Programming By Contract: Designing for Correctness

James C. McKim, Jr.

Rensselaer, 1999
Overview

What is Programming by Contract?

How do we use Programming by Contract to design correct classes?

Cashier example.

Conclusion.

What it is

Object Oriented Systems are made up of *Objects*.

An Object’s behavior is specified by a *Class*.

Classes consist of *Features*.

A Feature is either a *Command* or a *Query* or both.

A command may change the state of an object, while a query returns information about the state of the object.
What it is (Cont’d)

Programming by contract is a means of rigorously specifying what a feature provides.

In particular you can use it to:

1. Specify when it is appropriate to call a feature (Precondition).

2. Specify what a feature accomplishes when it is called correctly (Postcondition).

Collectively, preconditions and postconditions are called assertions.
What It Is (Cont’d)

If class A uses the services of class B we say A is the client of the supplier B.

The contract between the client and the supplier may be stated as:

The supplier guarantees the postcondition of any feature the client invokes, provided the client guarantees the precondition.
Example

class STACK[ G ]

top : G
require depth > 0

depth : INTEGER

capacity : INTEGER

push( x : G )
require depth < capacity
ensure top = x ; depth = old depth + 1

...
Example

class CLIENT

stk : STACK[STRING]

some_feature is
    -- need to push “Mary” onto the stack but is
    -- unsure if there’s room.
    if stk.depth < stk.capacity then
        stk.push( “Mary” )
    else
        -- alternative action
    end
How to Use It

Principle 1: Separate commands and queries.

Discussion: You must at least provide pure queries to have a chance at runtime checkability.

Justification: Separation of concerns.

Violations: If speed is of the essence (e.g. in a hard real time system) it may be necessary to combine commands and queries in some instances.

Example: Stack
How to Use It (Cont’d)

Principle 2: Do not be limited by your language. For example feel free to use universal and existential quantifiers as need be.

Justification: The goal is a complete specification that is readable by technical people. It’s nice if the specification is compilable, but it’s not necessary.

Example: in class SET[ G ] the following assertion

\[
\text{for_all } x : G (\text{not has( } x ))
\]

states that the set has no elements i.e. it’s empty.
How to Use It (Cont’d)

Principle 3: Identify a basic specification set of pure queries. Every other feature in the class must be completely defined in terms of the queries in the specification set.

Justification: Provides a controlled means of rigorously specifying many features in terms of a few. Once the specification set is understood, clients can quickly comprehend any feature in the class.
How to Use It (Cont’d)

class SET[ G ]

make( c : INTEGER )
cardinality, capacity : INTEGER
insert( x : G )
delete( x : G )
is_empty : BOOLEAN
is_full : BOOLEAN
has( x : G ) : BOOLEAN
How to Use It (Cont’d)

-- specification set: has, cardinality, capacity

cardinality, capacity : INTEGER

has( x : G ) : BOOLEAN

make( c : INTEGER )
require
  c >= 0
ensure
  cardinality = 0 ; capacity = c ;
  for_all x : G (not has(x))
How to Use It (Cont’d)

insert( x : G )
require not has( x ) ; cardinality < capacity
ensure has( x ) ; cardinality = old cardinality + 1
    capacity = old capacity ;
    for_all y : G (y/=x implies has( y ) = old has( y ))

delete( x : G )
require has( x )
ensure not has( x ) ; cardinality = old cardinality - 1
    capacity = old capacity ;
    for_all y : G (y/=x implies has( y ) = old has( y ))
How to Use It (Cont’d)

is_empty : BOOLEAN
ensure Result = (cardinality = 0)

is_full : BOOLEAN
ensure Result = (cardinality = capacity)
How to Use It (Cont’d)

We can add new commands merely by expressing their effects on the specification set.

\[
\text{expand}( \ c : \text{INTEGER} \ )
\begin{align*}
\text{require} & \ c > \text{capacity} \\
\text{ensure} & \ \text{cardinality} = \text{old cardinality} ; \ c = \text{capacity} \\
& \text{for_all} \ x : G \ (\text{has}(x) = \text{old has}(x))
\end{align*}
\]

\[
\text{contract}( \ c : \text{INTEGER} \ )
\begin{align*}
\text{require} & \ c < \text{capacity} ; \ c > \text{cardinality} \\
\text{ensure} & \ \text{cardinality} = \text{old cardinality} ; \ c = \text{capacity} \\
& \text{for_all} \ x : G \ (\text{has}(x) = \text{old has}(x))
\end{align*}
\]
Cashier Example

The CASHIER class is supposed to model a supermarket cashier in a simulation of the supermarket.

Customers always enter the cashier line at the end of the line, but may leave from anywhere. However it is assumed that the customer currently being served will not leave until completion of service.

It is part of CASHIER’s responsibilities to stamp each Customer’s arrival and departure time.
Cashier Example (Cont’d)

-- specification set: customer, length, clock

length : INTEGER
clock : CLOCK

customer( i : INTEGER ) : CUSTOMER
require i >= 1 ; i <= length

make( c : CLOCK )
ensure length = 0 ; clock = c
Cashier Example (Cont’d)

insert( c : CUSTOMER ) -- New customer arrives
ensure
  length = old length + 1 ;  c.start_time = clock.time
  customer( length ) = c ;  clock = old clock
  for_all i, 1..old length
    (customer( i ) = old customer( i ))

remove( i : INTEGER ) -- ith customer leaves.
require i >= 2; i <= length
ensure length = old length - 1 ; clock = old clock
  for_all j, i..length
    (customer( j ) = old customer( j+1 ))
Cashier Example (Cont’d)

finish -- Finish processing current customer.
require length >= 1
ensure length = old length - 1 ; clock = old clock
    old customer( 1 ).finish_time = clock.time
for_all i, 1..length
    (customer( i ) = old customer( i+1 )

current_customer : CUSTOMER
require length >= 1
ensure Result = customer( 1 )

no_customers : BOOLEAN
ensure Result = (length = 0 )
Conclusion

Rigorous specifications provide complete documentation useful both for coding and testing, and so are a good thing.

Programming by contract is a way to provide rigorous specifications in a way that is accessible to a good technical programmer.

Therefore Programming by Contract is a good thing!