Design by Contract

Jacek Wasilewski

Design by Contract (1)

- Design by contract is a software development technique to produce high-quality software
- It is achieved by guaranteeing that every component lives up to its expectations
- It is an extension of Hoare triples
- Each interface should have a specified contract
- Contract specifies what component expects of clients and what clients can expect of it

Design by Contract (2)

- Primary advantage is that contracts and obligations are made explicit instead of implicit
- Central to design by contract is the notion of assertion – expression about the state of a software system
- We evaluate assertions and expect them to evaluate to true
- Design by contract should be enabled only in development or debug
- Types of validations: preconditions, postconditions, invariants

Preconditions (1)

- Preconditions specify conditions that must be true before a method can execute
- Conditions can be imposed upon the arguments passed into the method or upon the state of the called class itself
- Preconditions impose obligations on the client of the method
    - if conditions are not met, it is a bug in the client

Preconditions (2)

- Java Example using iContract

```
/**
* @pre f >= 0.0
*/
public double sqrt(double f) { ... }
```

- Precondition ensures that the argument `f` of function `sqrt()` is greater than or equal to zero
- Clients who use that method are responsible for adhering to that precondition
- If they don't, we are simply not responsible for the consequences

Postconditions (1)

- Postconditions specify conditions that must be true when a method is finished executing
- Conditions can involve
  - the current class state
  - the class state as it was before the method was called
  - the method arguments
  - the method return value
- Postconditions impose obligations on the method
  - If the conditions are not met, it is a bug in the method

Postconditions (2)

- Java Example using iContract
```
/**
* @pre f >= 0.0
* @post Math.abs((return * return) - f) < 0.001
*/
public double sqrt(double f) { ... }
```
- Postcondition ensures that the **`sqrt()`** method calculates the square root of **`f`** within a specific margin of error (+/- 0.001)

Invariants (1)

- Invariants specify conditions that must be true of a class whenever it is accessible to a client
- Specifically when the class has finished loading and whenever its methods are called

Invariants (2)

- Example:

```
/**
* A PositiveInteger is an Integer that is guaranteed to be positive.
*
* @inv intValue() > 0
*/
class PositiveInteger extends Integer { ... }
```

- Invariant guarantees that the `PositiveInteger`'s value is always greater than or equal to zero
- Assertion is checked before and after execution of any method of that class

Design by Contract in Scala (1)

- Different libraries have been proposed for Java
    - cofoja, iContract, c4j, SpringContracts, Oval, …
    - mostly via annotations or annotations in comments
    - some annotations, e.g. in JPA, work like contracts
- In Scala we can implement contracts using
    - assert and assume
    - require
    - ensuring

`require` (1)

- `require` is meant to be used to check at runtime that a method's input values satisfy certain restrictions
- It is best to put at the beginning of the method's body
- It throws `IllegalArgumentException` if conditions is not fulfilled
- Example:

```
val a = 4
require(a > 8) // throws IllegalArgumentException
require(a > 2) // executes silently
```

`require` (2)

- Example:

```
// Given an non-negative integer m and a positive integer n
// div returns the result of the integer division of m by n
def div (m: Int, n: Int): Int = {
    // m must be non-negative
    require(m >= 0)
    // n must be positive
    require(n > 0)
    if (m < n)
        0
    else
        1 + div(m - n, n)
}
```

`assert` (1)

- `assert` can be used anywhere in the code
- Typically is used before a return point in the method with side effect to check that a method's postcondition is satisfied
- It throws `AssertionError` if conditions is not fulfilled
- Example:

```
val a = 4
assert(a > 8) // throws AssertionError
assert(a > 2) // executes silently
```

`assert` (2)

- Example:

```
object Counter {
    var c = 0
    def inc = {
        require(c >= 0) // precondition
        c = c + 1
        assert(c > 0)
    }
    def dec {
        require(c > 0) // precondition
        c = c - 2
        assert(c >= 0) // assertion is invalid
    }
}
Counter.inc
Counter.dec // will throw exception
```

`ensuring` (1)

- `ensuring` can be used anywhere in the code
- Typically is applied to the body of a method that return a value, to check that the returned value satisfies some requirement
- It throws `AssertionError` if conditions is not fulfilled
- Example:

```
val p = (x: Int) => x > 0
1.ensuring(p) // whole expression evaluates to 1 because p(1) evaluates
to true
0.ensuring(p) // throws exception because p(0) evaluates to false
{1; 2} ensuring (x => x == 2) // evaluates to 2
```

`ensuring` (2)

- Example:

```
// div returns non-negative values
def div (m: Int, n: Int): Int = {
    if (m < n)
        0
    else
        1 + div(m - n, n)
} ensuring (res => res >= 0)
// the returned value must be non-negative
```

Design by Contract in Scala (2)

```
// Given an integer value x and an integer list l,
// occurs(x, l) returns true iff x occurs in l.
def occurs(x: Int, l: List[Int]): Boolean = {
    l match {
        case Nil => false
        case h :: t => (h == x) || occurs(x, t)
    }
} ensuring { r =>
    // the result of occurs must be true if x is in l
    r == (l contains x)
}
```

Design by Contract in Scala (3)

```
// For each non-negative integer n, fact(n) returns the factorial of n
def fact(n: Int): Int = {
    require(n >= 0) // n should be non-negative
    var i = n
    var f = 1
    while (i > 0) { f = f * i; i = i - 1 }
    f
} ensuring { res =>
    n match {
        // if n is 0, the returned result should be 1
        case 0 => res == 1
        // if n is positive, the returned value
        // should be the same as (n * fact(n-1))
        case _ => res == n * fact(n - 1)
    }
}
```

Design by Contract in Scala (4)

```
// Given an integer array a and an integer i in a's range,
// min(a,i) returns the position of a mimimum element among those
// stored at or after position position i
def min(a: Array[Int], i: Int) = {
    // i must be a legal index
    require(0 <= && i < a.size)

    var m = i
    for (j <- i to a.length-1)
        if (a(m) > a(j)) m = j
    m
} ensuring { res =>
    // each element of a between positions i and the end of a
    // is at least as big as a(res)
    (i to a.size - 1).forall(j => a(res) <= a(j))
}
```

Design by Contract in Scala (5)

```
// Given an non-null integer array a,
// sort(a) sorts the array in place in non-decreasing order.
def sort(a: Array[Int]) {
    def swap(i: Int, j :Int) = {
        var t = a(i); a(i) = a(j); a(j) = t
    }
    val old_a = a.clone // copy of a

    for (i <- 0 to a.length-1) swap(i, min(a,i))

    // Assertion 1 – ascending order
    assert{(1 to a.size - 1).forall(i => a(i-1) <= a(i))}

    // Assertion 2 – occurrences match
    assert{a.toSet.forall{ e => a.count(_ == e) == old_a.count(_ == e)}}
}
```