Reframing the Liskov Substitution Principle through the Lens of Testing

Elisa Baniassad
University of British Columbia
Vancouver, Canada
ebani@cs.ubc.ca

Alexander J. Summers
University of British Columbia
Vancouver, Canada
alex.summers@ubc.ca

Abstract

In this essay, we explore a new pedagogical framing of the Liskov Substitution Principle (LSP). In addition to, or perhaps even in place of, teaching the specifics of the rule itself, we advocate an operationalised version of the rule: that a subtype must pass its supertype's *black box tests* for each of its overriding methods. We leverage the fact that black box tests should be written to capture conformance to a specification without overfitting or checking implementation internals (as would be checked by glass box tests). A type that violates the rules of substitutability will also fail a potential corresponding black box test for the supertype. Additionally, we argue that the over-strict nature of the classical LSP Postcondition Rule (which has been improved in subsequent work) can be a source of confusion for both instructors and for students learning this crucial concept for the first time. Pleasingly, many of the technical subtleties of this nuanced but important concept drop out naturally when thinking of substitutability via black box tests. We propose that this test-oriented means of teaching substitutability is a valuable alternative to the classical sense of checking the LSP, with the benefit of being intuitively accessible to students.

CCS Concepts: • **Software and its engineering → Software design engineering**.

Keywords: Testing, Substitutability, CS-Education

ACM Reference Format:
Elisa Baniassad and Alexander J. Summers. 2021. Reframing the Liskov Substitution Principle through the Lens of Testing. In *Proceedings of the 2021 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3484272.3484965

1 Introduction

The Liskov Substitution Principle [6] (hereafter, LSP) tells software developers that to be substitutable, a subtype must satisfy three rules (the following is a paraphrased quote from Liskov and Guttag's book *Program Development in Java: Abstraction, Specification and Object-Oriented Design* [5]):

- The *Signature Rule*: Subtype methods must be compatible with the signatures of corresponding supertype methods.
- The *Methods Rule*: Calls to the subtype's methods must *behave like* calls to the supertype's methods.
- The *Properties Rule*: The subtype's methods must preserve all properties provable about supertype objects.

The Methods Rule is the best known in education. It is often taught as originally presented [6] by breaking it down into two parts. The *Precondition Rule* states that subtype methods mustn't strengthen preconditions, while the *Postcondition Rule* states that subtype methods mustn't weaken postconditions. The *Signatures* rule is checked by the compiler so we will not address that through the rest of this essay.

It's hard for students to connect to the LSP. The Precondition and Postcondition Rules are tough to remember and difficult to operationalise, and at least in our anecdotal experience not often considered when students move on to make their own subtypes. For students new to software engineering, the implications and power of a specification are still not felt.

Furthermore, despite the importance of these original definitions [6], subsequent work identifies subtly different logical formulations of the Methods Rule which are more flexible and closer to the intuition of the original principle [2, 5], while the application of the Properties Rule to general object invariants and subtypes has led to an extremely diverse variety of different technical challenges and competing methodologies for tackling them [3]. We believe that the subtle clashes between the classical LSP rules (which are still often taught in practice, and are ubiquitous in verbal explanations of the principle) and programming scenarios arising in practice can cause an additional source of confusion for students.

Further along in Liskov and Guttag's same book is a section on *Testing a Type Hierarchy* which states: *"When there is a type hierarchy, the black box tests for a subtype must include those for the supertype." [5]* It then sets out a way for that to work — that all the black box tests of the supertype should be *reusable for a subtype* by simply rewriting initialisation code to call subtype constructors, instantiating the subtype instead of the supertype. Inspired by this idea, we have explored an alternative formulation of the notion of substitutability motivating the LSP itself. Our alternative formulation can be boiled down to the following condition: ***"A subtype has to pass all the supertype's (black box) tests. If it doesn't, it's not substitutable!"***

We see potential pedagogical benefits of framing substitutability in terms of *tests*, rather than constraints on subtype *specifications*. Tests are, after all, taught early in most programming curricula. In many courses, students are taught to write tests against a specification, and to write the tests prior to implementation. Passing tests can even translate into grades for students in some automatic-grading settings. Thus, software construction/software engineering students are familiar with the concept of tests, and their importance to software processes. That's not to say that students necessarily appreciate the concepts of test driven development, or adhere to them in their own practice, but testing (including specific pass and fail outcomes) is at least a concrete notion they can easily latch onto. So simplifying the complex formalism of the LSP into an easy to digest rule about tests would be a win.

In this essay we present a way of presenting the LSP through this lens of testing, exploring whether such an approach effectively captures all parts of the LSP, and has the potential to avoid the common pitfalls students experience when trying to learn it.

2 LSP Primer

The Liskov Substitution Principle (LSP) has various definitions in different sources. The principle was originally described in the seminal paper by Barbara Liskov and Jeanette Wing [6]. Liskov and Guttag then reworded it, and refined the Postcondition Rule (in a style which had been identified by Krishna and Leavens [2]) for a pedagogical audience in a text book called Program Development in Java: Abstraction, Specification and Object-Oriented Design [5]. It was their verbal formulation that we used in the introduction. In this section we walk through the original formulation via the language of the textbook; we will discuss variations of the definitions (in particular, for handling postconditions) in later sections.

As a basis for discussion, let's set up a type called `Super` - it has one method `behave(int i)`. Its implementation maintains a count of all the times `behave(int i)` has been called (e.g. in a simple `int`-typed field). We'll be using `Super` to look at the three LSP components (Signatures, Methods, Properties). In that exploration, we will introduce some convenient subtypes to illustrate each rule.

2.1 The Methods Rule

The Methods Rule is taught most in classrooms. It looks at methods in terms of a method's preconditions and postconditions. It states that for a given overriding method, the precondition should not be more strict (meaning that the subtype's method should be callable in at least all the situations where the supertype's method is callable), and the post condition should not be weaker (meaning it should not produce effects that the client doesn't expect). Should either of those be violated, the substitutability principle is not preserved. This is of course because clients of the type will not be able to use the subtype as they would the supertype: they will meet with unexpected restrictions and/or unwanted consequences of using the subtype. That means, of course, that the subtype isn't actually substitutable - it's not a usable substitute for the original.

In our experience, most educators concentrate on call situations in which supertype preconditions are satisfied (as should be required of a call against the supertype's interface). However, this focus naturally misses two important subtleties. Firstly, the classical Postcondition Rule is *not* qualified by the restriction to calls satisfying the precondition. As we elaborate on in Sec. 4.1, this makes the simple symmetrical formulation of this rule logically more restrictive than necessary in general; from a teaching perspective the requirement becomes overly-abstract as a consequence. This deficiency has been observed and resolved in later formal rules (both by Krishna and Leavens [2] and appearing in Liskov and Guttag's book [5]), which restrict the attention of the Postcondition Rule to calls satisfying the *supertype* precondition. This has the effect of allowing a subtype which chooses to weaken its precondition to have *no restriction* from the Postcondition Rule on how the function behaves in these newly-allowed call situations (c.f. Sec. 4.2).

In Table 1, there is a root type (`Super`) that has a single method. Its specification is shown in the row for `Super`. It accepts a certain range of inputs, and produces a certain range of outputs. We are using ranges for most examples here because this is a popular way to teach the LSP. The notions of weaker and stronger pre- and postconditions are, anecdotally, more involved and difficult to understand, being grounded in formalism. We believe that many instructors present examples that are similarly range-oriented, because on such simple examples the notions of weaker and stronger constraints align with wider or narrower ranges.

In Table 1 there are four subtypes specified: three are substitutable (`WiderPre`, `NarrowerPost` and `SubtractsFive`) and two are not (`NarrowerPre` and `WiderPost`). The reasons for the passing and failing in terms of the Methods Rule are provided. Most notably, while `SubtractsFive.behave(int i)` doesn't pass the *classical* formulation of the Postcondition Rule; simply checking the two postconditions alone yields the not-valid implication: $\texttt{result} = \texttt{i}-5 \Rightarrow 0 \leq \texttt{result} \leq 5$.

Table 1. Examples of Substitutable and Non-Substitutable Subtypes for the method `behave(int i)`

Classname	**Precondition**	**Postcondition**	**Pass or fail LSP?**
`Super`	$5 \leq i \leq 10$	$0 \leq result \leq 5$	This is the supertype
`WiderPre`	$2 \leq i \leq 20$	$0 \leq result \leq 5$	Passes LSP! The precondition is allowed to be wider.
`NarrowerPost`	$5 \leq i \leq 10$	$result = 3$	Passes LSP! The postcondition is allowed to be narrower
`NarrowerPre`	$6 \leq i \leq 8$	$0 \leq result \leq 5$	Fails LSP! The precondition must not be more restrictive.
`WiderPost`	$5 \leq i \leq 10$	$0 \leq result \leq 8$	Fails LSP! The postcondition must not allow more outcomes.
`SubtractsFive`	$5 \leq i \leq 10$	$result = i - 5$	Fails the classical Postcondition Rule, but passes the LSP!

However, the subtype implementation *is* in fact substitutable and passes the LSP. Understanding this relies on careful teaching of how to properly consider relational postconditions, as we explain in Sec. 3.5.

2.2 The Properties Rule

For illustration of this rule, we will introduce three subtypes:

- `WrappedCount` stores in a wrapper object the count of times its method `behave` is called, updating it via getters and setters.
- `FlakyCount` increments the count by one except if called with parameter values that are at least 10, in which case it increments its count twice.
- `ResettableCount` adds an additional method `resetCount` which resets the counter.

Recall that `Super` has the property that it maintains a count of each time its method `behave(int)` is called. For the LSP to hold for a subtype, the subtype also has to maintain that property. `WrappedCount` *in principle* maintains the property, but the notion of "preserve" is subtle; since calling the wrapped object will necessarily happen in a state while the property is temporarily violated, whether the Properties Rule is satisfied or not depends on exactly when object invariants can be temporarily broken, and when they must be reestablished. There is a large design space here (see e.g. [3] for a summary), and exactly which invariant discipline fits well may differ for different software engineering styles. By contrast, `FlakyCount` definitely doesn't maintain the property, since some sequences of calls to the subtype will produce different (out of sync) counts. `ResettableCount` also doesn't maintain the property: calls to the *additional* method will violate the original meaning of the count.

3 Common Learning Pitfalls

In our experience, students tend to be tripped up in multiple ways by the LSP, ranging from simple issues to subtle ones; some are fundamentally connected to the actual forms of the definitions taught to the students, which can potentially introduce false impressions or intuitions which do not carry over to practical examples.

3.1 Must Subtypes Be Clones of the Supertype?

One of the features of the LSP is that subtypes can be different from their supertypes, but only in ways that won't be problematic for a client of the supertype. Before students begin to deconstruct the LSP, their first instinct is to think that the LSP dictates that a subtype must be a tight reworking of the supertype, adding nothing, removing nothing. We believe (though don't have substantiation) that this contributes to students' dismissal of the LSP as a useful principle – we get the sense, from the questions they ask in class, that they see it as too restrictive and hampering, and that "in real life, subtyping can't possibly conform to such constraints".

Students have difficulty grasping the idea that a subtype satisfying the LSP can in fact both add behaviour and specialise existing behaviour. The way the principle is written makes this complex to reason about. Students see the rule as telling them what subtypes *cannot* do, but not making clear what they *can* do.

3.2 Looser vs. Tighter What?

Students very frequently confuse which condition can be broadened and loosened, because they don't seem to anchor the underlying notion of it - that the client will be unhappy for some reason - to these abstract rules about preconditions and postconditions.

There has even been work on making little tricks to remember *which is which* - which condition can be broadened and which can be narrowed [1]. In that work, the overriding method is to be represented as either a happy face or a sad face, with the mouth representing the subtype's overriding preconditions and postconditions. If a mouth is wider at the top, and narrower at the bottom, it forms a smile (hence, happy – and the LSP is satisfied), but if the mouth is narrower at the top, and wider at the bottom then the mouth forms a frown, which means the LSP is sad, and not satisfied. This is a trick that has been shown to be effective in recall of that one aspect of the LSP, but it hasn't been shown to improve students' understanding of the *semantics* or spirit of of the rule. Students likely still wouldn't be able to answer why the preconditions and postconditions should form a smile versus a frown. So while it is possible to ease recall about the rule itself, there's little evidence that it improves the intuition around substitutability, besides just freeing up students from having to constantly look up which precondition could be strengthened versus loosened.

3.3 What about Offset Ranges?

The example we've provided presents the specification for the pre-condition and post-condition for our `Super` class as a pair of ranges – so you know what's inside or outside the range.

This is straightforward for students if the ranges are very clear – for instance if the precondition allows values from 5–9, and the subtype precondition allows values from 4-10, then students can identify that this is a widened range. If the subtype precondition allows only values from 6-8, then they can identify that this is a narrowed range and hence stronger precondition.

However, we have observed that students are less good at knowing what it means if ranges are offset — for instance, if the postcondition specifies that numbers 5–9 are produced by the method, and the subtype postcondition specifies values from −10–4, the student may think this is a widened postcondition because the postcondition allows, technically, *more numbers*. Pedagogically, this requires careful explanation, possibly requiring an appeal to mathematical sets to invite students to consider whether the set of allowed states for the supertype's precondition is completely included in the set of allowed states for the subtype's precondition. As educators we might try to invoke the student's intuition about how a client might be *surprised* when being faced with constraints they were not expecting. But because the student is trying to map this back to narrowing and widening pre- and postconditions, their focus is not on the ideal learning outcome about client-side usage, but instead is fixated on the semantics of range changes.

3.4 What about Specific Outputs?

What if instead of the method producing a strict range of numbers, the method produces a specific string value? Students have been seen to be very confused about what happens if there is still a specific string, but the string is slightly changed – especially if it is a *subset* of the supertype's string. For example, if the supertype method produces *abbbba*, and the subtype method only produces *bbb*. Since the subtype's string is *within* the supertype's string, is that narrowing? Broadening? The fixation on ranges, and broadening and narrowing, will not help them understand that if the specification says to produce this one specific string, then the subtype is forced to produce precisely that also.

3.5 What about Relational Postconditions?

The original formulation of the LSP has a pleasingly-symmetrical sound to it: when we override a method, preconditions may only get weaker and postconditions only stronger. The former condition naturally translates to an analogous property of sets of states which can be simply visualised: the set of allowed input states may only grow. By symmetry, one is led to an intuition that what the LSP tells us about postconditions is that the possible output states can only shrink. Unfortunately, this neat symmetric intuition is misleading in general. As programming languages, specification languages such as JML and analysis and verification techniques have been better studied for object-oriented languages, is has become clear that many postconditions important in practice *cannot* be simply understood as a constraint on output ranges. For example, consider the following example function for a class with an `int` field `x`:

```
public class Grower {
  public int x = 0;

  public void grow() {
    this.x = this.x + 1;
  }
}
```

One meaningful property of this function is that[1] the resulting value of `this.x` is definitely larger than its *original* value on calling the function. A more-specific guarantee is that the value gets exactly incremented by one. Neither of these can be meaningfully expressed as a constraint purely on the ranges of possible output values; the allowed values *depend* on inputs (in this case, field values, but these could also be parameters) to the function, for example expressing that a setter function sets a field to the value of an input.

In general then, postconditions are not simple single constraints on sets of allowed output states, but rather describe a *pointwise relation* between *each* input state and the output states it is allowed to be mapped to. Specification languages such as JML [4] describe such relational postconditions using *two-state assertions:* a postcondition such as `x.f > old(x.f)` for `grow` captures the increasing guarantee, while a stricter postcondition could be `x.f == old(x.f) + 1`. Such postconditions express the required *relation* between the value of `x.f` in the function's post-state (when it returns), and the value it *had* when the function was called (using JML's `old` construct to wrap expressions to be considered/evaluated in the pre-state). If such a function has no precondition, the set of possible final values of `x.f` is unconstrained, but this is *not* the same as writing a postcondition `true`: the function (and all its overrides) must actually satisfy a fairly strict condition. In particular, thinking of the Postcondition Rule as symmetrical to the Precondition Rule (i.e. not growing output sets instead of not shrinking input sets) leads to the *wrong understanding* of relational postconditions. To see this concretely, imagine an override of `grow` in a subclass which simply does nothing (has an empty function body). This produces exactly the same set of possible output values, and so with a "range of outputs" intuition for the LSP might appear

[1]Note that we ignore overflows for the sake of simplicity, here.

to be a valid override. But, since postconditions are really relations and not sets, this is the wrong conclusion; the function's behaviour doesn't belong to the relation described by `x.f > old(x.f)` and so must be rejected.

This is as much a teaching pitfall as a learning pitfall. As educators, we tend towards simple examples based on e.g. ranges of values (without relational postconditions) as a way to teach the LSP because they simply convey the symmetrical intuition embodied in the classical Precondition and Postcondition Rules. However, in doing so we compromise on conveying an important but subtle point which the classical Postcondition Rule does not make explicit: when a postcondition expresses a relation (as is typical for mutator functions of a class such as our `grow` function above), the correct underlying notion is that an override's postcondition must describe an equal or stricter *relation*, not a subset of output values. The range-based examples from Table 1 (which we believe to be conceptually analogous to those used in many undergrad courses) can unfortunately serve to further reinforce the impression that the Postcondition Rule is symmetrical and analogously expressible in terms of output sets, since for postconditions that do not depend on input values, the issue cannot be seen. But this understanding breaks down when moving to slightly more-complex specifications and functions, such as our `grow` function, or more generally any specification which reasonably-precisely pins down the effect of a mutator function whose effect is not constant (depends on inputs).

4 Subtle LSP Concepts That Are Often Not Conveyed to Students

Because of the educator's instinct to make the LSP digestible by explaining it as ranges, some aspects of substitutability are not easy to convey. This could be our own weakness as instructors, but in our courses we limit examples to those we can straightforwardly convey using input and output sets. We know as instructors that the LSP covers all substitutable methods including those that don't conform to that format, and we've presented here some of the misunderstandings that can arise due to that adherence. However, there are also some more interesting and subtle points about the relationship between supertype and subtype methods that get lost in translation to range-level messaging.

4.1 Postcondition Meanings Should Be Relative to Preconditions

A further asymmetry between necessary requirements for preconditions vs. postconditions arises naturally from the observation that what a function's postcondition states should only actually matter for situations in which the corresponding function can be called. In particular, there is no need to impose any requirements on what a function's postcondition states for hypothetical call situations for which the function's precondition did not hold.

For example, consider the following weakly specified `foo` function and a possible override (we show only the function declarations without bodies, since we are only interested in the specifications):

```
public class C {
  // requires: x > 0.0              // pre
  // ensures: result > 0.0          // post
  double foo(double x) { ... }
}

public class D extends C {
  // requires: x > 0.0              // pre'
  // ensures: result == Math.sqrt(x) // post'
  double foo(double x) { ... }
}
```

The Precondition Rule (expressed logically as $\texttt{pre} \Rightarrow \texttt{pre}'$) enforces exactly what one needs for calls via a supertype interface to avoid precondition violations in a subtype; it holds trivially for this example, since the two preconditions are the same. However, the corresponding (symmetrical) Postcondition Rule's condition $\texttt{post}' \Rightarrow \texttt{post}$ *fails* for this example: while the library function `Math.sqrt` only returns non-negative square roots, it is not true in general that the square root of a floating point number need be strictly positive (and in fact, there isn't even a well-defined floating point result for negative values of `x`). This failure is artificially induced by the fact that the rule considers postconditions in isolation, whereas in practice we only need a relationship between the two postconditions *in states corresponding to valid calls* (i.e., states in which `x` is known to have a strictly positive value, due to the *precondition* enforced when actually calling `foo`).

A simple way to weaken the Postcondition Rule to restrict its attention to only *valid calls* to the overridden function is (using JML-like syntax) to instead require $\texttt{pre}' \land \texttt{post}' \Rightarrow \texttt{post}$, which allows the override described. Despite breaking the symmetry of the classical formulation of this rule *as it is typically taught*, this weakening is arguably a more-direct transcription of the original Methods Rule (emphasis changed): "*Calls to the subtype's methods* must behave like calls to the supertype's methods".

4.2 Weakening Preconditions Should Yield Implementation Freedom!

The Precondition Rule allows for an override of a function in a subclass to have a strictly weaker precondition, allowing for the new function to be called in *new ways the supertype function cannot.* Despite the fact that a client of the supertype's interface will never call the function in these new ways, the classical Postcondition Rule (even in its weaker form we argued for in the previous subsection) imposes strong constraints on what the override can do in these new cases. As has been observed by Krishna and Leavens [2], the intuitive description of the LSP as "avoiding surprising behaviour" actually leads naturally to a weaker-still Postcondition Rule, reflecting that if an override chooses to weaken a function's precondition, the supertype need not impose *any restrictions* on how the function behaves for *newly-permitted* calls. In other words, the override's postcondition need only be restricted in call situations for which the *supertype's precondition* holds, leading to a weaker-still alternative to the Postcondition Rule [2]: $\mathtt{pre} \wedge \mathtt{post}' \Longrightarrow \mathtt{post}$.

Extending the same foo example from the previous subsection, we illustrate this improved Postcondition Rule on a further potential subclass and override:

```
public class E extends D {
  // requires: x != 0.0
  // ensures: (x > 0.0 ==> result == Math.sqrt(x))
  //       && (x < 0.0 ==> result == Math.sqrt(-x))
  double foo(double x) { ... }
}
```

The precondition of foo in E is weaker than that in D, now allowing for strictly negative input values that D's specification did not permit. For these values, the specification chooses a postcondition guarantee which is different from that of D, which is conceptually fine since clients of D will have no expectations about these newly-supported use-cases of foo.

This weaker Postcondition Rule more-accurately captures the notion of what it means to *avoid surprises*: at least whenever one can call via the supertype's specification, an override will produce results allowed by the supertype's postcondition; for other calls only possible via the subtype's specification there is no danger of surprise, and this more-permissive rule gives complete freedom to the implementation of a subtype for these new cases.

5 Thinking in Terms of Tests

In the introduction we gave our student-friendly reframing of the LSP. To restate with a few more words, our principle is:

> *For a class to be substitutable for its supertype, it must pass all the supertype's black box tests.*

Tests, are, after all, a programmatic way to capture a specification. The rule plays on what Liskov and Guttag stated in their *Testing a Type Hierarchy* preamble: that a subtype's tests must include all the black box tests from the supertype (with some pruning). They never explicitly wrote that "if it doesn't pass those supertype tests, then it's not substitutable", but we make that connection here.

For our test-centric rule to work technically, we will need that substitutable subtypes pass all the supertype's black box tests, and that unsubstitutable subtypes fail at least *some* supertype tests.

Applying this rule in practice relies on a type having a somewhat comprehensive test suite. The sense that we want to espouse to students is that a test suite should capture the important behavioural properties of a type, and check for conformance to its specification. In the rest of this section, we look at minimal tests that can catch LSP violations, to demonstrate that the tests passing and failing can indeed correspond to the LSP passing and failing; a full set of black box tests should naturally be substantially larger.

In the rest of this section, we will look at whether tests catch LSP rule violations, and also what it means to say that a subtype should pass all a supertype's black box tests. We will skip the Signature rule, because the compiler typically checks that for all programs.

5.1 Can Tests Catch Methods Rule Violations?

Let's look at whether tests that thoroughly exercise a specification also result in the right passing/failure outcomes for subtype substitutability. We continue with the Super example from above. Super has a single test for its method, consisting of two assertions, one checking the bottom of the input range, the other the top, and both checking that the output is within range:

```
public void twoRangeTest() {
  Super s = new Super(); // or subtypes here!
  int i = s.behave(5);
  int j = s.behave(10);
  assertTrue(i>=0 && i<=5);
  assertTrue(j>=0 && i<=5);
}
```

Note that we're not checking what happens at input values 4 or 11. The precondition for Supertype.behave(int) tells us that the requirement for running the method is that the numbers be from 5-10 inclusive, so we are only testing values satisfying it.

We then instantiate s as all four of the subtype classes listed in the table, and run the test using each of these instances. Three of our give example subtype instances pass the test (as expected, these are WiderPre, NarrowerPost and SubtractsFive), and the other two (the ones violating the LSP) do not. But they fail in different ways, so let's consider catching each violation of the methods rule: narrowing the preconditions, and widening the post condition.

5.1.1 Catching a Strengthened Precondition.

Preconditions bring up an important caveat in the pedagogical approach for the test-mindset for the LSP. We want specification conformance tests to fail if the subtype does not conform to the supertype's specification, and we do not want to *over test*, because in doing so, we would be limiting the range of expansion for substitutable types. Restated: We want to run tests only that themselves honour the preconditions of a method, because if we wrote a test to check for specific

```
public class NarrowerPre extends Super{
    // requires: i to be between 6 and 8
    // effects:  will return a number between 0 and 5 (inclusive)
    public int behave(int i){
        assertTrue(i >= 6 && i <= 8); //Force a test failure for precondition violation
        return i-6;
    }
}
```

Figure 1. NarrowerPre code listing

behaviour given a precondition violation, there is no specification for what the result of the test should be. In fact this is the point: setting up the precondition means the developer is saying "buyer beware" with the method: if called outside these situations there's no telling what might happen. Things might work, but also might not.

Not testing outside the bounds of the precondition is actually a subtle point educationally. Students often instinctively err on the side of checking that correct behaviour doesn't happen, if the precondition isn't met. But actually no such test should exist, since if it *were* specified what happened when the precondition wasn't met, then that would effectively write additional constraints into the specification: constraints that would then need to be maintained by subtypes. In turn, this would mean that the subtypes could not broaden the precondition!

But this poses us a problem in terms of teaching the LSP through tests, because we want the supertype tests to fail if the subtype narrows the precondition. The trouble is, we don't know how they will fail, and they might even look like they pass. So for pedagogical reasons, the code actually does have to do something failure-inducing if the precondition isn't met such that the test fails. It's not specified what it has to do (that's the point!) - it just can't look like "good" behaviour and result in an erroneously passing test. This entails a kind of pedagogical scaffolding, necessary to illustrate the point. In a real situation, if the precondition isn't met, there may be no test that can capture what's supposed to happen: in reality such code may e.g. fail silently (this is a risk with all programmatic behaviour, but is especially true for preconditions, and potentially for postconditions if their impacts are not immediately felt).

Appropriate scaffolding might look like the code in Figure 1, in which a test assertion is used to force failure; this is an approach becoming more prevalent in software development.

5.1.2 Catching a Weakened Postcondition.

Catching a weakened postcondition is more straightforward than catching a strengthened precondition. The test has to check within the expected range for the result, and fail if the result is outside that range. The tests written for the supertype perform that check with no modifications. And it's for this reason that `WiderPost` fails the supertype tests.

5.1.3 Verdict on the Methods Rule.

Running the tests that checked conformance to the supertype's pre- and postconditions results in subtype instances passing the tests when substitutable (including when the precondition was widened, and the post conditition was narrowed) and failing tests when not substitutable (when the precondition was strengthened and an exception thrown to assert the precondition, and when the postcondition was strengthened).

5.2 Can Tests Catch Properties Rule Violations?

The test for the Properties rule for `Super.behave` needs to check that the number of calls to `behave` matches the count. We can assume that there is a getter for the count indicator (`getCount`) such that the count is observable, and hence an externally provable property about the class. The test can be as simple as:

```
public void countTest() {
  Super s = new Super(); // or subtypes here!
  s.behave(6);
  s.behave(10);
  assertEquals(2,s.getCount());
}
```

Of course, this isn't an exhaustive test, but works for our purposes; the usage of the boundary value 10 turns out to be significant[2]. We would run the test for the subtypes by changing the initialisation code. The test would naturally pass for `WrappedCount`; by employing testing we side-step questions about exactly when the property should be enforced as these expectations become implicit in the way the test itself was written for the supertype. For `FlakyCount` the test would naturally fail as expected.

For `ResettableCount` the situation is more subtle: the test itself will pass because the newly-added subtype function is never called. This is a subtlety that our direct testing methodology cannot immediately uncover; the need to produce an appropriate new *sequence* of calls to exhibit the bug goes beyond behaviours that a simple supertype test can capture.

[2]Such important values could also be uncovered by random/fuzz testing, of course.

Instead, additional testing in the *subtype* introducing the new functionality is necessary.

A natural approach would be to write tests in the subtype that (after calls to a number of its functions) check the supertype's invariant for preservation. That way, if the subtype inadvertently violates the properties rule for the supertype, these new tests will uncover the violation. By exploiting features such as `@Afterall` in JUnit, one can generate families of tests which perform additional checks for simple single-state object invariants.

For complex properties governing the evolution of objects, this is a limitation of the test-oriented approach – in particular, for the combination of such properties with scenarios in which an object may be aliased via both supertype and subtype, it may ultimately be difficult to write tests that expose the violations. Nonetheless, we believe that even understanding this limitation may be an important pedagogical message for students, motivating the need to carefully write *new* black box tests specifically for subtypes that add additional functionality.

5.3 Testing vs. Proving Substitutability

Our test-oriented approach to capturing the LSP straightforwardly tells students that if a subtype is failing a supertype's black box test, that it is also failing the LSP. This is a natural consequence of running the supertype's tests on the subtype. No specific LSP-check tests like those we wrote for illustrative purposes, need to be written, beyond expanding one's testing practice to ensure that precondition violations are caught somehow. This is analogous to finding bugs with tests: if a test fails, you *know* you have encountered a bug.

But what if the subtype passes all the supertype tests? Does this mean, unequivocally, that the subtype is *really* passing the LSP? Just as it is true that a type passing all its own tests is not necessarily bug free, a subtype passing all its supertype tests need not *necessarily* pass the LSP Method's Rule; we could be missing its point of failure with the current set of supertype tests. The more tests that a type has (and the more comprehensive they are), the more confidence can be built with respect to both bugs and substitutability (which, if failed, is itself a cause of bugs). So in this way, testing is a convenient and intuitive approximation for ensuring the LSP holds in a subtype, although not a complete *proof* that the subtype maintains substitutability.

6 Can Tests Help Students Avoid the Common Pitfalls?

In this section we revisit the learning pitfalls we outlined earlier, considering how using the testing approach might change the landscape for students or educators.

6.1 Subtypes Are Not Clones

Our first pitfall was that students erroneously believed that subtype methods had to effectively be clones of the supertype method to be able to be substitutable. When presented with the rule that *a subtype must pass all the supertype's black box tests* students may, initially, still have that impression: superficially, passing all the tests of another type does sound as those those two types should more or less be the same. But using the testing approach, we can leverage the precise definition of a black box test: we can point out that that students shouldn't be writing black box tests that overfit to implementation details.

Students may also initially trip up by thinking that that the rule is symmetrical, but of course it is not: New tests written for the subtype do not have to be passed by the supertype. This provides an operational, and very practical, way for students to concretise a subtype specification without getting bogged down in formalism.

6.2 Questioning the Need for Learning the Precondition/Postcondition Rules

Given our new alternative formulation of the LSP via testing, it is natural to question whether our condition (passing supertype black box tests) could simply *subsume* the need to teach the classical formulation of the LSP via precondition and postcondition constraints. Yes, if students go on to a career of reasoning about formal subtyping rules, they will need to have a grasp of what the constraints are on preconditions and postconditions in subtypes. But certainly starting out, we want students to have an *operational* understanding of whether a subtype is substitutable or not.

Focusing on passing/failing supertype tests can give them this operational grasp. It gives an automatic way to check whether they have developed a subtype that is substitutable: run the supertype black box tests on an instance of the subtype. If any fail, then substitutability did not hold. If no tests fail, then there are no identified situations in which substitutability fails. The strength of confidence in substitutability depends on the strength of the test suite.

Note that we are certainly not advocating that understanding a function's specification becomes unimportant. Indeed, the correct definition of a valid black box test for a function is intrinsically reliant on a function's specification, using the precondition to define suitable test inputs, and the postcondition to define the right conditions to test (and how to avoid over-fitting and producing a glass box test accidentally). However, it seems appealing to investigate the effectiveness of our testing-oriented approach as an introduction to how to get substutability correct in its own right.

6.3 Capturing Offset Ranges

The narrowing/widening/strengthening/weakening language leads to student confusion around what range-changes are permitted for subtype pre and post conditions. In addition to not being able to remember which should be strengthened and which can be loosened (as discussed above), they also become confused about whether it is the size of the range, or the span of the range that matters for the rule. As we detailed, a student may feel like a subtype range that is smaller in size, but doesn't fall within the supertype range, has, in fact, *narrowed* the range. This is an irritating confusion that is an artifact of trying to remember a formal rule without necessarily really grasping the formalism. Assuming the range was specified for the postcondition, we could write a test to capture the supertype's constraint like this:

```
public void rangeTest() {
  Super s = new Super(); // or subtypes here!
  int i = s.behave(5);
  assertTrue(i>=0 && i<=5);
}
```

If the result of the call to `behave` fell outside the supertype's postcondition-allowed range, the test would fail. This test-oriented formulation releases students from wondering about range size vs. range span, focusing their thinking on whether subtypes conform to the expectations on behaviour laid out by the supertype.

6.4 Checking Specific Outputs

Our testing-oriented approach also quickly helps illustrate why a supertype with postcondition guaranteeing output *abbba* cannot be substituted by a subtype guaranteeing *bbb*, even though it *looks* narrower. We could rewrite the postcondition using this test:

```
assertEquals(foo.call(),"abbba");
```

If the result of `foo.call()` is *bbb* then clearly this test will not pass. Since this is a check of the specification, and not an over-specific glass box test, it must pass for the subtype to be substitutable.

6.5 Capturing Relational Postconditions and Implementation Freedom

As we explained in Sec. 3.5, general postconditions capture properties not just of output states but *relations* between input states and outputs. This view on a function's requirements clashed with the traditionally-stated Postcondition Rule, but is very natural when thinking in terms of black box testing. Indeed, all a black box test does is check a relation between an input state (perhaps fixed, perhaps generated randomly to be any value satisfying precondition constraints) and the corresponding output state; a test harness can (and often does) provide facility to write test assertions relating to inputs (and this can be easily manually achieved with additional local variables).

For example, testing the `grow` function's relational postcondition `x.f > old(x.f)` is very simple:

```
public void growTest() {
  Grower g = new Grower();
  int r = (new Random()).nextInt();
  g.x = r;    // or use fixed initial value
  g.grow();
  assertTrue(g.x > r);    // r == old(g.x)
}
```

Testing that a subtype *also* passes this test is now as simple as changing the initialisation code to create an instance of the appropriate class; the view of postconditions as relations between input and output states becomes very natural when viewed through the lens of testing.

Similarly, in Sec. 4.1 we argued that properties of postconditions should (contrary to the classical Postcondition Rule) only be enforced for states in which the corresponding precondition held; this is again completely natural in the testing domain, since initialisation code will be written to ensure that a test only exercises valid initial states, Indeed, even the more-permissive rule of Sec. 4.2 comes for free: since our requirement is that *supertype* black box tests must pass for a new subtype, the initialisation code of these tests will naturally only explore behaviours that the supertype's precondition allows. As explained previously, this more-flexible rule allows for subtypes to exhibit *completely new* behaviour for situations in which the supertype function could not be called, as for our `foo` function in class `E`, which works on negative inputs although its supertype's function did not. Naturally, such new functionality should be exercised by *additional* black box tests for the subtype, but these need have nothing to do with the supertype tests.

7 Potential Drawbacks and Trade-offs

We've illustrated a variety of delicate facets of the notion of substitutability which are naturally well-handled by our test-oriented approach to teaching the LSP. What are the potential disadvantages to this approach? We identify two main potential weaknesses and discuss their implications here.

7.1 Black Box Testing for Preconditions

As we discussed earlier, postconditions are naturally checked by tests. However, as outlined in Sec. 5.1.1, developer practices for testing preconditions may vary: in particular, for unit testing a function, preconditions may simply be implicit in the initialisation code (generating values to conform to these preconditions without an explicit check), or may be included as additional debug asserts to check for mistakes in the test initialisation code itself. If preconditions are (only) treated as implicit aspects of test initialisation code, checking the Precondition Rule is not something which arises naturally; in particular, when checking a supertype's black box tests pass for a subtype, the subtype's precondition wouldn't even enter into the picture.

On the other hand, developers may or may not include explicit precondition checks in the implementations of functions themselves: as either debug asserts (removed in production builds but enabled for testing), or as runtime checks which remain in the final code. In such development styles, our testing-based approach will naturally detect precondition strengthening during testing.

Instructors adopting our approach need to consider which treatment of preconditions their students have been taught to employ when writing unit tests. As a catch-all approach one can advocate adding explicit subtype precondition checks as `assert` statements when migrating unit tests from a supertype to a subtype.

7.2 The Need for a High-Quality Test Suite

Students will notice, and it should be highlighted, that testing for substitutability relies on a comprehensive test suite that exercises relevant qualities of the overall specification. This may not be fully attainable in practice, just like a complete test suite to check for correctness is impossible in general.

However, we believe it can be very informative for students to see specific examples of subtypes which pass a test suite but exhibit some new bug *exactly because* the test suite was missing a useful black box test. Indeed, complementary course content on the evaluation and generation of test suites might flag up such issues: if e.g. the branch coverage of a supertype's test suite on a subtype's function implementation turns out to be low, a student can consider carefully whether this indicates a weakness in the supertype's test suite to start with, a corner case which was not obviously significant for the supertype's implementation, or simply a new scenario which the supertype could not have encountered.

Additionally, motivating that tests should be divided into those that will be inherited by a subtype, and those that are specific to the specific class (glass box tests), can help students to think more systematically and in a more organised way about their test suites.

8 Personal Experience

We have taught the LSP through this lens of testing the past 3 years, and have garnered the benefits we outlined above. We observed that students were able to better understand the responsibilities of the subtype in a general sense, as opposed to being limited to rote recall of the details of the Precondition/Postcondition Rules. We still explain these rules, since they are so commonly-associated with the LSP. However, our primary teaching method is via this lens of testing, providing examples of tests that would pass and fail if the rule were violated. We do still examine students' proficiency in understanding the rules, and the results are stable: the same proportion of students (roughly 85%) were able to correctly indicate that "A subtype shouldn't set a narrower range of postconditions of its methods than its super type" should be true in both the cohorts prior to the introduction of this technique, and currently. Our anecdotal experience gives us confidence that students are able to learn the spirit of the LSP using this teaching method, which (as we have outlined) generalises well to more-complex examples and avoids many of the learning pitfalls we have illustrated in this essay.

9 Conclusions

In this essay, we have proposed reframing the Liskov Substitution Principle via tests: a subtype must pass all its supertype's black box tests. We rely on the definition of black box tests as checking conformance to a type's specification (only). We have illustrated a number of teaching and learning pitfalls related to subtleties of the LSP and its presentation, and demonstrated that testing can uncover a variety of violations of the LSP that are often subtle for students to fully appreciate through traditional methods. For instance, if a subtype's method produces outputs that were not expected by the specification of the supertype, then the test for those outputs would fail. We have shown that using this testing approach, we can capture the entirety of the methods rule of the LSP, as well as simple, non-aliasing scenarios for the Properties rule. In so doing, we can avoid the pedagogical pitfalls associated with memorising and internalising a formal, and heavily nuanced set of rules. We posit that this approach frees educators from simplistic examples that are typically employed for the sake of clarity about teaching the principle itself. We suggest to educators that this approach could be employed either on its own or in addition to teaching the more formal rule in the traditional way.

Acknowledgments

We would like to thank Paul Carter, Meghan Allen, Noa Heyl and Gregor Kiczales for their in-depth feedback, insights and perspectives on this work and its presentation.

References

[1] Elisa Baniassad. 2018. Making the Liskov Substitution Principle happy and sad. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. 17–20.

[2] Krishna Kishore Dhara and Gary T. Leavens. 1996. Forcing Behavioral Subtyping through Specification Inheritance. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Germany) *(ICSE '96)*. IEEE Computer Society, USA, 258–267.

[3] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. 2008. A Unified Framework for Verification Techniques for Object Invariants. In *ECOOP*. Springer.

[4] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. 2008. JML reference manual.

[5] Barbara Liskov and John Guttag. 2000. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[6] Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841. https://doi.org/10.1145/197320.197383