Design By Contract

Ian Bayley

Oxford Brookes

Advanced O-O Prog

Outline

Specification vs "Code"

- Code is tedious and time-consuming to read
  - eg millions of lines of C++ for Microsoft Word
- Specification of an operation says what we want to do as a property
  - eg output is a sorted permutation of the input
  - it's also how we know we've done it right (see testing later!)
- Implementation says how exactly we want the operation to be performed
  - eg the complete code for quicksort (? 20-30 lines)
- State is the value of all relevant variables
  - eg fields of an object, parameters, return values etc
- The Hoare approach to specification is to give the necessary properties of the state both before and after the operation as pre/post-conditions

Pre-condition

Definition of pre-condition

A pre-condition is a boolean expression that describes the state that the variables in a program (or program fragment) must be in for the program to work correctly

Example of a pre-condition

The precondition for operation sqrt(x) is `x >= 0`
because negative numbers do not have (real) square roots

- A pre-condition of TRUE means the operation works whenever TRUE is TRUE ie there is no pre-condition
- A pre-condition of FALSE means the operation works whenever FALSE is TRUE ie it never works

Post-condition

Definition of post-condition

A post-condition is a boolean expression that describes the state that the variables in a program (or program fragment) must be in when the program has finished, if the pre-condition was met when the program started

Example of a post-condition

`sqrt(x) * sqrt(x) ≈x`

- The Hoare style deliberately says nothing about what will happen if the pre-condition is not true

How we now specify a program

(* pre *)
program
(* post *)

Meaning of that specification

Executing program when pre is true leaves post true

Everyday Examples of Specifications

1. the owner's manual for a car says that it should operate successfully when the air temperature is -10 °C or more. That is a pre-condition.
   - Below -10 °C, there are no guarantees what might happen
     - the car may fail to work,
     - it may be damaged, or
     - it may work normally
   - in all cases, the car's behaviour satisfies its specification
2. on the back of a Mac, above the power socket is a sticker reading 110V (US). That is a pre-condition.
   - I give it 240V (UK).
   - What happens?
   - Who is responsible?

Specifications of Operations (Pascal Notation)

Specification of Sqrt

```
function Sqrt (x:  real):  real;
(* pre:  x >= 0
   post:  abs (result * result - x) < eps *)
```

- eps (ϵ = "epsilon") is a small tolerance value eg 0.001

Specification of Push and Top operations on Stack

```
procedure Push (var s:  Stack; x:  T)
(* pre:  stack s is not full
   post:  new s it old s with item x pushed on top *)
function Top (s:  Stack):  T;
(* pre:  stack s is not empty
   post:  result is top item of s *)
```

Invariants

Definition of Invariant

An invariant is a property of the states of the variables that must hold at all times (after initialisation)

- it must be maintained by operations
  - ie if it is true before the operation, it must be true after the operation

Example of an Invariant

The stack can have no more than 100 elements

- how do operations Push and Top maintain this invariant?

Invariants and Inheritance

Definition of Liskov Substitution Principle

if S is a subtype of T then objects of type T in a program may be replaced with objects of type S without altering any properties of the program

- this a guideline for good practice which cannot be enforced (e.g. method m always terminates)
- LSP has the following consequences for DBC
  - invariants of T must be preserved in S
  - precondition of S.m cannot be stronger than that of T.m
  - postcondition of S.m cannot be weaker than that of T.m

Outline

Introduction to Design By Contract (DBC)

- Design By Contract is a practical application of the Hoare approach in which pre/post-conditions specify a (business) contract between the supplier of a software component and the component's client
- implemented in the programming language Eiffel by Bertrand Meyer
- Eiffel uses two keywords
  - requires used for pre
  - ensures used for post

Example of a Contract: Window Cleaning

The Contract for Cleaning Windows

"Downstairs windows cleaned for £7"

- Window cleaner (supplier of window cleaning service)
  - Expectation: Gets £7
  - Obligation: Must clean downstairs windows
- House-holder (client of window cleaning service)
  - Expectation: Gets clean downstairs windows
  - Obligation: Must pay £7

Another Example of a Contract: Home-Delivery Service

The Contract for Home Delivery

"If you leave your garage door unlocked we will deliver your package"

- Delivery person (supplier of home-delivery service)
  - Expectation: Finds garage door unlocked
  - Obligation: Must leave package in garage
- House-holder (client of home-delivery service)
  - Expectation: Finds pacakge in garage
  - Obligation: Must leave garage door unlocked
- What if there is no package? Whose fault is it?

DBC as a new Cleaner Style of Programming

- Never check the pre-condition. Assume it is true.
  - it is the caller's obligation to ensure that it is
  - an exception is keyboard input because correct format cannot be guaranteed
- Why? What can we do if pre-condition is false? Guess?
  - Treat `sqrt(-4)` as though it was `sqrt(4)`? `if x<0 then x:= -x;`
  - Return a special value like `-1` or `0`? But what if the special value could be a legitimate value? For example,
    - operation Top on empty stack
    - function `DaysLater(y1,m1,d1,y2,m2,d2: integer):integer;`
- Sometimes testing pre-condition takes longer than operation
  - eg checking the sortedness precondition $O(n)$ of binary search $O(log_2 n)$

Outline

The Concept of a Dictionary

- A dictionary relates a key to a value.
- There is at most one value for a given key.
- It is like a dictionary in Python or a HashMap in Java
- Mathematically, it is a partial function from type KEY to type VALUE
- We need to implement four operations
  - `initialize` to create an empty dictionary
  - `put (k: KEY; v: VALUE)` to add (k,v) to dictionary
  - `value_for (k: KEY): VALUE` to look up `k` in the dictionary
  - `remove (k: KEY)` to remove `k` from the dictionary

The Skeleton of the Dictionary Class

```
class DICTIONARY [KEY, VALUE]
creation
    initalize
feature -- basic queries
    value_for(k:  KEY): VALUE
        -- The value associated with key 'k'
feature -- creation commands
    initialize
        -- Initialize a dictionary to be empty
feature -- other commands
    put (k:  KEY; v:  VALUE)
        -- Insert key 'k' associated with value 'v'
    remove (k:  KEY)
        -- Remove key 'k' from the dictionary
end -- class DICTIONARY
```

Principles of Design By Contract

1. Separate queries from commands (yes, we did)
   - queries deliver a result but do not change visible properties of the object
   - commands might change the object but do not deliver a result
2. Separate basic queries from derived queries,
   (`value_for` is a basic query and we have no derived queries)
   - a derived query is a query that can be defined in terms of basic queries
3. For each derived query, (we have none)
   write a post-condition in terms of the basic queries
4. For each command,
   write a post-condition that specifies the values of every basic query
   (so specify `put`, `initialize`, `remove` in terms of `value_for`)
5. For each query and command, add pre-conditions
6. Write invariants (eg `count >= 0`)

Initialization and Invariant

```
feature -- creation commands
    initialize
        -- initialize a dictionary to be empty
        ensure
            dictionary_is_empty:  count = 0

    invariant
        count_never_negative:  count >= 0
```

Basic Queries

```
feature -- basic queries
    count:  INTEGER
        -- The number of keys in the dictionary

    has(k:  KEY): BOOLEAN
        -- Does the dictionary contain key 'k'?
        require
            key_exists:  k /= Void
        ensure
            consistent:  (count = 0) implies (not Result)

    value_for(k:  KEY): VALUE
        -- The value associated with key 'k'
        require
            key_exists:  k /= Void
            key_in_dictionary:  has (k)
```

Operation Put

```
feature -- other commands
put (k:  KEY; v:  VALUE)
    -- Put key 'k' into dictionary with associated value 'v'

    require
        k_not_in_dictionary:  not has(k)
        k_exists:  k /= Void

    ensure
        count_increased:  count = old count + 1
        key_in_dictionary:  has (k)
        value_for_k_is_v:  value_for(k)=v
```

Operation Remove

```
remove(k:  KEY)
    -- Remove key 'k' from the dictionary
    require
        key_exists:  k /= Void
        key_in_dictionary:  has(k)
    ensure
        count_decreased:  count = old count - 1
        key_not_in_dictionary:  not has(k)
        value_for_k_is_undefined:
            -- pre-condition on value_for is false
```

Outline

Assertions

Examples of assertions (checking pre-conditions)

```
assert (ints != null) :
"Violated pre-condition:  ints must not be null";
```

Syntax for assertions

```
assert boolExpr;
assert boolExpr :  errorString;
assert boolExpr :  anythingPrintable;
```

How to turn on assertions

1. Select Project Properties from File menu
2. In Run Category, add -ea to VM options
3. Check it works with `assert false;` at start of `main` method

Assertions to Check Pre and Post-conditions

```
public static int sum(int...  ints) {
/** @pre.  ints is not null
 * @post.  result is sum of the integers in ints
 */
    assert (ints !=null) :  "ints must be null";
    int total = 0;
    for (int i:  ints)
        total+= i;
    assert (total == Arrays.stream(ints).sum()):
        “sum has not been calculated correctly;
    return total;
}
```

How to "javadoc" pre- and post-conditions

javadoc -tag pre.:m:”Pre:” -tag post.:m:”Post:” Main.java

Output of Example Program

```
The words stuck together make:  Hello, World!
Today is FRIDAY, after WEDNESDAY
The sum of the ints is 24
Exception in thread "main" java.lang.AssertionError:
Crash!!
        at Main.demonstrateAssertsStaticImports(Main.java:46
)
        at Main.main(Main.java:53)
```

Outline

Using the Java Modelling Language (JML)

Example of JML in use

```
public class BankingExample {
    public static final int MAX = 1000;
    private /*@ spec_public @*/ int balance;
    //@ public invariant balance >= 0 && balance <= MAX;
    //@ requires 0 < amount && amount + balance < MAX;
    //@ assignable balance;
    //@ ensures balance == \old(balance) + amount;
    public void credit(final int amount) {
        this.balance += amount;
    }
}
```

JML is a type-checkable language entirely within Java comments for

- run-time assertion checking (asserts invisibly added)
- (compile-time) static checking

Syntax of the Java Modelling Language

- Java sub-language of boolean expressions (variables, literals, operators)
- `requires` for pre, `ensures` for post, `invariant`, `loop_invariant`
- `spec_public` so private/protected variables can be used in specification
- `assignable` to indicate fields that are allowed to change
- `\old(<expression>)` to refer to value of expression at time of entry

- `signals` indicate exceptions raised when postcondition is false
- `\result` used to represent the return value of the method
- `(\forall <decl>; <range-exp>; <body-exp>)` and `\exists`
- further boolean operators ==>, <==, <==>

Using the Spec# Language

Implementation of method sum in C# and Spec#

```
public static int SumValues(int[]!  a) // !  means not null
    ensures result == sum{int i in (0:  a.Length); a[i]};
    {
        int s = 0;
        for (int n = 0; n < a.Length; n++)
        invariant n <= a.Length;
        invariant s == sum{int i in (0:  n); a[i]};
        {
                s += a[n];
        }
        return s;
}
```

Outline

From DBC to Testing

- if the post-condition has been written properly, it is immediately obvious how to test it

Reminder of method credit

```
//@ ensures balance == \old(balance) + amount;
public void credit(final int amount) {
    this.balance += amount;
}
```

- better than "adds amount to the balance"

Manual Test with Test Harness

1. set balance to 200
2. use credit to add 100 to it
3. get new balance
4. note whether or not it is 300

Automated test

```
b.setBalance(200);
b.credit(100);
assert (b.getBalance()
    == 200 + 100);
```

Example Class to Test

```
package junitdemo;
public class Main {
    public static int sum(int...  ints) {
/** @pre.  ints is not null
 * @post.  result is sum of the integers in ints
*/
        assert (ints !=null) :  "ints must be null";
        int total = 0;
        for (int i:  ints)
            total+= i;
        return total;
    }
}
```

Template Tester Created By NetBeans I

```
package junitdemo;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class MainTest {

    public MainTest() {
    }

    @BeforeClass
```

Template Tester Created By NetBeans II

```
public static void setUpClass() throws Exception {
}

@AfterClass
public static void tearDownClass() throws Exception {
}

@Before
public void setUp() {
}

@After
public void tearDown() {
}

@Test
```

Template Tester Created By NetBeans III

```
    public void testSum() {
        System.out.println("sum");
        int[] ints = null;
        int expResult = 0;
        int result = Main.sum(ints);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove
the
        fail("The test case is a prototype.");
    }
}
```

Modifying the Method testSum

Method testSum before

```
System.out.println("sum");
int[] ints = null;
int expResult = 0;
int result = Main.sum(ints);
assertEquals(expResult,
result);
// TODO review and remove fail
fail("The test case is a
prototype.");
```

Method testSum after

```
System.out.println("sum");
int[] ints= {2,3,5,7,11};

assertEquals("Sum is 28",
    28, Main.sum(ints));
assertEquals("Sum is 29",
    29, Main.sum(ints));
```

- in this case, to demonstrate unit testing, since we know that testSum is correct we purposely have a correct test followed by an incorrect test

Output when Modified Tester is Run

```
Testcase:  testSum(junitdemo.MainTest):  FAILED
Sum is 29 expected:<29> but was:<28>
junit.framework.AssertionFailedError:
Sum is 29 expected:<29> but was:<28>
        at junitdemo.MainTest.testSum(MainTest.java)
```

How to JUnit test a class in Netbeans

1. hit CTRL+SHIFT+U in the pane of the class you want to test, selecting JUnit 4.x and accepting the default options
2. add an instance variable for the class to be tested
3. put initialisation code in `setUp` method
   - optional as you can reinitialise with each test
4. modify `assertEquals` call, adding captions
   - you can use `assertTrue` and `assertFalse` too
5. remove the `fail` call
6. right click in code window and select Run