Introduction to Software Change Contract by Examples

Jooyong Yi

jooyong@comp.nus.edu.sg

National University of Singapore

1 About this document

The purpose of this document is to help readers understand software change contract at an intuitive level. To achieve it, we explain change contract mainly by examples. The examples we use are deliberately simple for the exposition purpose. Extending the application of change contract to more sophisticated programs would not be difficult.

Each change contract example we will show deals with a distinct common pattern of program changes. We will combine a few keywords such as `ensured`, `ensures`, `signaled` and `signals` in different ways to express different changes. It can be viewed that each combination forms a phrasal verb that describes a different kind of program changes.

In this tutorial, we introduce a specification language to describe change contracts for Java programs. Other languages should be able to be dealt with similarly. Currently, change contracts focus on method-level changes. Some changes are restricted to method bodies, some are restricted to method signatures, and some involve both kinds of changes. We will consider those changes one by one.

2 Basics of change contracts for purely behavioral changes

A change contract mainly specifies behavioral changes of a Java method of two different consecutive versions. As a concrete example of such changes, consider the `Cls` class in the following box. In its previous version, method `getStr` simply returns the `str` field of `String` type. However, returning a string as it is can be risky sometimes. A caller of `getStr` may assume that the returned string does not start or end with whitespace, not expecting a return value like “ SoC ”.

[Previous version]

```
public class Cls {
  private String str;

  public String getStr() {
    return str;
  }
}
```

[Change contract for `getStr`]

```
ensured !\result.equals(\result.trim());
ensures \result.equals(\result.trim());
```

[New version]

```
public class Cls {
  private String str;

  public String getStr() {
    return str.trim();
  }
}
```

So we want to tell the developer of method `getStr` to modify the code appropriately. Our requirement can be expressed as a change contract shown in the middle of the above box. In a nutshell, a change contract specifies how the behavior of `getStr` should change. To do that, a change contract typically compares the behavior of the previous version method with the behavior of the new version method.

The overall meaning of the above change contract is as follows. If there is an input I to `getStr` – in this example, an input to method `getStr` is simply an instance of class `Cls` because `getStr` does not take a parameter – that *ensured* in the previous version the given condition following the `ensured` keyword, “`!\result.equals(\result.trim())`”, then the same input I *should ensure* in the new version the next given condition following `ensures` keyword, “`\result.equals(\result.trim())`”. We say that an input I *ensured* a condition C when the following three conditions are met.

1. when given input I, the method associated with a change contract (in our example, `getStr`) terminates without throwing an exception *in the previous version*,
2. the given condition C (e.g., "`!\result.equals(\result.trim())`") is interpreted right before the method terminates, and
3. the above interpretation result of C is true.

Similarly, we also say that an input I *ensures* a condition C when the following three conditions are satisfied. The second and the third conditions are identical to before.

1. when given input I, the method associated with a change contract terminates without throwing an exception *in the new version*,
2. the given condition C is interpreted right before the method terminates, and
3. the above interpretation result of C is true.

The `\result` keyword. In the above example, we use a special keyword `\result` to refer to the return value of `getStr`. Recall that the given condition is interpreted right before a method terminates. Thus, we can access the return value of a method through `\result`. Expression "`\result.equals(\result.trim())`" compares the return string of `getStr` with its trimmed string.

You may wonder, what if an input I does not ensure "`!\result.equals(\result.trim())`" in the previous version in the first place? For example, imagine I is an instance of `Cls` whose `str` field is "SoC". If the new version is also given the same input I, then what kind of output should we expect from the new version? The answer is not explicitly specified in the given change contract. However, it is natural to assume that unless specified otherwise, the new version should behave in the same way as the previous version does when given the same input. Therefore, given the same input I, the new-version `getStr` should return the same output "SoC" as was returned in the previous version. While the notion of behaving in the same way ought to be formally defined, in this lightweight tutorial we will simply mean that return values of the previous-version and the new-version methods are the same.

3 More examples of change contracts for purely behavioral changes

We provide three more examples of change contracts. They will show how change contracts can deal with the diversity of Java method behaviors.

3.1 Removing an unexpected exception

A Java method can throw an exception. Yet, some of them are thrown unexpectedly. For example, in the following previous-version method `getStr`, a `NullPointerException` is thrown when `str` is `null`. However, a caller of `getStr` usually does not expect to get a `NullPointerException`.

[Previous version]

```
public class Cls {
  private String str;

  public String getStr() {
    return str.trim();
  }
}
```

[Change contract for `getStr`]

```
signaled (NullPointerException) str == null;
signals (NullPointerException) false;
```

or equivalently,

```
signaled (NullPointerException) str == null;
not_signals NullPointerException;
```

[New version]

```
public class Cls {
  private String str;

  public String getStr() {
    if (str != null) {
      return str.trim();
    } else {
      return str;
    }
  }
}
```

The simplest fix for this problem is not to throw a `NullPointerException`. Such a behavioral change can be expressed with the change contract shown in the middle of the above box. The overall meaning of the above change contract is as follows. If there is an input I to `getStr` that *signaled* a `NullPointerException` in the previous version under the condition, "`str == null`", then the same input I *should not signal* a `NullPointerException`. The latter then clause is a paraphrase of the direct interpretation of "`signals (NullPointerException) false;`". The same meaning can be conveyed more directly by using "`not_signals NullPointerException`". We say that an input I *signaled* (or *signals*) an exception E under a condition C when the following three conditions are met.

1. when given input I, the method associated with a change contract (in our example, `getStr`) throws exception E *in the previous version* (or *in the new version*),
2. the given condition C (e.g., "`str == null`") is interpreted right before E is thrown, and
3. the above interpretation result of C is true.

3.2 Suggesting an alternative exception

In the previous example, we applied a very simple fix. After the fix, method `getStr` returns `null` if the `str` field is `null`. Although a `NullPointerException` is not thrown, returning `null` is likely to cause an another unexpected `NullPointerException` in other places of a program.

[Previous version]

```
public class Cls {
  private String str;

  public String getStr() {
    return str.trim();
  }
}
```

[Change contract for `getStr`]

```
signaled (NullPointerException) str == null;
signals (NoStrException) true;
```

[New version]

```
public class Cls {
  private String str;

  public String getStr()
  throws NoStrException {
    if (str != null) {
      return str.trim();
    } else {
      throw new NoStrException();
    }
  }
}
```

A better solution would be to throw a checked exception. For example, the above updated change contract specifies that a custom checked exception `NoStrException` should be thrown in the new version. Accordingly, the new-version `getStr` throws a `NoStrException` when `str` is `null`. Also, notice that the signature of `getStr` now ends with "`throws NoStrException`" indicating that `NoStrException` is a checked exception.

The above code change can be expressed with the change contract in the middle that means the following. If there is an input I to `getStr` that *signaled* a `NullPointerException` in the previous version under the given condition, "`str == null`", then the same input I *should signal* a `NoStrException` under the condition of `true`.

3.3 Directly constraining the input domain of interest

Recall the meaning of the following change contract:

```
ensured !\result.equals(\result.trim()); ensures \result.equals(\result.trim());
```

It means that if there is an input I to `getStr` that ensured in the previous version the condition appearing after "`ensured`", then the same input I should ensure in the new version the condition appearing after "`ensures`". Notice that the above change contract constrains the input I only *indirectly*. It does not directly constrain a component of input, e.g., the `str` field.

It is handy in many cases to constrain the input indirectly. Programmers usually decide to modify a program when they see a method returning an unexpected or outdated result, and throwing an unexpected exception. It would be a hassle if programmers have to figure out themselves the input condition that causes unexpected or outdated behaviors in order to write a change contract.

[Previous version]

```
public class Cls {
  private boolean ignoreWS;
  private String str;

  public String getStr() {
    return str;
  }
}
```

[Change contract for `getStr`]

```
requires ignoreWS == true;
ensured !\result.equals(\result.trim());
ensures \result.equals(\result.trim());
```

[New version]

```
public class Cls {
  private boolean ignoreWS;
  private String str;

  public String getStr() {
    if (ignoreWS) {
      return str.trim();
    } else {
      return str;
    }
  }
}
```

Despite the above fact, there are some cases where it is necessary to constrain input domain *directly*. In those situations, you can directly constraint the input using the `requires` keyword. An example is shown in the above. Now class `Cls` has an additional field `ignoreWS` in the both versions. The intention of this field is to ignore whitespace only if `ignoreWS` is true. We can express such an intention in a change contract with "`requires ignoreWS == true;`", resulting in a change contract shown in the above box.

This updated change contract means the following. If there is an input I to `getStr` that satisfies "`ignoreWS == true`" and at the same time ensured in the previous version the condition appearing after "`ensured`", then the same input I should ensure in the new version the condition appearing after "`ensures`".

4 Change contracts for purely structural changes

In the previous examples, we assumed that the program structure does not change; class name remains the same, and the fields and method signatures (i.e., the method name and the list of the method parameters) of the class remain the same over the two versions. Changes are by and large given to the method body to modify the behavior of the method.

While developers commonly modify the behavior of a method, that is not the only kind of changes they make. Another common code change is to modify the structure of a method or a class. For example, programmers often switch a field to a parameter or vice versa. The following is such a case where the `ignoreWS` field of the previous version is replaced with the parameter of the `getStr` method of the new version.

[Previous version]

```
public class Cls {
  private boolean ignoreWS;
  private String str;

  public String getStr() {
    if (ignoreWS) {
      return str.trim();
    } else {
      return str;
    }
  }
}
```

[Change contract for `getStr`]

```
old_field ignoreWS:boolean;
new_param ignoreWS:boolean;
matches ignoreWS == \prev(ignoreWS);
```

[New version]

```
public class Cls {
  private String str;

  public String
  getStr(boolean ignoreWS) {
    if (ignoreWS) {
      return str.trim();
    } else {
      return str;
    }
  }
}
```

When this kind of structural changes are made, programmers need to be sure that the applied structural changes do not change the behavior of the program. In the above example, regardless of whether `ignoreWS` is a field or a parameter, we want the `getStr` method to return the same result if the value of `ignoreWS` is the same.

We earlier mentioned that a change contract assumes that given the same input, the same behavior should be observed between two versions of a method unless otherwise specified. Unlike before, however, we now need to specify when two inputs are deemed the same because the input structure changes across versions. This necessity is fulfilled by the above change contract shown in the middle. It specifies the following three things:

1. the previous version had a field `ignoreWS` of `boolean` type. (the `old_field` part)
2. the new-version `getStr` has a parameter `ignoreWS` of `boolean` type. (the `new_param` part)
3. we deem that the new-version input is the same as the previous-version input only if the `ignoreWS` parameter of the new-version input has the same value as the `ignoreWS` filed of the previous-version input. It is implicitly assumed that the variables shared between the inputs of the two versions such as `str` have the same values across the versions. (the `matches` part)

We used the `\prev` keyword in the above change contract to distinguish between the two `ignoreWS` variables. Keyword `\prev` means "previous". Expression `\prev(ignoreWS)` refers to the `ignoreWS` of the previous version, which is a field. Meanwhile, `ignoreWS` without `\prev` refers to the `ignoreWS` of the new version.

Meanwhile, one may conversely want to switch a parameter to a field. In the above example, this corresponds to the change from the right-hand-side program to the left-hand-side one. This change can be expressed as a change contract similarly. Instead of `old_field` and `new_param`, one can use `old_param` and `new_field`, respectively. The `matches` part can be used without a change although `\prev(ignoreWS)` will this time refer to the `ignoreWS` parameter of the previous version.

5 Change contracts involving structural and behavioral changes

By now we showed how purely behavioral changes and purely structural changes can be expressed as change contracts. Now it is time to consider the cases both kinds of changes take place at the same time. For example, the following is a case to add an additional field `ignoreWS` to the previous version to obtain the new version whose `getStr` method may behave differently depending on the value of `ignoreWS`.

[Previous version]

```
public class Cls {
  private String str;

  public String getStr() {
    return str.trim();
  }
}
```

[Change contract for `getStr`]

```
new_field ignoreWS:boolean;
matches ignoreWS == true;
```

[New version]

```
public class Cls {
  private boolean ignoreWS;
  private String str;

  public String getStr() {
    if (ignoreWS) {
      return str.trim();
    } else {
      return str;
    }
  }
}
```

When such multiplex changes occur, programmers at least need to be sure that the applied changes do not break the existing code. In other words, things that used to work should keep working well. In the case of the above example, the previous version was working fine when ignoring whitespace. Thus, the new version should keep the same way as the previous version when `ignoreWS` is `true`. The change contract shown in the middle describes that requirement. It specifies when two inputs of the previous and the new versions are deemed the same. If in the new version, the `ignoreWS` field is `true` and the rest of variables shared between the two versions have the same values, then we consider it to be the same as the previous-version input.