Lecture 2: Specification and Design By Contract

EC3307:Object & Component Technology

Overview

- Specification Vs Code
- Precondition
- Postcondition
- Invariant
- Design By Contract

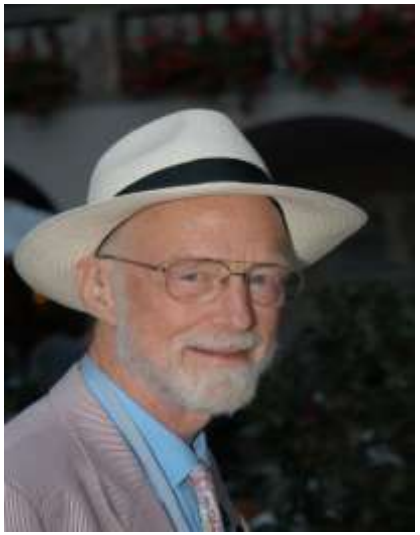
Specification versus ‘code’

- Reading ‘code’ is tedious and time-consuming*
- We should decide what we are going to do before we do it:
  - We should specify *before* we program

- * Microsoft Word™ is 2½ million lines of C++

Professor Sir C. A. R. (Tony) Hoare

- Formerly Head of Programming Research Group (PRG), Oxford University, England
- Microsoft Research, Cambridge, England
- Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12 (1969) 576-580
- [WH66] N. Wirth and C. A. R. Hoare. A contribution to the development of Algol. Comm. ACM, 9(6):413-432, June 1966



Specification

- Say **what** we want to do – not **how** we do it
- Hoare's approach:
  - Determine necessary properties of the state **before** some operation: the operation's *pre-condition*
  - Determine necessary properties of the state **after** the operation: its *post-condition*
- *State*: the values of all the relevant variables, files …

What are Preconditions and Postconditions?

- One way to specify function requirements is with a pair of statements about the function.
- The precondition statement indicates what must be true before the function is called.
- The postcondition statement indicates what will be true when the function finishes its work.

Pre-condition

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

Example: the pre-condition of a program to find the square-root of *x* is:

x >= 0

since negative numbers do not have (*real*) square-roots.

Pre-condition: *TRUE*

- A pre-condition of *TRUE* means there is no pre-condition.
- (It works whenever *TRUE* is true)

Pre-condition: *FALSE*

- A pre-condition of *FALSE* would specify a program that never works.
- (It only works when *FALSE* is true)
- We try to avoid writing programs that have *FALSE* as pre-condition!

Post-condition

A *post-condition* gives a state that should be true when the program has finished.

Example:

sqrtx * sqrtx ≈ x

The result of the square-root operation *sqrtx*, multiplied by itself should be (approximately) equal to *x*

Specifying with pre-+ and post

We abbreviate pre-condition to *pre*

We abbreviate post-condition to *post*

And specify a program as follows:

(pre *)*

program

(post *)*

Meaning: Executing *program* when *pre* is true leaves *post* true

Example

```
void write_sqrt( double x)

//  Precondition:  x  >=  0.
//  Postcondition:  The square root of x has
//  been written to the standard output.
```

- In this example, the precondition requires that

  x >= 0

  be true whenever the function is called.

Pre-condition not satisfied?

- Which of these function calls meet the precondition ?

- write_sqrt(-10);
- write_sqrt(0);
- write_sqrt(5.6);

Pre-condition not satisfied?

- The *Hoare* style of specification intentionally says *nothing* about what will happen if the pre-condition is *not true* – not *satisfied* – when the operation is used.

Everyday example

- The owner's manual for my car says that it should operate correctly when the air temperature is –10°C or more.
- This is a pre-condition of the car's successful use.

Everyday example

- If the temperature falls below –10°C, there is no guarantee what will happen:
  - the car might fail to work
  - it might be damaged
  - it might even work normally.
- In all cases, the car's behaviour will be deemed to have satisfied its specification.

My first, American-model, Macintosh

- On the back, above the power socket:
  - ‘110’

- It expects 110 volts (US).
- I give it 240 volts (UK).
- What happens?
- What is the pre-condition?
- Who is responsible?

Specifying with pre- and post

Example:

(TRUE *)*

Add

($z = x + y$ *)*

Specifying with pre- and post

Pascal:

PROCEDURE **Heading**(parameters);

(pre: what is required*

*post: what will be true afterwards *)*

Note: I prefer to put the procedure's heading *before* the specification. Some people put the specification first.

Example: specification of *Sqrt*

function Sqrt (x : real): real;

(pre: x >= 0*

*post: abs(result * result – x) < eps *)*

- *eps*, ‘epsilon’, ε, a small, tolerance, value.
- Example

const eps = 0.001;

Examples

procedure Push (**var** s: Stack; x: T);

(pre: stack* s *is not full*

post: item x *has been pushed on to top of* s **)*

function Top (s: Stack): T;

(pre: stack* s *is not empty*

post: top item of s *delivered *)*

Invariant

- A property of the states of the variables that holds at all times (after initialisation).
- Maintained by operations:
  - Must be true *before*
  - Must be true *after*
- Example:
  - Only enrolled students may register for classes

Design by contract

- What is meant by "design by contract" or "programming by contract"?

- **contract**: An agreement between classes/objects and their clients about how they will be used.
  - used to assure that objects always have valid state
  - non-software contracts: bank terms, product warning labels
- To ensure every object is valid, show that:
  - constructors create only valid objects
  - all mutators preserve validity of the object
- What is the cost of enforcing a contract?

Example contract issue

- A potential problem situation: queue class with `remove` or `dequeue` method
  - client may try to remove from an empty queue
- What are some options for how to handle this?
  - declare it as an error (an exception)
  - tolerate the error (return null)
  - print an error message inside the method
    - bad because it should leave this up to the caller
  - repair the error in some way (retry, etc.)
    - bad because it should leave this up to the caller

The decision we make here becomes part of the contract of the queue class!

Programming by contract

- What is a precondition? What happens when a precondition is not met?

- **precondition**: Something that must be true before object promises to do its work.
  - Example: A hash map class has a put(*key*, *value*) and a get(*key*) method.
    - A precondition of the get method is that the key was not modified since the time you put it into the hash map.
  - If precondition is violated, object may choose any action it likes
    - If key was modified, the hash map may state that the key/value is not found, even though it is in the map.
  - Document preconditions in Javadoc with `@pre.condition` tag.

Postconditions

- What is a postcondition? Whose fault is it when a postcondition is not met, and what should be done?

- **postcondition**: Something that must be true upon completion of the object's work.
  - Example: At end of `sort(int[])`, the array is in sorted order.
  - Check them with statements at end of methods, if needed.
  - A postcondition being violated is object's (your) own fault.
  - Assert the postcondition, so it crashes if not met.
    - Don't throw an exception -- it's not the client's fault!

  - Document postconditions in Javadoc with `@post.condition`

Class invariants

- What is a class invariant? How is it enforced?

- **class invariant**: A logical condition that always holds for any object of a class.
  - Example: Our account's balance should never be negative.
  - Similar to loop invariants, which are statements that must be true on every iteration of a loop.
  - Can be tested at start or end of every method.
  - Assert your invariants, so it crashes if they are not met
    - don't throw an exception -- it's not the client's fault!
  - Document class invariants in the Javadoc comment header at the top of the class. (no special tag)

Exceptions in the contract

- A precondition is something assumed to be true (and, as far as the client knows, not checked for by the callee)
  - NOT the same thing as throwing an exception on precondition violation
- Example: A `Stack` class has a `pop` method to remove and return the top element. Making the stack throw an exception when the client calls `pop` on an empty stack is **not** a precondition.
  - The caller can see that the callee checks for this and has a predictable action when it fails.
  - We say instead that the `EmptyStackException` is part of the contract.
  - Document exceptions in Javadoc with `@throws` tag.

Which one is it? (1)

- Is each of the following best done as a precondition, postcondition, invariant, an exception in the contract, or none?
  - Our `Queue` class's underlying linked list must never be `null`.
  - Once we add an element to our `SortedLinkedList`, the list should be in sorted order.
  - No one should try to add a `null` element to our `PlayerList`.
  - Our `ArrayList` class's capacity should always be larger than its size.
  - We don't allow computer-only games, so when constructing a new Game object, the array of players they pass should not be composed entirely of computer players.
  - In our `Dictionary` class, they construct the dictionary object by passing in the filename full of words to read. Each line of that file should contain a valid English word.
  - Our `LinkedList` class has a `sort` method that arranges the elements according to their `compareTo` method. To do the sort, every element in the list must be `Comparable` and of the same type. (The `LinkedList` is able to hold non-`Comparable` elements if so desired, just not sort them.)

Which one is it? (2)

- Is each of the following best done as a precondition, postcondition, invariant, an exception in the contract, or none?
  - Our `PlayerList` has a current player index. This index should always be between 0 and the number of players.
  - Our `Game` has a public `play` method, but some plays are not valid at any given time. We also have a `canPlay` method that tells whether the move would be valid. How should play respond when an attempt is made to make an invalid play?
  - Every square on the board should be occupied by a `Player` who is in the game.
  - Our `PlayerList` has a `getHighScoringPlayer` method that examines the list of players and returns the player with the highest score. The `Player` object that we return should never be `null`. Also, no one should call `getHighScoringPlayer` if the `PlayerList` is empty.
  - When a `Player` is constructed, their board letter should not be the same as the `EMPTY` board square letter constant.

Design by Contract

- Practical context for the ‘formal’, Hoare approach
- Pre- and post-conditions are used to specify a (business) contract between the supplier of a software component and the component’s clients.
- Design by Contract by Example, Richard Mitchell and Jim McKim, Addison Wesley, 2002

Motivation

- "Reliability … is particularly important in the object-oriented method because of the special role given by the method to reusability:"
  - "Unless we can obtain reusable software components whose correctness we can trust … reusability is a losing proposition."

- "To be sure that our object-oriented software will perform properly, we need a systematic approach to specifying and implementing … software elements and their relations …"

- "Under the Design by Contract theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations – contracts."

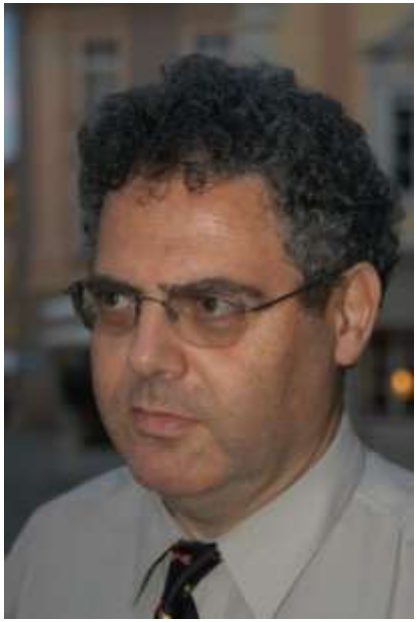
All quotes here and in the following pages from Bertrand Meyer:
www.eiffel.com/doc/manuals/technology/contract

What is Design By Contract?

- “View the relationship between two classes as a formal agreement, expressing each party’s rights and obligations.” ([Meye97a], Introduction to Chapter 11)
- Each party expects benefits(rights) and accepts obligations
- Usually, one party’s benefits are the other party’s obligations
- Contract is declarative: it is described so that both parties can understand what service will be guaranteed without saying how

Bertrand Meyer

- Bertrand Meyer, ISE California, ETH Zürich, Monash University Australia.
- Designer of Eiffel programming language

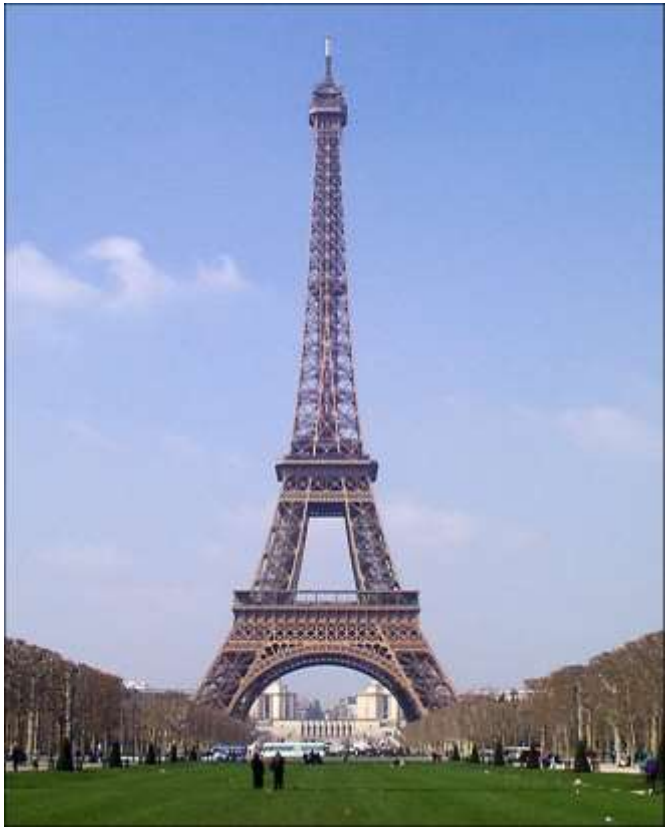


Eiffel: *requires* and *results*

requires for pre

results for post

- Idea is exactly the same



Example: Airlines Reservation

- Customer (Client Class)
- Obligations:
  - Be at KLIA airport at least 1 hour before scheduled time
  - Bring acceptable baggage
  - Pay ticket price
- Rights:
  - Reach Jakarta

Example: Airlines Reservation(cont.)

- Airline (Supplier Class)
- Obligations
  - Bring customer to Jakarta
- Rights
  - No need to carry passenger who is late, has unacceptable baggage or has not paid ticket

Pre and Post conditions + Invariants

- Obligations are expressed via pre and post conditions

"If you promise to call me with the preconditions satisfied, then I, in return promise to deliver a final state in which the postcondition is satisfied"

Precondition: {x>=9} Postcondition: {x>=13}

component: {x:=x+5}

Pre and Post conditions + Invariants

- Invariants

  "For all calls you make to me, I will make sure the invariant remains satisfied."

  - Invariant: {x>=y}
  - Precondition {x>0, y>0}
  - Component: {x:=x+y}

Contract: 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

Contract broken: home-delivery service

‘If you leave your garage door unlocked we will deliver your package’

Supplier: (delivery person)

Expectation:

finds garage door unlocked

Obligation:

Must leave package in (unlocked) garage

On *client*: (house-holder)

Expectation:

finds package in garage

Obligation:

Must leave garage door unlocked

No package? Whose fault?

Redundant Checks

- Redundant checks: naïve way for including in the source code

```
public char pop() {
        if (isEmpty (this))  {
         … // Error-handling
        } else  {
                …..}
```

This is redundant code: It is the responsibility of the client to ensure the precondition

New style (cleaner) style of programming

- You have probably been taught to ‘test pre-conditions’, ‘check user input’, or similar, when writing a procedure.
- With Design by Contract you never *check* the pre-condition, you *assume* it! It is the user’s (caller’s) obligation to *ensure* that the pre-condition is true.
- Only exception: cannot assume end-user will type correct values

Contract Documentation

"A short form, which is stripped of all implementation information but retains the essential usage information: the contract … ."

```
class interface DICTIONARY [ELEMENT] feature
    put (x: ELEMENT; key: STRING) is
        -- Insert x so that it will be retrievable
        -- through key.
    require
        count <= capacity
        not key.empty
    ensure
        has (x)
        item (key) = x
        count = old count + 1

     ... Interface specifications of other features

    Invariant
        0 <= count count <= capacity
end -- class interface DICTIONARY
```

Example: automatic teller machine

- Automatic teller machine (UK ‘cashpoint’)
- Customer types-in PIN and amount wanted.
- Software requires digits (0 .. 9) only.
- How is this (pre-condition) ensured?

- Only digits on keypad! (no letters)

Rationale

- Not supplier's problem!
- Supplier cannot know what to do about it.
- What value should be delivered if we write, for example:

  sqrt(–4)?

- Sometimes it takes longer to test the pre-condition than to do the task:

  example, Binary Search test: O(n), search: O($\log_2 n$)

Liberation!

- Supplier does not have to decide (guess!) what to do when pre-condition not satisfied.
- What do you think of this?

```
function Sqrt (x : real): real;
  (* pre: x >= 0
  post: abs(result * result – x) < eps *)
begin
        if x < 0 then x := –x;
 (* calculate square-root of x *)
```

No more arbitrary, ‘code’ values

- What should be delivered by the top-of-stack function, when the stack is empty? By sqrt when the parameter is negative.
- –1?
- 0?
- Why?

Summary

- Specifying with pre- and post-conditions concentrates on **what** is done, not **how** it is done.
- Simple ‘formal’ method.
- Design by contract makes clear the expectations and obligations on supplier and client of a service.
- Liberates supplier of service from need to ‘guess’ what to do.

What is DBC?

Classes of a system communicate with one another on the basis of precisely defined benefits and obligations.

[Bertrand Meyer, CACM, Vol. 36, No 9, 1992]

What is DBC? (cont.)

- Preconditions of methods

  A boolean expression which is assumed true when the method gets called

- Postconditions of methods

  A boolean expression which the caller can assume to be true when the method returns

- Class invariants

  consistency conditions of objects must hold for all instances

Benefits of Design by Contract

- Better understanding of software construction
- Systematic approach to building bug-free oo systems
- Effective framework for debugging, testing and quality assurance
- Method for documenting software components
- Better control of the inheritance mechanism
- Technique for dealing with abnormal cases, effective exception handling

Rationale

- A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope
- The obligations of the supplier become the benefits to the client

Rationale restated

a contract protects both sides:

- Protects the client by specifying **how much** should be done; the client is entitled to receive a certain result
- Protects the contractor by specifying **how little** is acceptable; the contractor must not be liable for failing to carry out tasks outside of the specified scope