDK** and beyond

codePost has an open API and a Python SDK





Dataset of the comments


```
{
  "assignment": {
    "id": 2763,
    "name": "Hello"
  },
  "submission_id": 122350,
  "comment_id": 285902,
  "grader": "xxxxxxxx@princeton.edu",
  "point_delta": 0.0,
  "rubric_comment": null,
  "feedback": 0,
  "comment": {
    "code_blobs": [
      {
        "language": "java",
        "code": "\nboolean isOrdered = ((a < b) && (b < c)) || ((a >
b) && (b > c))\n"
      }
    ],
    "content": "you can declare and initialize the boolean in one
statement:\n```\nboolean isOrdered = ((a < b) && (b < c)) || ((a > b)
&& (b > c))\n```",
    "length": 133,
    "wordcount": 30
  },
  "location": {
    "filename": "Ordered.java",
    "extension": ".java",
    "start_line": 5,
    "start_column": 0,
    "end_line": 6,
    "end_column": 65
  },
    "tests": {
    "total": 29,
    "passed": 28,
    "failed": [
      3609
    ]
  },
  "variables": {
    "file": [
      "args",
      "b",
      "isOrdered",
      "a",
      "c"
    ],
    "comment": [
      "isOrdered",
      "a",
      "c",
      "b"
    ],
    "coincidence": [
      "b",
      "isOrdered",
      "a",
      "c"
    ],
    "overlap": true
  },
  "indicators": {
    "uses_rubric_comment": false,
    "uses_code": true,
    "uses_learner_tokens": true
  },
  "statistics": {
    "ratio_code": 49.62406015037594,
    "ratio_test_passed": 0.9655172413793104
  }
}
```

THANK YOU

to you +

to the organizers of SIGCSE 2020 and board