1. (10pt) **Name:** ____

INSTRUCTIONS: Carefully read each question, and write the answer in the space provided. If answers to free response questions are written obscurely, zero credit will be awarded. The correct answer to a free response question with a short answer (i.e., one word or phrase) will never contain any significant words used in the question itself (i.e., "crossword rules"). You are permitted to use one 8.5x11 inch sheet of paper (double-sided) containing **hand-written** notes; all other aids (other than your brain) are forbidden. Questions may be brought to the instructor. You are required to turn in your notes with your exam; they will be returned to you with your graded exam.

You have **80 minutes** to complete the exam. You may turn in your exam as soon as you are finished. There are **800 points** available on the exam, and most questions are worth a number of points that is divisible by ten. These point values are a rough guide to how long I think you should spend on each question.

For **TRUE** or **FALSE** and multiple choice questions, circle your answer.

On free response questions only, you will receive **20%** credit for any question which you leave blank (i.e., do not attempt to answer). Do not waste your time or mine by making up an answer if you do not know. (Note though that most questions offer partial credit, so if you know part of the answer, it is almost always better to write something rather than nothing.)

You may rip out any page in this exam. If a page has questions on it, make sure your UCID is on it before you separate it from the rest of your exam.

To get credit for this question, you must:

- Print your name (e.g., "Martin Kellogg") in the space provided on this page.
- Print your UCID (e.g., "mjk76") in the top-right of **each** page of the exam with a question on it.

Contents (blanks for graders only):

Writing your name and UCID:	10 / 10
Pages 2 and 3:	340 / 340
Pages 4 and 5:	150 / 150
Pages 6 and 7:	300 / 300
Extra Credit:	60 / 0
Total:	860 / 800

I. Multiple Choice and Very Short Answer (180pts). In the following section, either circle your answer (possible answers appear in **bold**) or write a very short (one word or one phrase) answer in the space provided. No partial credit is possible in this section.

2. (20pt) **TRUE** or **FALSE**: the daily standup meetings in an Agile process are a form of toil.

3. (20pt) Which of these words best describes a test that fails non-deterministically?

   A flaky

   B hermetic

   C isolated

   D brittle

4. (20pt) **TRUE** or **FALSE**: if you create a derivative work of an open-source library with a copyleft license like the GPL by enhancing it with a new feature, you can release your version of the library under a more-permissive license, such as an MIT license.

5. (20pt) You have to pay special attention to your **monitoring** infrastructure when operating a service, because without it you can't even tell if your service is working.

6. (20pt) In the **imperative** programming paradigm, programs destructively update state; in the **functional** paradigm, programs instead yield new states over time.

7. (20pt) Which of these should the author of a code review do before or during the review (select all that apply)?

   A write a response to each reviewer comment in the code review tool

   B respond to all reviewer concerns by changing the code or the comments

   C make sure the diff is clean before asking for a review

   D review it themself and fix any obvious problems before asking someone else to review it

8. (20pt) A non-functional requirement that is **verifiable, falsifiable, testable, or any synonym thereof**, such as "the system shall be available at least 99.9% of the time," is superior to an informal requirement like "the system shall be highly available."

9. (20pt) Which of the following are **mistakes** that one could make during a whiteboard technical interview (select all that apply)?

   A solve the problem as quickly as you can, to show how smart you are

   B always program in the language that the company uses internally, to show you care

   C brute force the solution first, and then optimize, to show you can improve your code

   D ask questions about the problem, to show that you can perform requirements elicitation

10. (20pt) **TRUE** or **FALSE**: you should avoid committing binary files to your version control system because the VCS records the *differences* between files, and two different versions of a binary might be completely different even if they're logically related.

A. differential testing	**B.** playbook	**C.** code smell	**D.** cathedral model	**E.** factory pattern
F. A/B testing	**G.** sprint	**H.** user story	**I.** actionable	**J.** model-view-controller
K. blameless	**L.** linter	**M.** logging	**N.** mutation testing	**O.** microservice
P. observer pattern	**Q.** opinionated	**R.** waterfall	**S.** bazaar model	**T.** dataflow analysis

II. Matching (160pts). This section contains a collection of terms discussed in class in an "Answer Bank" (choices **A.** through **O.**). Each question in this section describes a situation associated with an answer in the Answer Bank. Write the letter of the term in the Answer Bank that best describes each situation. Each answer in the Answer Bank will be used at most once.

11. (20pt) **S. bazaar model** Andy participates in a democratic organization that governs an open-source software project that he contributes to regularly.

12. (20pt) **A. differential testing** When refactoring her program, LaMonica makes sure that her changes are semantics-preserving by comparing the outputs of the refactored program to the outputs of the original for a set of randomly-generated inputs.

13. (20pt) **O. microservice** Cory's team is small and organized around a single, well-defined business capability.

14. (20pt) **C. code smell** While refactoring, Frank fixes many small, individually inconsequential issues that together impede maintenance.

15. (20pt) **K. blameless** After an outage, Thomas' postmortem explaining how he will personally do better in the future is returned to him for correction by one of his senior peers.

16. (20pt) **L. linter** An automated tool warns Nellie about a style problem in her code before she commits it.

17. (20pt) **H. user story** To make sure that his project's specification is aligned with customer requirements, Rob writes down scenarios about how he expects his customers to interact with his software.

18. (20pt) **P. observer pattern** Because she expects that an object she is designing will eventually have many dependencies, Bonnie defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified.

III. Short answer (120pts). Answer the questions in this section in at most three sentences.

19. (60pt) Suppose that you are a software engineer at NVideo, a large online video sharing platform. You are on-call during an incident triggered by a code change pushed by one of your coworkers. Your team always uses a staged deployment strategy wherein the amount of traffic that is served by the new version of the software increases by a factor of e every five minutes. Your team also has automated monitoring and rollback that will contain this bug once 10,000 failures have been detected. Suppose that it takes your automated rollback system 12 minutes to detect and completely rollback the failure. Your monitoring system reports that in the last full minute before the fix was deployed, about 4,000 requests failed.

    Support or refute the following claim about this situation: you do not need to take any additional action to contain this outage. **Must refute. The equation $CU = RK$ governs the order of a fault (U), and the question provides all of the other variables: C is 10,000; R is 4,000, and K is five minutes. Plugging it in, we find a fault with approximately order of 2, which indicates that each failed request is damaging data that will be used by some future request, causing another request to fail. We need to investigate to determine what data was damaged and revert the system's *state* (e.g., the database), not just the code. Partial credit: 40 points for realizing to use $CU = RK$ but making a mistake with the math or units, or for general caution around incidents. 20 for saying nothing to do at this time, but saying something reasonable about next steps, such as "write a postmortem" or "preserve logs".**

20. (60pt) Suppose that you are a software engineer at EasyTwo, a cloud-services startup, on a team supporting the system for users to manage their instances (i.e., their cloud servers). This system supports a variety of operations: creating new instances, stopping old instances, automatically creating duplicates of existing instances, etc. At first, your company only had one kind of instance, because the servers in your datacenter were all identical. However, shortly before you joined the team, the company made another large purchase of a different type of server, and your coworkers added an option to allow users to pick which kind of server to use for new instances. Now, your hardware team informs you that they plan to purchase a third type of server, and asks your team to add support for it, as well. Your teammates are concerned that this will be a large job: there are a lot of places in the codebase where new instances are created, and your team will need to add a new branch to the `if` statement between the two existing server types to each. Is there a better way to handle this situation that you can suggest that would make it easier to add new server types in the future? **You should suggest using the factory pattern. Your application can ask the user for their desired server type in one place, and then initialize the appropriate factory. The `if` statements can all be replaced with calls to the factory's method that creates a new instance. Half-credit for saying OOP ideas ("liskov SP" or similar) or saying refactoring without mentioning a pattern. Full credit for abstract factory or named constructor patterns. "Use a design patter" without naming which one: 20 points.**

A. *Software Architecture*
B. *Verifying Object Construction*
C. *A Field Study of Refactoring Challenges and Benefits*
D. *Machine Learning: The High-Interest Credit Card of Technical Debt*
E. *The Tail at Scale*
F. *Do Not Blame Users for Misconfiguration*
G. *Gender Differences and Bias in Open Source: Pull Request Acceptance of Women versus Men*

IV. "Your Choice" Reading Quiz

21. (30pt) Select **one** "Your Choice" reading about which you'd like to answer this question and circle the letter corresponding to it in the table above. Each of the items below is a direct quote from one of the "Your Choice" readings. Circle the letter of the quote from the "Your Choice" reading that you selected in the table above. If you didn't do any of the "Your Choice" readings, leave this question blank to recieve 6 points.

    (a) "In a product line approach, one must also consider requirements for the *family* of systems, and the relationship between those requirements and the ones associated with each particular instance." A

    (b) "Most binaries that were released in Windows Vista have fewer post-release defects in Windows 7." C

    (c) "Just as fault-tolerant computing aims to create a reliable whole out of less-reliable parts, large online services need to create a predictably responsive whole out of less-predictable parts." E

    (d) "We evaluate SPEX with the latest versions of one commercial system from a major U.S. storage vendor, and six open-source server software including Apache, MySQL, PostgreSQL, OpenLDAP, VSFTP, and Squid." F

    (e) "Research solutions that provide a tiny accuracy benefit at the cost of massive increases in system complexity are rarely wise practice. Even the addition of one or two seemingly innocuous data dependencies can slow further progress." D

    (f) "Our key contribution is *accumulation analysis*, a special case of typestate analysis that can be performed modularly without an alias analysis." B

    (g) there is no quote from the reading I selected in this list G

V. Document-based Questions (300pts). All questions in this section refer to a documents **A** through **C**. These documents appear at the end of the exam (I recommend that you tear them out and refer to them as you answer the questions).

Document **A** is an excerpt from a blog post written by an engineering manager from a large tech company introducing the concept of property-based testing ("PBT"), written for an audience of software engineers. Document **B** is the section on property-based testing from the "software testing" article on Wikipedia[1]; it includes a definition, since you are not expected to have encountered PBT before. In this section, you will be asked to apply your knowledge of testing from this course to this unfamiliar testing concept.

22. (80pt) Support or refute the following claim: "property-based testing" is just another name for "fuzzing" (which we did cover in this course, unlike PBT). **Must refute. The core difference between fuzzing and PBT is that PBT requires the programmer to write an oracle (the property), while fuzzing uses an implicit oracle. Both fuzzing and PBT rely on random or semi-random input generation, though. Half-credit for mentioning similarities without mentioning oracles.**

23. (80pt) Suppose that you are a software engineer at BankleysUS, the American arm of an English banking conglomerate. Your team is doing quality assurance work on the bank's "core", which is the system that a bank uses to process financial transactions between its accounts. One of your coworkers ("Alice") suggests that you should use property-based testing to ensure that financial rules like "debiting one account for $X should result in crediting another account for the same $X amount". Do you agree with Alice? Write a short (no more than one paragraph) email either supporting Alice's position or suggesting an alternative, addressed to your boss ("Bob"). **Likely support: this is a classic case where PBT is a good fit. Any professional email that engages with the question gets full credit, as long as the justification sounds reasonable. The only full-credit refute answer is to argue for formal verification instead. Refute suggestions with reasonable alternatives ("unit testing", "integration testing", etc) get 20 points. Zero credit for "fuzzing", since the previous question asked about it (lack of creativity).**

[1] Document **A** mentions a Wikipedia article on PBT itself, but it sadly appears to have been deleted.

24. Consider the small TypeScript program in Document **C**.

    (a) (40pt) Describe and justify a code-level design improvement that you could make to this program. **There are many. Easiest answer is the name "calculateTotalPrice" (a verb) should be noun-y, but we gave credit for anything reasonable.**

    (b) (40pt) Briefly describe a bug in the program. **There are at least three kinds of missing input validation, but it's easiest to answer this question with the fact that nothing stops the discount from being greater than the total cost. A common mistake on this question was to make an incorrect claim about TypeScript's semantics, which gets no credit. In particular, `acc` *is* defined: it is a local variable in the anonymous function being passed to `reduce`. Avoiding that mistake was a subtle functional programming question.**

    (c) (60pt) Write a reasonable property that you could use with a property-based testing framework to discover the bug in the program that you identified in question 24b. Write your property in first-order logic; if you need a refresher about first-order logic (which should have been covered in your Discrete Math course), one has been provided for you in Document **D**. $\forall cart, calculateTotalPrice(cart) > 0$ **is the easiest answer. CTE from part b was treated as leaving the question blank (12 points).**

VI. Extra Credit. Questions in this section do not count towards the denominator of the exam score.

25. (10pt) In section II (Matching), there is a theme to the names used in the situation descriptions. What is the theme? **First names of sitting members of New Jersey's congressional delegation (i.e., US Senators and Representatives): Andy Kim, LaMonica McIver, Cory Booker, Frank Pallone Jr., Thomas Kean Jr., Nellie Pou, Rob Menendez, and Bonnie Watson Coleman.**

26. (30pt) On the midterm, the following question appeared:

> Consider the following situations where we might apply delta debugging. Recall that delta debugging relies on three assumptions about the Interesting function: it must be monotonic, unambiguous, and consistent. For each situation below, circle the assumption(s) that are violated in that situation.
>
> (c) An application fails to start if its .xml configuration file is malformed. A developer has a large, failing config file and wants to find the minimal set of lines that causes the failure. The "Interesting" function is a script that prints the selected lines into a new file and feeds it to the application.

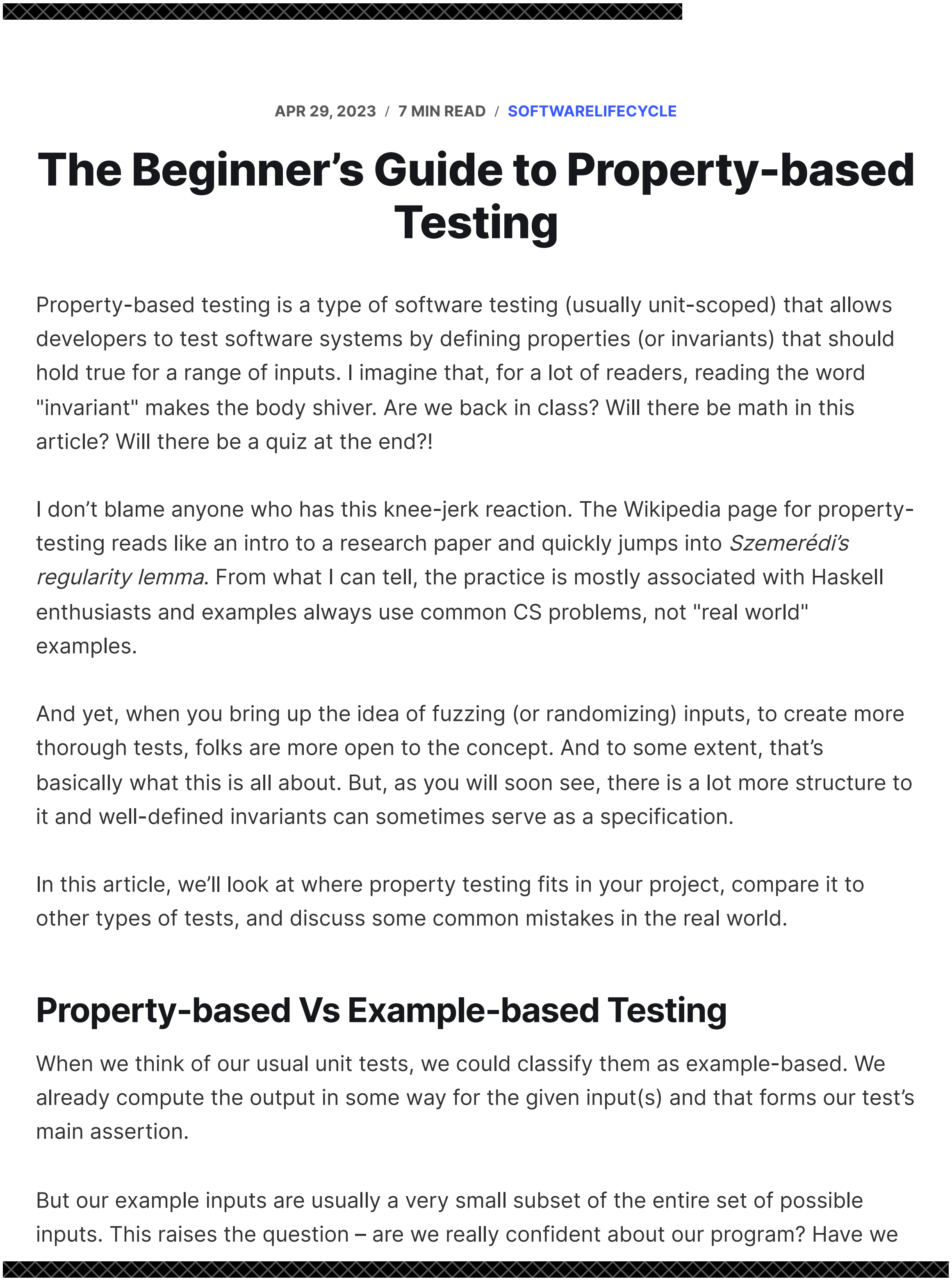
The midterm key was incorrect. How? What is the correct answer, and why? **The midterm key said that the answer was "monotonic", but the correct answer is "consistent". This situation violates consistency because most XML files are malformed in some way and don't pass the parser, and thus won't trigger the bug. This question is being asked here because I presume that the students most motivated to answer this will be those who were unfairly penalized on the midterm, and therefore they'll do particularly well - hopefully making up for my mistake in the original midterm key, to some extent, since regrading all the exams was not feasible at the time the error was discovered.**

27. (10pt) What was something interesting you learned from the engineer panel? **Any answer related to anything one of the engineers talked about that day gets full credit.**

28. (10pt) Would you be willing to serve on an engineer panel in a future semester, after you've graduated? If so, write an email address that you will continue to monitor after you've graduated in this space. If not, write "No" to recieve credit for this question. **Any email address or "No" recieves credit.**

This page intentionally left blank (you may use it as scratch paper, and the proctor will have more scratch paper at the front if you need more).

Document A:



APR 29, 2023 / 7 MIN READ / SOFTWARELIFECYCLE

The Beginner’s Guide to Property-based Testing

Property-based testing is a type of software testing (usually unit-scoped) that allows developers to test software systems by defining properties (or invariants) that should hold true for a range of inputs. I imagine that, for a lot of readers, reading the word "invariant" makes the body shiver. Are we back in class? Will there be math in this article? Will there be a quiz at the end?!

I don’t blame anyone who has this knee-jerk reaction. The Wikipedia page for property-testing reads like an intro to a research paper and quickly jumps into *Szemerédi’s regularity lemma*. From what I can tell, the practice is mostly associated with Haskell enthusiasts and examples always use common CS problems, not "real world" examples.

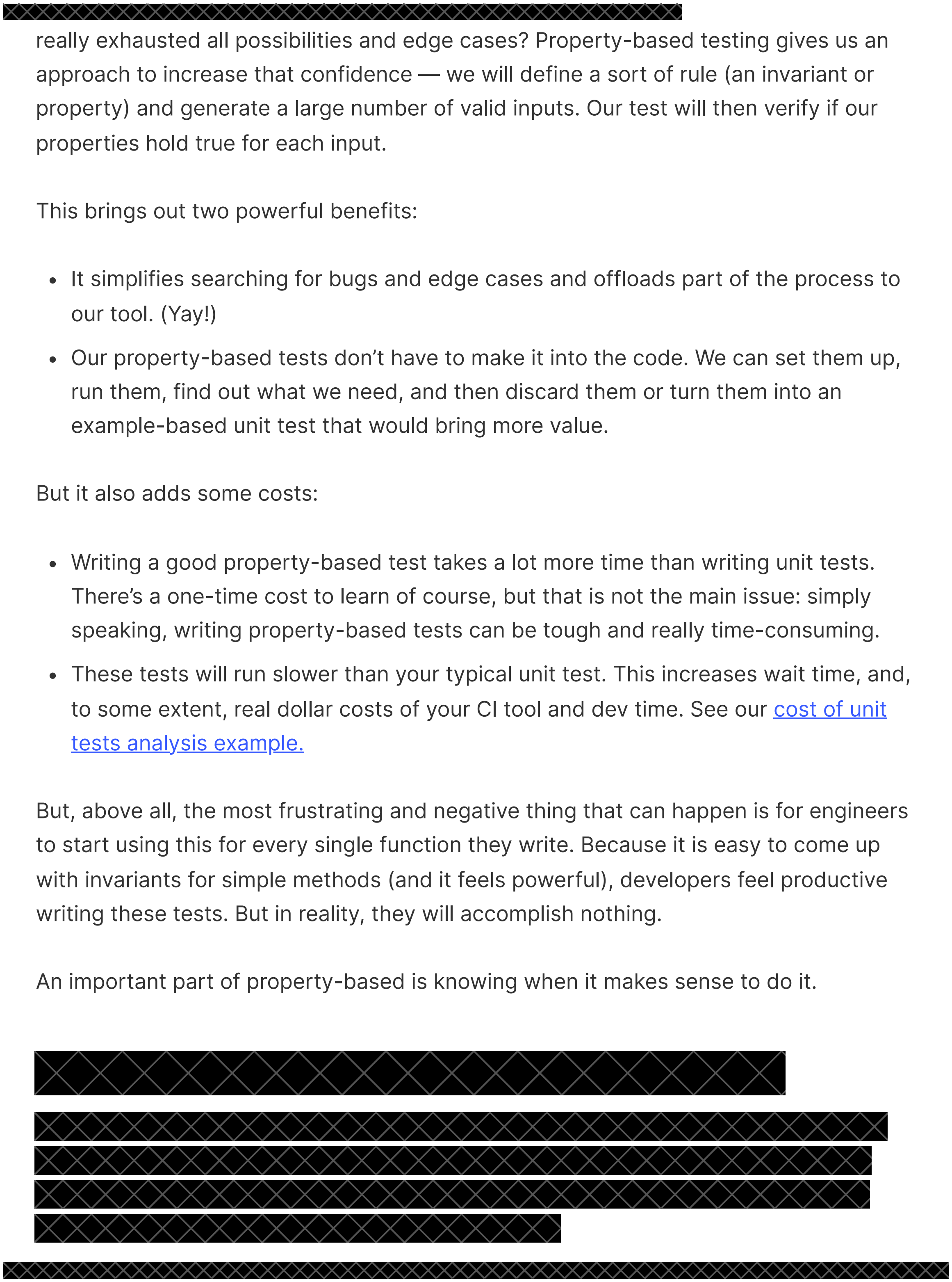
And yet, when you bring up the idea of fuzzing (or randomizing) inputs, to create more thorough tests, folks are more open to the concept. And to some extent, that’s basically what this is all about. But, as you will soon see, there is a lot more structure to it and well-defined invariants can sometimes serve as a specification.

In this article, we’ll look at where property testing fits in your project, compare it to other types of tests, and discuss some common mistakes in the real world.

Property-based Vs Example-based Testing

When we think of our usual unit tests, we could classify them as example-based. We already compute the output in some way for the given input(s) and that forms our test’s main assertion.

But our example inputs are usually a very small subset of the entire set of possible inputs. This raises the question – are we really confident about our program? Have we



[redacted]

really exhausted all possibilities and edge cases? Property-based testing gives us an approach to increase that confidence — we will define a sort of rule (an invariant or property) and generate a large number of valid inputs. Our test will then verify if our properties hold true for each input.

This brings out two powerful benefits:

- It simplifies searching for bugs and edge cases and offloads part of the process to our tool. (Yay!)
- Our property-based tests don't have to make it into the code. We can set them up, run them, find out what we need, and then discard them or turn them into an example-based unit test that would bring more value.

But it also adds some costs:

- Writing a good property-based test takes a lot more time than writing unit tests. There's a one-time cost to learn of course, but that is not the main issue: simply speaking, writing property-based tests can be tough and really time-consuming.
- These tests will run slower than your typical unit test. This increases wait time, and, to some extent, real dollar costs of your CI tool and dev time. See our cost of unit tests analysis example.

But, above all, the most frustrating and negative thing that can happen is for engineers to start using this for every single function they write. Because it is easy to come up with invariants for simple methods (and it feels powerful), developers feel productive writing these tests. But in reality, they will accomplish nothing.

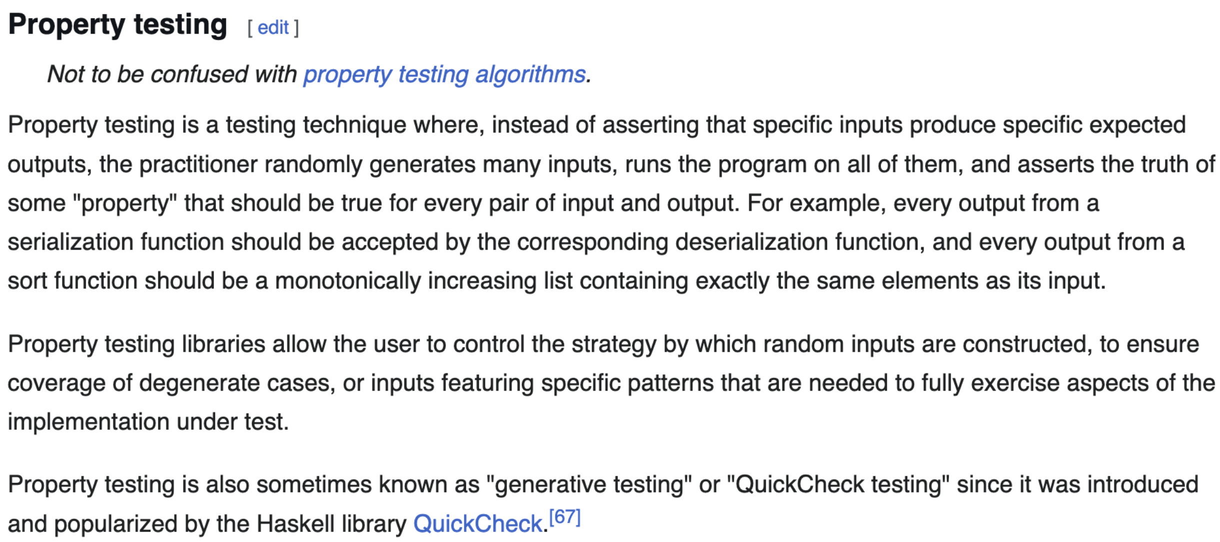
An important part of property-based is knowing when it makes sense to do it.

[redacted]

[redacted]

[redacted]

Document B:



Property testing [edit]

Not to be confused with property testing algorithms.

Property testing is a testing technique where, instead of asserting that specific inputs produce specific expected outputs, the practitioner randomly generates many inputs, runs the program on all of them, and asserts the truth of some "property" that should be true for every pair of input and output. For example, every output from a serialization function should be accepted by the corresponding deserialization function, and every output from a sort function should be a monotonically increasing list containing exactly the same elements as its input.

Property testing libraries allow the user to control the strategy by which random inputs are constructed, to ensure coverage of degenerate cases, or inputs featuring specific patterns that are needed to fully exercise aspects of the implementation under test.

Property testing is also sometimes known as "generative testing" or "QuickCheck testing" since it was introduced and popularized by the Haskell library QuickCheck.[67]

Document C:

```
type Product = { name: string; price: number; }
type Discount = {
    kind: ’flat’ | ’percentage’;
    amount: number; // if percentage, then a number between 0 and 1
}
type ShoppingCart = {
    items: {
        product: Product;
        quantity: number;
    }[];
    discount?: Discount;
}
const calculateTotalPrice = (cart: ShoppingCart): number => {
    const preDiscountPrice = cart.items.reduce((acc, item) =>
      acc + item.product.price * item.quantity, 0);
    if (!cart.discount) { return preDiscountPrice; }
    if (cart.discount.kind === ’flat’) {
        return preDiscountPrice - cart.discount.amount;
    }
    return preDiscountPrice * (1 - cart.discount.amount);
}
```

Document D (First Order Logic Reference):

Symbol	Name	Meaning & English Translation
$\neg P$	Negation	**Not** P. True only if P is false.
$P \wedge Q$	Conjunction	P **and** Q. True only if both are true.
$P \vee Q$	Disjunction	P **or** Q. True if at least one is true. (Inclusive or).
$P \rightarrow Q$	Implication	**If** P, **then** Q. False *only* if P is true and Q is false.
$P \leftrightarrow Q$	Biconditional	P **if and only if** Q. True if P and Q have the same truth value.
$\forall x, P(x)$	Universal Quantifier	**For all** x, $P(x)$ is true. False if you can find a single counter-example.
$\exists x, P(x)$	Existential Quantifier	**There exists** an x such that $P(x)$ is true. True if you can find at least one example.

Example Use of First-Order Logic with Code:

Suppose that you have the following program:

```
const max = (x : number, y : number) => {
  if (x > y) { return x; }
  else { return y; }
}
```

Then, you could write the following first-order logic formula to express `max`'s specification:

$\forall x, y, x > y \rightarrow max(x, y) = x \wedge y > x \rightarrow max(x, y) = y$