Danish University Colleges

Contract-Based Software Development: Inheritance and Frame Rules

Kongshøj, Simon

Publication date: 2016

Document Version
Other version

Link to publication

Citation for published version (APA):
Summary

This note explains how contract-based techniques interact with the classic object-oriented notion of inheritance, and how the subclass-superclass relationship presents some challenges to contract-based programming. We also investigate the concept of frame rules, which express the parts of objects that do not change when a given command is executed.

Introduction

When an object-oriented programmer writes a subclass of some existing class, it is usually the case that she wants to redefine some of the methods inherited from the superclass. However, this raises some questions about how the contracts of those methods should be affected: If contracts can only be inherited in their entirety, then that would be an unacceptably severe restriction on inheritance. On the other hand, if contracts could be changed without restriction, then the ability to “substitute equals for equals” would be lost. What we need is a disciplined technique for contract inheritance, one that respects the essence of the superclass-subclass relation.

Just like a class may be a subclass of another, we may view inherited and partially overridden contracts as subcontracts. A useful metaphor is that the subclass functions as a “subcontractor” for the superclass — in this view, the superclass must still be able to fulfill all its contracts, but it is free to outsource its work to subcontractors, as long as these also fulfill the base contract.
Contract Redefinition and Subcontractors

The central notion of contract-based programming is the “contract”; so named because it enforces obligations and confers benefits — just like legal contracts. As we have seen in numerous examples, the obligations of a method in some class can be viewed on the form that if the method’s preconditions are satisfied, then a correct implementation must guarantee that its postconditions are also satisfied. The benefits of the method can be viewed on the form that it is allowed to assume that the preconditions are satisfied (otherwise, the DbC tool will terminate execution), and never has to include any code for dealing with situations where they aren’t satisfied. The obligation placed on client code is that it must never call the method without satisfying its preconditions, and the benefit is that it is then allowed to assume that the postconditions hold.

It makes sense to start with a metaphorical, non-software example when investigating how “subcontractors” can be made to not interfere with a main contract. As a guiding example, we shall use a delivery company. Our hypothetical delivery company presents its customers with the following base contract for its service:

- **PRECONDITION**: weight $\leq$ 5 kg
- **POSTCONDITION**: deliverytime $\leq$ 3 hours
- **INVARIANT**: insurance $\geq$ 100,000

Under which circumstances would our delivery company be able to hire subcontractors, and still be considered acceptable by its customers? From the customers’ point of view, the actual courier service will be irrelevant as long as they don’t lose any benefits or can’t trust any of the obligations. In other words, a customer would accept a courier service that allowed weights of up to 10 kg (since that service would still be able to respect the obligations), or one that could guarantee deliveries in less than 2 hours (since that service would still be able to confer all the benefits), but wouldn’t accept one that could only deliver packages of up to 3 kg (because then the obligations wouldn’t be respected) or could only guarantee delivery within 8 hours (since that wouldn’t confer the benefits).

This indicates a broader principle, one that is in fact well-established in software engineering: The Liskov Substitution Principle, also sometimes simply named the Principle of Substitution, which corresponds to the theoretical computer science notion of behavioural subtyping. This principle states:

*It should always be possible to substitute an object from a derived class where an object from the base class is expected.*
In other words, this principle states precisely the kind of relation we would expect our hypothetical delivery company to have with its subcontractors: If the customers expect delivery of packages up to 5 kg in less than 3 hours, then it is possible to substitute a subcontractor which can deliver packages up to 8 kg in less than 2 hours, because such a subcontractor would still have all the properties expected of the main company. In the context of contracts:

- Preconditions may not be strengthened.
- Postconditions may not be weakened.

Strengthening preconditions corresponds to demanding stricter obligations on the client, and weakening postconditions corresponds to conferring smaller benefits on the client. In both cases, the result would be that situations now exist where it would be impossible to substitute an object from the subclass where an object from the base class was expected, and that would be a violation of the Liskov substitution principle. Consider the following two class skeletons:

```java
class A {
    //INARIANT I
    public void pip(){};
    //PRE p
    //POST q
};
class B extends A {
    //INARIANT I
    public void pip(){};
    //PRE p
    //POST q
};
```
For this subclass relationship to work, the following logical predicate must hold: \( p \circ p' \circ q \circ q' \circ I' \circ I \). In other words,

- If the precondition \( p \) of \( A.pip() \) is satisfied, then the precondition \( p' \) of \( B.pip() \) is also satisfied.
- If the result of \( B.pip() \) satisfies the postcondition \( q' \), then it also satisfies the postcondition \( q \) of \( A.pip() \).
- If the object state invariant \( I' \) is preserved by all operations in \( B \), then the invariant \( I \) from \( A \) is also preserved.

It follows that the preconditions in subclasses are or’ed together with those of the superclass, whereas its postconditions (and invariants) are and’ed together with those of the superclass.

The primary design options we have for overriding contracts is then that we can weaken preconditions and strengthen postconditions in subclasses, but we may never do the opposite. Recall that a class invariant can be viewed as the postcondition of all methods; it thus follows a similar logic to method postconditions.

Guarded Postconditions

Consider a classic Dictionary class, like the one provided by the Map ADT in the Java Collections library. One common design question about dictionaries' concerns whether it is legal to insert a new key-value pair in a dictionary for an already existing key, and whether it is legal to attempt to delete a key that isn’t in the dictionary. Although many dictionary implementations are quite lax about these constraints — typically simply overwriting an existing key-value pair in the first case and performing a do-nothing operation in the latter — let us assume that we have written a rather strict one. Its specification could include, among other things, the following contracts:

---

1The local customs of your preferred programming language’s culture might call them tables, maps or associative arrays, but at any rate, they’re the same data structures: Collections of key-value pairs accessed by key.
By the preceding discussion and the Liskov substitution principle, it should be perfectly all right to make a relaxed dictionary subclass and use in place of the strict one: Anywhere where the strict dictionary could be used, the relaxed one would have exactly identical behaviour; the only observable differences would occur in contexts where the strict one couldn’t even be used in the first place. It stands to reason that it should be possible for us to make a subclass with appropriately weakened preconditions.

However, consider the contract for put(). If our subclass was to weaken its precondition by removing the requires !has(key) constraint, then by its postconditions (which cannot be weakened), it would have to guarantee that a put() operation would increment the result of count() even when the key was already there. A put() implementation that implements this behaviour would clearly be impossible to implement.

This problem can be solved with guarded postconditions, which are a standard feature in both Eiffel and JML². From a logical point of view, the subclassing behaviour we would like can be summed up as follows:

- If the dictionary doesn’t have the key, then put() should guarantee that the result of count() is incremented by one.
- If the dictionary does have the key, then put() should not provide this guarantee.

With guarded postconditions, each postcondition can be qualified with a guard (which should express a precondition), such that the postcondition only guarantees to

²CodeContract doesn’t allow changing preconditions at all.
hold when the corresponding precondition holds. Then, while writing base classes, we
must think about which postconditions follow from which preconditions, and make
this implication explicit. Then, when a subclass weakens the preconditions, it can pro-
vide its own appropriate postconditions. Postconditions that are universal, in the sense
that they don’t follow from any specific precondition but should apply to all possible
implementations of the method in all possible subclasses, shouldn’t be guarded.

In the example of our dictionary base class (the strict one), the postcondition for
put() could be rewritten as follows:

```java
/*@ requires !has(key);
@ ensures has(key);
@ ensures (!\old(has(key))) ==> (valueFor(key)==value);
@ ensures (!\old(has(key))) ==> (count()==(\old(count())+1));
@*/
public void put(Object key, Object value);
```

In other words, the first postcondition has(key) should apply to all subclasses’ im-
plementations of put(), but the two others apply only if the precondition !has(key)
holds. The ==> operator should be read as logical implication.

As discussed above, adding new preconditions in a subclass corresponds to or’ing
them onto the base preconditions, and adding new postconditions corresponds to and’ing
them to the base postconditions. In this case, if we were to use the following contract
in our subclass’ put():

```java
/*@ also
@ requires has(key);
@ ensures \old(has(key)) ==> valueFor(key)==\old(valueFor(key));
@ ensures \old(has(key)) ==> count()==\old(count()); @*/
public void put(Object key, Object value);
```

Then the full precondition would become has(key) OR !has(key) (which is obviously
tautological — and perfectly valid, since this is a weakening of the precondition),
and the full postcondition would become:

```java
has(key)
AND (!\old(has(key))) ==> (valueFor(key)==value)
AND (!\old(has(key))) ==> (count()==(\old(count())+1))
```
AND \old(has(key)) ==> valueFor(key)==\old(valueFor(key))
AND \old(has(key)) ==> count()==\old(count())

Which is a strengthening of the postcondition. The new, full postcondition both states the postcondition of the base class and that of the subclass, and qualifies both scenarios depending on the precondition.

When creating classes with DbC that are intended for subclassing, it is therefore generally a good idea to always guard your postconditions with their corresponding preconditions.

Frame Rules

Recall how, in our work with the Q language, we gave each algorithm a specification that had an environment, a frame, and the pre- and postconditions. The frame specified the specific working variables that the algorithm would change, whereas all other variables were implicitly assumed to not change. So far, we have treated our JML specifications similarly: We specify which attributes have changes under a given command, and assume that all other attributes don’t have changes.

In a complete specification, we should also specify what isn’t changed. Such a specification of unchanging attributes is called a frame rule.

For example, recall the discussion of the original specification for put() in our queue ADT:

```java
/**
 * @ ensures size()==\old(size())+1;
 * @ ensures get(size()-1)==o;
 * @*/
public void put(Object o);
```

Does this tell the whole story of inserting into a queue? As we established, it does not: A pathological implementation could perhaps destroy existing elements in our queue. While demonstrating shallow copy and immutable lists, we then implemented what was essentially part of a frame rule:
/**
 * @ensures size()==\old(size())+1;
 * @ ensures get(size()-1)==o;
 * @ ensures (∀ int i; 0<=i && i<=size()-2;
 * @ \old(shallowCopy()).get(i) == get(i));
 * @*/
public void put(Object o);

Using this version with quantifiers and shallow copies, we have created a specification that also specifies that — apart from the newly inserted element — the queue should contain the same elements after a put() as it did before. It is tempting to believe that we have now created a complete specification, but there is in fact still some opportunity for mistakes in the implementation, because there is still one assumption we’re implicitly making: We’re assuming that the elements themselves haven’t changed. For a complete frame rule, we’d like to specify that:

- Queue elements before put() must have the same position after put()
- The elements themselves are unchanged.

Due to the reference semantics of the shallowCopy() operation, it cannot be used to specify the latter property. To gain such a guarantee, we would have to implement a deepCopy() operation, which has value semantics — it shouldn’t merely copy references, it should copy the objects themselves. Writing a deep copy algorithm is left as an exercise to the reader.

Would using immutable lists help? In a word, no. The equals operations we defined on immutable lists also just checked for reference equality, so with immutable lists we would also have some further work ahead of us. The Eiffel language provides deepClone and deepEqual operations, used precisely to express complete frame rules.

The big question concerning very detailed frame rules is: Is it worth it? The sheer amount of effort involved in writing full deep-equality tests is daunting, and in the context of an agile development methodology it would impose an inappropriate amount of up-front design and specification work. In many cases, the payoff could also be considered rather marginal: For deep equality to matter in a data structure like our queue, a mistaken implementation would have to explicitly modify existing elements, which is an extremely unlikely situation, and one more likely to result from malice than from mistake. However, for mission-critical components, or components that will be reused in millions of products, catching any bug before shipping may be worth even large amounts of extra effort. The textbook Design by Contract by Example suggests the following general approach:
• Add frame rules when there is evidence that client programmers misunderstand what a method does and does not do.

• Adopt a convention that all classes have an implicit frame rule that says that unless a postcondition asserts that a property changes, no properties change. Any unexpected changes are then symptoms of poor documentation or of bugs.

• When doing design reviews, check against both explicit frame rules and the implicit “nothing is changed” rule.

Exactly how much to specify in frame rules depends primarily on the project at hand and the development process in use.

**Literature**

Mitchell & McKim: *Design by Contract, by Example*, Addison-Wesley 2002