# Property-based testing - how it works and when to use it

Property-based testing (PBT) is an approach to software testing that offers significant advantages over the example-based approach that's widely used in software development today.

This article explores how property-based testing works, how to implement it effectively, and which kinds of systems benefit most from it.

## What is property-based testing?

To understand what property-based testing is, it's easiest to first compare it to the common approach of example-based testing. In example-based testing, each test sets up a concrete example with specific inputs.

Let's say you've written a function that searches for a target integer in a list. In example-based testing, you'd write individual tests that take some example lists and check that you get the expected result. For example, you might check that if you search for `5` in the list `[5, 3, 1, 4]` it returns `0`, the position of `5` in the list, whereas if you search for `5` in `[2, 3, 8, 4, 7]` it returns `-1` to indicate that it hasn't found it. This is time-consuming because you have to write each example by hand. Worse, you're limited by the examples you manage to think of.

In property-based testing, you instead define general properties that you want your function to have. In this case, you'd check that your function returns the index of the target integer in the list if it finds it, and -1 otherwise. You then let the computer do the work of generating lots of random lists of integers and checking that the properties always hold.

## Related terms

Property-based testing, "fuzzing" or "fuzz testing," and "generative testing" are essentially interchangeable terms. The difference between them is mostly historical: the term "fuzzing" has been used more in work that focuses on crashes or security, whereas "property-based testing" and "generative" are used more often in tests focusing on logical correctness – often for unit tests, verification of data structures, or end-to-end correctness tests. Today the terms are often used interchangeably.

Although the terms have a difference in emphasis, fuzzing always requires checking that some property holds (for example, that the program does not crash), while property-based testing needs a source of input randomness. "Generative testing" is a looser term that's been used as a synonym for both fuzzing and property-based testing.

We prefer to use “property-based testing” because “fuzzing” has historically been used more in tests that focus on crashes or security, whereas “property-based testing” is more often used in tests that focus on more fine-grained logical properties of the code.

## How do I implement property-based testing?

You can apply property-based testing at different levels, from individual functions up through whole programs, and even distributed systems.

To use property-based testing on specific functions or data structures in your system, you can use language-specific libraries. This approach was popularized by the Haskell library QuickCheck. There are now property-based testing libraries for most common languages – examples include Hypothesis for Python and FsCheck for C# and other .NET languages. Libraries often include a way to shrink failing inputs to smaller ones, which are often more helpful for understanding what’s causing the bug.

At the level of whole programs, you can use fuzzing tools like AFL. Fuzzers submit random inputs to your program and are often used to check more coarse-grained properties, like whether the program crashes or hangs.

Tooling at the level of distributed systems is less common. However, Antithesis is able to apply property-based testing to your whole software system, including your program and the environment it runs in. To use Antithesis, you package your system along with its dependencies and specify the properties you want to test. Antithesis then uses deterministic simulation testing to run your software in a simulated environment where you can reliably reproduce a bug that you see on a given run.

## How does property-based testing work?

Property-based testing requires:

- Identifying and defining properties that you want your system to have. For example, you might want to check that doing and then undoing an operation takes you back to the starting place, or that the size of an array stays the same. For some other ideas, see Scott Wlaschin’s Choosing properties for property-based testing.
- Picking a property-based testing tool. Although you could generate simple random inputs yourself for properties you want to test, you’ll probably want to use a tool to generate more structured inputs, or if you want to shrink examples.
- Getting your system into a state where you can use your property-based testing tool. The specifics will depend on exactly what type of tool you use, but you may need to mock or stub services your software interacts with, just as you would for example-based testing.
- Using your testing tool to exercise your system with many different random starting inputs and check whether the properties hold.

- Analysing your results. If any properties fail to hold, investigate what's causing the bug.

## What are the strengths and limitations of property-based testing?

Property-based testing saves time and increases confidence in your software, because:

- You can generate lots of tests for a given property without having to manually create each one.
- Properties tend to be more enduring, higher-level features of the software than specific examples. This makes tests more maintainable as they are more resilient to software changes.
- Random choice of inputs can find "unknown unknowns" that you didn't think to test for deliberately.
- As you exercise your system more thoroughly, you can encounter rare bugs before they turn up in production.
- Even before you run your tests, thinking about what properties your software should guarantee can help you write better software. Software with clearly defined properties is easier to reason about and debug.

However, property-based testing comes with its own challenges:

- Starting to use property-based testing requires a mindset shift and some upfront work in selecting your tools.
- You'll need to put in more work to identify and define useful properties. In some cases, you may need to clarify what your system should or should not guarantee (but you should probably be clear on this anyway!).
- There may be extra setup that you need to do before you can use your property-based testing tool, such as mocking external dependencies.

## What kinds of systems benefit most from property-based testing?

You can use property-based testing on any kind of software. However, it's a particularly strong approach in domains where you have expected properties that you need to guarantee. For example, databases have consistency models that define what they are allowed to do.

Some examples include:

- Financial transaction engines such as Formance.
- Databases like MongoDB and FoundationDB.
- Tasks like optimization work and rewrites, where you have some existing "oracle" system and you want to check that the new system satisfies the same properties.

- Blockchain platforms like Ethereum.

## Conclusion

Property-based testing lets you write a smaller number of more powerful, resilient tests than the traditional example-based approach, saving developer time and uncovering bugs in scenarios you didn’t think to test for directly.

Introducing property-based testing to your codebase can take some upfront work, but you can expect higher developer productivity and better confidence in your software.

---

Antithesis offers a powerful property-based testing platform for distributed systems. Book a demo for a walkthrough of how we test the systems no one else can.