Contract-based Software Development

Rasmus Guldborg Pedersen

January 2015

Overview

Pre- and post conditions for methods

What is pre- and post conditions for a method? You may give examples and show different levels of formality. How can program assertions be used to give a formal proof of the correctness of a method? How about termination, in case the implementation of the method contains a loop?

Assertions

- Assertion is a predicate.

Assertions

- Assertion is a predicate.
- A property we *think* is true at that place during execution.

Assertions

- Assertion is a predicate.
- A property we *think* is true at that place during execution.
- An assertion is valid if it's always true.

Assertions

- Assertion is a predicate.
- A property we *think* is true at that place during execution.
- An assertion is valid if it's always true.
- Abort if invalid.

Preconditions

- Evaluated before method execution.

Preconditions

- Evaluated before method execution.
- Expectations from the caller.

Preconditions

- Evaluated before method execution.
- Expectations from the caller.
- Part of the specification.

Postconditions

- Evaluated after method execution.

Postconditions

- Evaluated after method execution.
- What the caller can expect.

Postconditions

- Evaluated after method execution.
- What the caller can expect.
- Part of the specification.

Informal Specification

```
public interface ISimpleDictionary {
    /* Put 'key' into the dictionary with associated
        'value'  */
    void Put(string key, object value);

    /* Remove 'key' from the dictionary */
    void Remove(string key);

    /* Does the dictionary contain 'key'? */
    bool ContainsKey(string key);
}
```

Formal Specification

```
public interface ISimpleDictionary {
    /* Pre: key != null && !ContainsKey(key)
        Post: ContainsKey(key) */
    void Put(string key, object value);

    /* Pre: key != null && ContainsKey(key)
        Post: !ContainsKey(key) */
    void Remove(string key);

    /* Pre: key != null */
    [Pure]
    bool ContainsKey(string key);
}
```

Proving correctness

We can formalize specification of code with assertions.
For an assertion to be valid we must prove correctness.
Introducing loop invarients.

Loop Invariant

```
// { Q }
// S_0
// { P }
while(B) {
    // { P ∧ B }
    // S
    // { P }
}
// {P ∧ ¬ B ⇒ R }
```

Loop Invariant: Example

Algorithm for summing integers in a array.
$a[0] + a[1] + \ldots a[N-1] = (\Sigma i | 0 \leq i < N : a[i])$

Loop Invariant: Example

```
// { 0 ≤ N }
int n = 0;
int s = 0;
// { s = (Σ i | 0 ≤ i < n : a[i]) }
while (n != N) {
    // { s = (Σ i | 0 ≤ i < n : a[i]) ∧ n ≠ N }
    s = s + a[n];
    n = n + 1;
    // { s = (Σ i | 0 ≤ i < n : a[i]) }
}
// { s = (Σ i | 0 ≤ i < N : a[i]) ∧ n = N }
```

Loop Invariant: Example proof

Basis: $n = 1$

$a[0] = (\Sigma i | 0 \leq i < 1 : a[i])$

Inductive step: $n + 1$

$a[0] + a[1] + \ldots + a[n-1] + a[n] = (\Sigma i | 0 \leq i < n + 1 : a[i])$

Loop Invariant: Example proof

```
while (n != N) {
    s = s + a[n];
    // { s = (Σ i | 0 ≤ i < n + 1 : a[i]) }
    n = n + 1;
}
```

Loop Invariant: Termination

Function T such that loop execution ends when $T = 0$.
$T = N - n$ for the example.

The End

"Testing shows the presence, not the absence of bugs."
— Edsger W. Dijkstra