Property-Based Testing in Intro CS: A Feasibility Study

Xiaorui Liu
xrl@engineering.upenn.edu
University of Pennsylvania
Philadelphia, PA, United States

Jeffrey Tao
jefftao@engineering.upenn.edu
University of Pennsylvania
Philadelphia, PA, United States

Harrison Goldstein
hgoldste@buffalo.edu
University at Buffalo
Buffalo, NY, United States

Benjamin C. Pierce
bcpierce@engineering.upenn.edu
University of Pennsylvania
Philadelphia, PA, United States

Andrew Head
head@engineering.upenn.edu
University of Pennsylvania
Philadelphia, PA, United States

Abstract

Property-based testing (PBT) is a software testing technique that is increasingly applied in industry, complements testing approaches like unit testing, and offers an intuitive onramp to program specification, yet is not commonly taught to undergraduates. We wondered: Would it fit in an introductory CS course? How should it be presented? Does it help students in coding assignments? What difficulties do they encounter? Which parts of "industrial-strength" PBT workflows can be abstracted away in this setting?

To address these questions, we integrated PBT into an introductory CS course at our university. Students were given guidance on how and when to write properties and asked to use a simple PBT framework in two data structures assignments. To understand how this material was received, we collected student submissions, survey responses, and answers to exam questions.

We found that students at this level can understand and apply PBT productively—indeed, many even reached for sophisticated PBT patterns like differential tests—and that they found PBT useful in completing assignments. We also encountered some challenges, including how to motivate PBT when unit tests are sometimes more effective for simple data structures. We offer our course materials and findings as inspiration for others interested in teaching PBT.

CCS Concepts

• **Social and professional topics** → **Computer science education**; • **Software and its engineering** → *Software testing and debugging*.

Keywords

Property-based testing; CS1/CS2; software testing; specification

ACM Reference Format:
Xiaorui Liu, Jeffrey Tao, Harrison Goldstein, Benjamin C. Pierce, and Andrew Head. 2027. Property-Based Testing in Intro CS: A Feasibility Study. In *Proceedings of the 58th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2027), February 17–20, 2027, Sacramento, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3857222.3860217



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCSE TS 2027, Sacramento, CA, USA
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3075-7/2027/02
https://doi.org/10.1145/3857222.3860217

1 Introduction

Like many CS educators [10, 13, 18, 25], we believe that testing should play a major role in introductory programming. In our first-year CS course, students have for years learned unit testing and applied it through test-driven development (TDD) [9]. This approach encourages students to think about program behavior before implementation and gives them tools for establishing the correctness of their code. The success of this approach led us to wonder whether we could further enrich students' engagement with testing by adding property-based testing (PBT) to the course.

PBT is a lightweight, automated testing technique that evaluates an executable specification against a program over many randomly generated inputs. It is used both in leading companies [4, 11] and in open-source development [7], and PBT tools are downloaded millions of times a week [23]. Its workflow encourages students to think about program behavior at the level of general properties rather than individual examples, developing a specification-centered mindset. This mindset is increasingly valuable in the age of LLM-based programming, where it is valuable to have experience with precisely specifying and verifying program behavior.

We report here on an investigation into the feasibility of integrating PBT into an introductory CS course. We first introduce PBT and compare our integration of PBT to related experiments elsewhere (Section 2). We then describe our integration in detail (Sections 3 and 4). Analyzing students' reports on their experience, we find that PBT was both approachable and useful, with students often implementing strong properties and reporting success catching bugs with PBT (Section 5). We close with reflections on how to further improve students' first experiences with PBT (Sections 6 and 7). Our course materials can be found at pennhci.com/pbt-in-intro-cs.

2 Background

Property-based testing. PBT [5] is an automated testing technique that checks whether an executable statement about a program's behavior holds for a large number of randomly generated inputs. To demonstrate, consider this unit test in OCaml about sets:

```
let test () : bool =
  let s = add 2 (add 1 empty) in
  member 3 (add 3 s)
;; run_test "correct membership" test
```

This says that if we add 3 to the set {1, 2}, it becomes a member of the resulting set. Generalizing, we could expect that if *any* value

is added to *any* set, that value should be a member of the resulting set. This expectation can be written as a property-based test:

```
1 let property ((x, s) : int * int set) : bool =
2   let s_after_add = add x s in
3   member x s_after_add
4 ;; run_property_test "correct membership"
5     (gen_pair gen_int (gen_set gen_int))
6     property
```

This property (lines 1–3) states that, given a value `x` and a set `s`, if `x` is added to `s`, then `x` should be a member of the resulting set. The property is run against random inputs produced by a generator (line 5). Here, the generator produces a pair containing a random integer (`x`) and a random set (`s`). The test harness (line 4) repeatedly calls the generator and passes the inputs to the property, reporting successes and failures to the user.

Related work. We are aware of two prior efforts to teach PBT early in the CS curriculum. Moy and Patterson [20] teach it in a CS logic course. PBT is a core component of this course, and they give students exercises like writing custom generators for provided data types and applying PBT to complex algorithms. Interviewed students reported leaving with a positive impression of PBT, having learned "the value of extracting logical properties" and how PBT could help validate them. Our experiment targets an earlier point in the curriculum.

Nelson et al. [21] and Wrenn et al. [26] introduce PBT as a way to test relational problems, where there are multiple valid outputs for an input (e.g., a function that finds a spanning tree where there are many possible valid trees). PBT is a natural fit for these problems because it uses general statements of expected program behavior without using specific outputs, whereas a unit test's hard-coded output may not match the returned output of a correct implementation, causing the test to erroneously fail. Nelson et al. and Wrenn et al. have students write generators and properties for these relational problems. In contrast, we introduce PBT to students as a method of testing their own data structure implementations.

3 Context

We added PBT to a "CS1.5" course, the first required course for CS majors at the University of Pennsylvania.[1] Described as "a fast-paced introduction to the fundamental concepts of programming and software design," it assumes basic familiarity with programming from AP CS in high school or a preparatory course we offer. For many students, it leads into a traditional data structures and algorithms course (CS2). The first half of the course covers functional programming in OCaml [2], including traditional CS1 topics such as data types, abstraction, and fundamental programming patterns (recursion, iteration, etc.). The second half covers object-oriented programming in Java [1]. Testing is a required and graded component of most homework assignments. In the semester we ran our study, the course had 164 students.

We added PBT content to weeks 4–7 of the course (Figure 1), which span two homework assignments on data structures—one on purely functional trees and sets and one on imperative queues and deques. We chose these assignments because of PBT's natural fit with these data structures problems. Data structures have rich sets of properties to test [16], and PBT is commonly used in industry to test data structures [11]. By waiting until week 4, we let students acclimate to unit testing, the course's testing framework, and TDD.

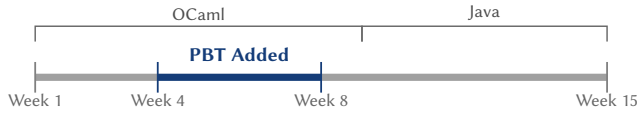


Figure 1: Course timeline. We introduced PBT in a four-week segment of a CS1.5 course, during which students completed two data structures homeworks in OCaml, released in weeks 4 and 6.

PBT framework. Good PBT libraries exist in most major programming languages. In our course, we adapted QCheck [6], a PBT library for OCaml. To simplify the experience of working with QCheck, we added a thin layer to streamline its API and make it look more similar to the course's unit testing framework.

QCheck, like other PBT libraries, provides rich facilities for creating generators, but we opted to hide these from students. Creating generators requires both effort and nuance to get right [11], and we did not believe we could properly teach these skills in the time offered. Instead, we provided a collection of pre-written generators (like `gen_int`) and simple combinators for building bigger generators from smaller ones (like `gen_pair`). The property-based test in Section 2 is written with our variant of QCheck.

Learning objectives. Our changes to the course were motivated by the following learning objectives (**LOs**):

- **LO1**: Students should learn basic PBT concepts, mainly properties and generators.
- **LO2**: Students should be able to use a PBT framework to write property-based tests for their own code.
- **LO3**: Students should be able to come up with properties on their own.
- **LO4**: Students should understand when it is appropriate to choose PBT vs. plain unit testing.

4 Interventions

We next describe how we adapted our course to teach PBT. Changes spanned a lecture, a recitation, and two core homework assignments (HW3 and HW4). Throughout, we note the learning objectives that our changes addressed.

Lecture. We introduced PBT in a lecture before HW3, juxtaposing it against unit testing (LO1). This took around 20 minutes.

Recitation. After this lecture but before HW3 was released, we added some PBT exercises to TA-led recitations. First, we reviewed our PBT framework (LO2). Then, we had students complete an activity based on Sortacle [26], a relational sorting problem with multiple valid outputs per input, where unit testing is insufficient for establishing correctness. This activity was meant to provide practice coming up with properties with TA guidance (LO3).

HW3. The first assignment where we added PBT was originally designed to provide practice with OCaml's features for abstraction and modularity by having students implement higher-order functions, trees, and sets. We augmented it to motivate PBT, teach its concepts

[1] https://www.engineering.upenn.edu/~cis120

and workflow (LO1), provide practice with a PBT framework (LO2), and provide practice thinking of properties (LO3).

The most substantial addition to HW3 was a new tutorial section. It began by showing how PBT could be used to generalize a suite of unit tests. It then introduced generators, describing them as tools for sampling random distributions of basic data types (LO1), and explained how to interpret PBT test outputs (LO2). The tutorial concluded with an exercise where students implemented several property-based tests for a simple list module with `sort`, `reverse`, and `add` functions. The tutorial provided properties in English, and students translated them into actual property-based tests (LO2). These properties were described in terms of common categories found in the literature [11, 16, 24]:

- **Post-condition properties**, which evaluate the output of a function against some expected condition.
- **Validity properties**, which check that operations maintain data structure invariants (e.g., adding an element to a sorted list leaves it sorted).
- **Round-trip properties**, which check that pairs of inverse or one-sided inverse operations recover the original input (e.g., printing then parsing).
- **Differential properties**, which validate that two implementations of the same functionality (e.g., a reference implementation of some data structure and a high-performance one) behave the same across all inputs.
- **Metamorphic properties**, which state relationships between sequences of operations (e.g., changing the order in which two elements are added to a set results in "observationally the same set").

We believed these categories would provide useful scaffolding for coming up with properties independently, and we referred back to them elsewhere in the homeworks. To give students more practice with PBT, we had them write property-based tests for binary search trees. We gave them descriptions of the properties in English and also indicated which category each property belonged to (LO2).

Finally, we had students do a graded PBT task. We asked students to write out, in plain language, six properties that could be used to test set operations. We provided a few properties as examples, such as how "adding an element to a set should not change its existing members." Then, we asked students to implement four property-based tests for their set implementations (LO2, LO3).

HW4. The second assignment provided experience with imperative programming by having students implement mutable queues and deques. We asked students to write at least one property-based test and one unit test for each function they were responsible for implementing (two functions for queues, five for deques). We told them they would be graded on the basis of the robustness of their test suites (LO4).

We did not prescribe a specific process for coming up with and implementing properties in HW3 and HW4. That said, HW3 did break down the work of creating properties into two separate stages of coming up with them and implementing them. In HW4, we offered tips on coming up with properties, namely to review data structure invariants, reference property categories, and write properties in English before implementing them.

Evaluation. We evaluated the student experience by reviewing student homework submissions, required feedback surveys at the end of homeworks, exam performance, and (optional, paid) interviews and observations with students. All data were collected according to a protocol approved by our institutional review board.

Our findings mainly draw from student submissions, survey responses, and answers on exams. These data provided useful summary evidence of broader outcomes and reception. PBT portions of homeworks were manually graded by the first author. Surveys asked two types of questions. First, 7-point Likert scale ratings (1 = *Strongly disagree* to 7 = *Strongly agree*) like "PBT was helpful for testing sets in setTest.ml." Second, open-ended free-response questions like "In what situations does it make sense to write a property-based test instead of a unit test?" PBT questions were included in both a midterm and the final. The interviews and observations are only lightly used to add color when appropriate; these data are still under analysis and will be the subject of an in-preparation paper.

AI use. Course policy prohibited use of AI to generate code. We spot-checked submissions to verify that they appeared to be students' own work. The first author reviewed playback of 20 random students' HW4 submissions, finding only one student blatantly pasting code from an external source. Across HW3 and HW4, fewer than 3% of submissions were flagged by the TAs and lead instructor for academic dishonesty.

5 Outcomes

We next compare observed results to the learning objectives. Our top-level finding is that PBT can be taught productively in CS1.5.

Understanding PBT. Most students appeared to internalize PBT's basic concepts (LO1). In the HW3 survey, 86% of students responded positively (5+ on a 1–7 scale) to "I understand the concepts of PBT." In interviews, all 13 students were able to explain the role of properties (e.g., "a property is a general rule or invariant that describes how a program or function should behave for all valid inputs") and generators (e.g., "you have to define the type of the input [for the generator]... and then you can use it to come up with randomized inputs").

Using a PBT framework. On the whole, students were successful at using the provided PBT framework (LO2). All students submitted a property-based test for each of the five deque functions in HW4. Survey responses provided a view into the usability of the framework. Fifteen students described the framework getting easier to use over time: the syntax of the framework was "unfamiliar" at first but became "much easier than anticipated" after a little practice. Eight students indicated that it was straightforward to use: tests followed a "basic template" and it was "easy to recreate the structure of tests once I got a hold of the logic." For eight others, the syntax was more involved. At the end of HW4, one student wrote that they "struggled and took time in implementing the syntax."

Meaningful engagement with PBT. Students independently thought of properties (LO3). In HW3, 97% of students submitted six correct plain-language descriptions of properties for set operations. These ranged from simple validity properties testing set invariants to more

```
let rec trunc_list (x: 'a) (l: 'a list) : 'a list =
  begin match l with
    | [] -> []
    | hd::tl -> if hd = x then [x]
        else hd :: trunc_list x tl
  end

let property ((ops, x): int op list * int): bool =
  let q = init_with ops in
  let tlist = trunc_list x (to_list q) in
  truncate x q;
  to_list (q) = tlist
;; run_property_test "truncated list = list of truncate"
    (gen_pair gen_ops gen_int)
    property
```

Figure 2: Many students wrote fairly complex properties. Here, a student wrote a differential test of a `truncate` function. The test compares their implementation to a reference implementation the student wrote called `trunc_list`. Thirty-eight students submitted similar differential properties for `truncate`.

complex ones, such as "two sets with exactly the same elements are equal, regardless of the order in which elements were added."

In HW4, many students were clearly proficient in thinking of and implementing properties (LO2, LO3). As previously mentioned, all students submitted at least one property-based test for the deque functions they were responsible for testing. Furthermore, many students wrote complex and effective properties. One kind of property, differential tests, was written by 38 students for the queue function `truncate`. A student's implementation (Figure 2) shows how involved this kind of property is. Since differential tests provide strong coverage [16], it was encouraging to see students reaching for them. Students also submitted many other sensible properties, including properties that used relevant characteristics of the data structures, such as comparing expected changes to length, and properties that specified precise, unique post-conditions, like checking that all elements before a truncated element were preserved. The variety of properties suggests that students were not solely repeating known properties but rather coming up with their own.

Beyond individual tests, students also submitted diverse *suites* of properties. Many students wrote more properties than required for HW4. For example, over half of students wrote two or more properties for at least one of the required queue functions. One student even wrote seven properties for one function. But were students writing tests productively? While we observed students doing HW4, some appeared to be able to assess coverage across their test suites. One student articulated how two properties they wrote uniquely addressed different aspects of the queue function `delete`. One test checked that `delete` didn't erroneously alter elements and the other checked that `delete` deleted the correct element. Students also appropriately elaborated on when to use PBT over unit tests in the HW3 survey (LO4), citing PBT's use in "verifying general invariants," catching "hidden edge cases," and testing "when the input space is very large and hard to measure case by case."

Exam performance. We tested student knowledge of PBT in two exams. The midterm, held between HW3 and HW4, contained two questions evaluating students' ability to translate properties between plain language and code (LO2). Just over half of students earned full credit for translating a property from plain language to code (53%) and from code to plain language (58%). Most students received at least partial credit (89% and 71%, respectively).

On the final exam, students performed well on all three of the PBT questions we included. When judging whether candidate properties were valid for a novel data structure (LO3), 77% of students received full credit and nearly all (97%) answered at least four of five items correctly; per-item accuracy averaged 95% (range: 89–98%). On a cloze-style completion of a property-based test, where students filled holes from multiple-choice options (LO2, LO3), two-thirds (67%) received full credit. Most chose the correct generators (93%) and at least one of the two expected logical clauses (97%). The most common error was not PBT-related but a misuse of structural equality (a perennial gotcha in OCaml). Finally, when judging whether candidate properties would catch a buggy implementation (LO4), 77% received full credit and 92% misjudged at most one of four properties; per-item accuracy averaged 91% (range: 81–97%).

Student perception. Student feedback also suggested that PBT was received well. Many responded that PBT was useful (5+ on a 1–7 scale) in the homeworks for testing sets (86%) and queues/deques (85%). In both assignments, around 30% of all students reported finding a bug in their implementations using PBT. (This may be an underestimate. Recall during interviews was poor even shortly after submission, leading us to think students may also have had trouble recalling specifics in the surveys.) Qualitatively, students appreciated the strength PBT lent to their tests, describing how it "provided more comprehensive coverage," "made me more confident," and "helped with verifying the functionality of the entire module."

Students described several ways PBT let them test thoroughly and easily. Over 80 survey responses mentioned generators as a key benefit. Students valued avoiding "wasting time writing multiple unit tests" and being able to "test many different cases automatically, especially edge cases I might have not thought of." One student described how generators shifted their focus from being "creative with my examples" to "think[ing] more at the level of detailed specifications, things I would ideally want to prove about my programs."

Students also found properties apt for testing data structures. Ten students pointed out how PBT was well suited to testing the data structures from the homeworks. One student wrote that "the nature of queues and deques made it more difficult than previous assignments to make unit tests. I personally found it easier to think up property tests." Even in HW3, a student described how "many set properties are intuitive and map directly to mathematical laws." Two other students wrote that properties were especially useful for testing "interactions between operations," tests they might not have written if they were "relying only on example-based tests."

Finally, nine students expressed that the process of writing properties helped them understand their code or data structures in some way. Properties helped them to think "through how the functions work before implementing" and "about the problem further." One student felt "neutral" about PBT but admitted that "it helped me understand more what the functions were supposed to do." Some students even described PBT impacting their implementation (three students) and debugging (one student). They described properties as helpful for "discovering common patterns for each test case,

which made it easier to write the code" and "actually understanding how the queues themselves worked in the first place, which in turn helped me debug successfully."

6 Reflections

While we feel the experiment of bringing PBT to an intro course was largely successful, it also surfaced a number of subtle considerations—many more than we expected—around how to teach PBT effectively. Some of these we were able to address in real time during the semester; others are topics for future improvement.

Juxtaposing PBT against unit testing. Since students used both PBT and unit tests, this raised some interesting tensions. Some students saw property-based tests as inferior to unit tests. PBT could require more effort than desired; one student wrote that "writing the PBTs was frustrating at times and writing the unit tests were significantly easier and more intuitive."

A few other students pointed out issues with the coverage provided by PBT. One student told us, "I believe I could have implemented a similar level of rigorous testing with simply unit tests." Indeed, unit tests sometimes *do* provide better coverage than property-based tests. Consider a property that checks that a set's size changes by at most one after an element is removed. This property—which a student might well write—doesn't include a check that the correct element is removed. A unit test, on the other hand, *will* naturally check this. This example shows that arbitrary property-based tests are not necessarily stronger than unit tests, and this point takes finesse to deliver well.

For these reasons, we do not advocate *replacing* unit testing with PBT in introductory CS. Unit testing is easier to explain at the start of the course, simpler to use, and, as described above, equally or more effective in some cases. When PBT and unit testing are offered as complements, instructional materials should make property-based tests as approachable to write as possible and emphasize when they offer useful coverage.

How much to hide randomness. When we observed students doing the homeworks, they were often concerned about the distribution of random values produced by generators—wondering, in particular, if they could be trusted to cover edge cases. We suspect this concern led some students to depend on unit tests, as they wrote them "just in case" certain inputs were not generated. We wonder, then, whether it is possible to give students more control over input distributions without teaching them to write generators. One option could be to let students augment generators with specific examples they want to have tested. Indeed, popular PBT frameworks like Hypothesis [19] let users do this. Students might also be given tools to help them understand what values are generated (e.g., [12]). Another option, which we are currently testing, is to remove the temptation to think about generators entirely by replacing them with fixed sets of predetermined inputs. While this may solve the problem of students fixating on generators, it has the downside of removing randomized input generation, which students appreciated.

Helping students assess property strength. Some students shared concerns about "coming up with strong properties that actually capture the behavior of a function" and "thinking about enough properties to still have sufficient testing." One solution is to be explicit about which kinds of properties achieve higher coverage. For example, when applicable, a differential test is often the strongest property a student can write. While straightforward, this approach may diminish parts of PBT's workflow that are valuable for students to engage with, such as finding specific properties, selecting some to implement, and thinking critically about specification strength. Alternatively, students could receive feedback on their tests' efficacy, through automated methods like those proposed by Nelson et al. [21] or how well their tests can catch "mutants" of correct code [8]. We are exploring adding the latter.

Bringing PBT to open-ended assignments. After HW4, 71% of students indicated some agreement and 17% responded neutrally to the statement "I could envision wanting to use PBT in future programming work." We wanted, but were not able in the span of one semester, to incorporate PBT into later assignments. These include assignments that have students develop GUIs and games.

One challenge is that these later assignments do not present clear opportunities for PBT to shine. For example, while a GUI does have properties about its layout ("widgets should never cross this border") or interactions ("clicking this button should always reset the state"), turning these into executable properties while also generating random state is difficult. Data structures, on the other hand, have plenty of precise, actionable specifications. An additional technical hurdle is that students define their own data structures in these assignments, which requires custom generators. Students would need to learn to use generator derivation tools like QCheck's `ppx_deriving_qcheck` or to write generators by hand.

7 Next Directions

PBT with other programming languages. We expect minimal difficulty in adding PBT to courses that use other programming languages. Since PBT is supported in most major programming languages, much of the necessary infrastructure should already exist for instructors to adapt. Future work should explore the outcomes of introducing PBT with other programming languages and tooling.

We did find, however, that random input generation was harder to explain for imperative-style programs, since a test that mutates its input can lose access to the original state. We addressed this by generating random sequences of operations that construct the initial mutable state (e.g., `gen_ops` in Figure 2), rather than the data structures themselves. Students could then mutate the state while still referencing the operations after the fact. Courses teaching imperative-style programming should anticipate this complexity.

Bringing PBT to more courses. We believe PBT could be a good fit for many other CS courses. Given our positive experiences teaching PBT with simple data structures, a traditional CS2 course is a natural fit for PBT—arguably even more natural than CS1.5. CS2 typically covers more complex data structures and algorithms with clear properties to test. Additionally, since students are more comfortable with programming in a CS2 course, it may be more feasible to teach them how to create their own generators there. PBT is also a natural addition to upper-division courses in domains where it has been successfully applied, such as compilers [22], software engineering [3, 11, 17], and cybersecurity [14, 15].

8 Conclusion

This experiment shows that PBT can be a viable and valuable addition to the early CS curriculum. We hope our overall positive experience, identification of key challenges, and course materials will prove useful for other instructors looking to weave PBT into their own courses.

Acknowledgments

We thank the students and teaching assistants in the Spring 2026 offering of CIS 1200 for participating in our study. We also thank Talie Massachi and Swapneel Sheth for their feedback on the paper. This research was supported with funding from the National Science Foundation under grant #2402449, *SHF: Medium: Usable Property-Based Testing*.

References

[1] 2026. Java. Retrieved November 9, 2026 from https://www.java.com/en/
[2] 2026. OCaml. Retrieved November 9, 2026 from https://ocaml.org
[3] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–4.
[4] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850.
[5] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279.
[6] Simon Cruanes. 2026. c-cube/qcheck. Retrieved November 9, 2026 from https://github.com/c-cube/qcheck
[7] Isadora de Oliveira, Arthur Lisboa Corgozinho, Henrique Rocha, and Marco Tulio Valente. 2026. Property-based testing in Python: empirical insights. *Empirical Software Engineering* 32, 1, Article 30 (Aug. 2026), 36 pages.
[8] Richard DeMillo, Richard J. Lipton, and Fred Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41.
[9] Stephen H. Edwards. 2003. Rethinking computer science education from a test-first perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*. Association for Computing Machinery, New York, NY, USA, 148–155.
[10] Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. 2002. An experience in integrating automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin* 34, 4 (Dec. 2002), 125–128.
[11] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 187, 13 pages.
[12] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. 2024. Tyche: Making Sense of PBT Effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages.
[13] Michael H. Goldwasser. 2002. A gimmick to integrate software testing throughout the curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 271–275.
[14] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 557–560.
[15] Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing noninterference, quickly. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 455–468.
[16] John Hughes. 2020. How to Specify It!: A Guide to Writing Properties of Pure Functions. In *Trends in Functional Programming: 20th International Symposium, TFP 2019, Vancouver, BC, Canada, June 12–14, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12053)*. Springer, Cham, 58–83.
[17] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 135–145.
[18] Edward L. Jones. 2001. Integrating testing into the curriculum — arsenic in small doses. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*. Association for Computing Machinery, New York, NY, USA, 337–341.
[19] David R. MacIver, Zac Hatfield-Dodds, and many other contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43, Article 1891 (Nov. 2019), 3 pages.
[20] Cameron Moy and Daniel Patterson. 2025. Teaching Software Specification (Experience Report). *Proc. ACM Program. Lang.* 9, ICFP, Article 264 (Aug. 2025), 16 pages.
[21] Tim Nelson, Elijah Rivera, Sam Soucie, Thomas Del Vecchio, John Wrenn, and Shriram Krishnamurthi. 2021. Automated, Targeted Testing of Property-Based Testing Predicates. *The Art, Science, and Engineering of Programming* 6, 2, Article 10 (Nov. 2021), 29 pages.
[22] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. Association for Computing Machinery, New York, NY, USA, 91–97.
[23] PyPI Stats. 2026. *Download Statistics for the hypothesis Package*. Retrieved October 31, 2026 from https://pypistats.org/packages/hypothesis
[24] Savitha Ravi and Michael Coblenz. 2025. An Empirical Evaluation of Property-Based Testing in Python. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 412 (Oct. 2025), 27 pages.
[25] Lilian Passos Scatalon, Jeffrey C. Carver, Rogério Eduardo Garcia, and Ellen Francine Barbosa. 2019. Software Testing in Introductory Programming Courses: A Systematic Mapping Study. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 421–427.
[26] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2020. Using Relational Problems to Teach Property-Based Testing. *The Art, Science, and Engineering of Programming* 5, 2, Article 9 (Nov. 2020), 23 pages.