12-2009

Object Oriented Program Correctness with OOSimL

José M. Garrido
Kennesaw State University, jgarrido@kennesaw.edu

Recommended Citation

Garrido, José M. "Object oriented program correctness with OOSimL." Journal of Computing Sciences in Colleges 25, no. 2 (2009): 85-91.

OBJECT ORIENTED PROGRAM CORRECTNESS WITH OOSIML[*]

José M. Garrido
Department of Computer Science and Information Systems
Kennesaw State University
Kennesaw, GA 30144
jgarrido@kennesaw.edu

ABSTRACT

Software reliability depends on program correctness and robustness and these are extremely important in developing high-quality software. Correctness is also essential when considering aspects of software security. However, experience applying these concepts, associated methods, and supporting software with Eiffel and Java have shown that students find some diffculty learning program correctness and in learning the software tools provided. We have developed an experimental language, OOSimL, that includes an assertion notation similar to that of Eiffel but which has much more flexibility, and that provides the same semantics as Java.

The first part of this paper provides an overview of concepts and methods on software reliability then briefly describes our experience in teaching these. The second part introduces the Design by Contract (DBC) using the OOSimL programming language, which we recently developed.

INTRODUCTION

Program correctness is a quality factor that indicates whether a program performs according to its specification. Program robustness indicates whether a program can handle abnormal situations that were not covered in the specification in a graceful manner. These two important concepts combined define software *reliability*. These are important for developing reliable and reusable software components and for defining software security aspects.

The Design by Contract (DBC) principle, defined originally by Bertrand Meyer in [1] and [2], is one of the most widely-used approaches for improving development of reliable software. DBC is supported by the Object Constraint Language (OCL) [6], which is a specification language used with the Unified Modeling Language (UML). At the implementation level, DBC is supported by the Eiffel programming language [3] [4].

The *Design by Contract* approach specifies a contract between a supplier class and its client classes. The respective responsibilities (obligations) between the supplier class and the client classes must be stated very clearly and precisely. These mutual obligations are called *contracts*, and assertions are used to check whether an application complies with a contract.

Assertions are Boolean expressions that define correct program states and are placed at specific locations in the code. The assertion must normally evaluate to true. Failure of an assertion (evaluate to false) is typically a symptom of a bug in the software, so it must be reported to the user.

Assertions used in programming are: *preconditions* that have to be satisfied (evaluate to true) on method invocation, *post-conditions* that are evaluated after method execution, and *class invariants* that express conditions for the consistency of data in the class.

The assertions in a class specify a contract between its instances and their clients. According to the contract, a server promises to return results satisfying the post-conditions if a client requests available services and provides input data satisfying the preconditions. Class invariants must be satisfied before and after each service (method invocation).

Including assertions in classes provide a powerful tool for finding errors. Assertion monitoring is a way to check what the software does against what its developer described it should do. This results in a good approach to debugging, testing, in which the search for errors is based on consistency conditions provided by the developers themselves.

EXPERIENCE WITH DBC

We have used Design by Contract in our Object Oriented Software Development course (CS4650, a senior elective course) for several years. For the first few years of teaching the course, we used the Eiffel programming language, which has direct support for assertions and DBC. One of the problems found is that students found the language and its environment difficult to master.

Our next approach in teaching DBC was to direct students to write contracts as comments in C++ programs. The problem observed was that students viewed DBC as an afterthought.

The Java programming language has very limited support for including assertions. However, several software tools [7] have emerged that indirectly allows the Java programmers to use assertions.

In this approach, we directed students to develop code in Java and write contracts using jContractor. This is a Java library that uses methods to include contracts in a class.

The main challenge in the course is that students find the assertion notation fairly complex.

We have started to experiment with a new approach for teaching DBC: use DBC in a lower-level (required) course, Introduction to Data Structures (CS3401). This course directs students to write class specifications in OCL then implement the classes in the OOSimL programming language, which we have developed recently [5]. This programming language is higher level than Java and has built-in support for assertions.

The OOSimL language has appropriate syntax statements to process assertions, and at runtime, it checks the assertions via the exception-handling mechanism. The OOSimL compiler generates Java code. Students use the jGRASP or Eclipse environments to develop their programs.

USING ASSERTIONS WITH OOSIML

There are several types of assertions that are included in a source program, these are discussed in the following subsections.

Preconditions and Postconditions

The **precondition** clause introduces an input condition, or precondition; the **postcondition** clause introduces an output condition, or postcondition. The following OOSimL example introduces the use of precondition and postcondition in a function:

```
description
    This function inserts an element x to a list, the element must be
    retrievable through key. */
function put
parameters x of class Element,
       key of type string
is
precondtion
count <= capacity and key.length() > 0
begin
    ... instructions of insertion algorithm ...
postcondition
    find(key) == x and
count == old_count + 1
endfun put
```

In the precondition, *count* is the current number of elements and capacity is the maximum number. In the post-condition, the Boolean query which tells whether a certain element is present invoking function *find*, which returns the element associated with a certain key. The variable *old_count* refers to the value of *count* on entry to the routine, e.g., in the previous state of the object.

Class Invariant

In OOSimL, class invariants are written using the keyword **invariant** followed by a Boolean expression. Class invariants are written after all attribute declarations. The following example shows how a simple class invariant is written:

```
invariant
      count >= 0 and count <=
      capacity
```

Loop Invariants and Variants

Loop invariants and variants are used to help verify the *correctness* of loops. A loop invariant is a Boolean expression that evaluates to *true* on every iteration of the loop. This Boolean expression normally includes variables used in the loop.

The loop invariant has to be true before each iteration of the loop body, and after each iteration of the loop body. If the invariant is true before entering the loop, it must also be true after exiting the loop.

The initial value of the loop invariant helps determine the proper initial values of variables used in the loop condition and body. In the loop body, some statements make the invariant false, and other statements must then re-establish the invariant so that it is true before the loop condition is evaluated again.

In OOSimL, a loop invariant is written using the keyword **loopinv** followed by a Boolean expression. This Boolean expression is true after loop initialization and maintained on every iteration. This is the general property of a loop invariant.

```
loopinv            // loop invariant
    max_elem >= a[i]
```

Invariants can serve as both aids in recalling the details of an implementation of a particular algorithm and in the construction of an algorithm to meet a specification.

The loop variant is an integer expression that always evaluates to a non-negative integer value and decreases on every iteration. The loop variant helps to guarantee that the loop terminates (in a finite number of iterations). In OOSimL, a loop variant is written using the keyword **loopvariant** followed by an integer expression.

In the following example of a loop variant, an integer expression that decreases in value on every iteration of the loop:

```
loopvariant             // loop invariant
    num_elements - i
```

An Example in OOSimL

The following example in the OOSimL language is the class implementation of a stack; it illustrates the use of assertions and the Design by Contract principle.

```
import all psimjava
description
  This class defines a stack implemented by a simple linked list.
  This version supports Assertions. */
class DStack is
 private
 constants
    define N = 100 of type integer // capacity of stack
 variables
    define count of type integer  // current number of items
    define old_count of type integer // previous state
 object references
    define mList of class DList
 invariant
    count >= 0 and count <= 100
 public
 description
    Initializes the stack. */
 function initializer is
 begin
    set count = 0
    create mList of class DList
 postcondition
    count == 0
 endfun initializer
 //
 description
    Return true if the stack is empty */
 function isEmpty return type boolean is
 variables
    define temp of type boolean
 begin
    set temp = call mList.isEmpty // is LinkList object empty?
    return temp
 endfun isEmpty
 description
    Return true if the stack is full */
 function isFull return type boolean is
 variables
    define temp of type boolean
 begin
    if count == N // is stack full?
    then
    set temp = true
    else
    set temp = false
    endif
    return temp
 endfun isFull
 description
    Push a Link or node to the top of the stack */
 function push
      parameters idata of class Data is
 precondition
    count < N // stack not full
 begin
    set old_count = count         // set previous state
    call mList.insertFirst using idata
    set count = count + 1
 postcondition
    count > 0 && count == old_count + 1
 endfun push
 //
 description
    Return a copy of the data in the Link or node
    at the top of the stack. */
 function top return class Data is
 variables
    define old_count of type integer
 object references
define tdata of class Data
 precondition
    count > 0 // stack not empty
 begin
    set old_count = count
set tdata = call mList.getFirst
 postcondition
    count == old_count && count > 0 // no change in stack
```

```
return tdata    // return reference to Data object
 endfun top
 //
 description
    Remove a Link or node from the top of the stack. */
 function pop is
 precondition
    count > 0 // stack not empty
 begin
    set old_count = count    // previous state
call mList.deleteFirst
    set count = count - 1
 postcondition
    // stack not full and stack has one less node
    count < N && count == old_count - 1
 endfun pop
endclass DStack
```

The complete class definition, which includes the implementation of the class can be downloaded from the following Web page:

    http://science.kennesaw.edu/~jgarrido/psim.html

CONCLUSION

An important property of software is reliability, which depends on correctness and robustness. Design by Contract (DBC) is a relatively easy principle to understand and learn; the challenge is to apply this principle in good object-oriented software design and implementation. DBC is based on the notion of client and supplier contracts that specify all operations in terms of responsibilities and obligations. Contracts in object-oriented software systems are expressed in with assertions that are included in specified location in the class definitions.

After several years teaching DBC and working with students applying various approaches, we have started to experiment teaching DBC by specifying classes with the Object Constraint Language (OCL) and implementing them with the new OOSimL language. With OOSimL, preconditions, postconditions, and class invariants can be used for the specification of classes. Contract violations are signaled as exceptions during program execution. These assertions together with loop invariants and variants can help enhance the understanding, learning, and development of program correctness.

REFERENCES

1. Meyer, Bertrand. *Object Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

2. Meyer, Bertrand. “Applying Design by Contract". *IEEE Computer*, Vol 25, Issue 10, Oct 1992.

3. Meyer, Bertrand. *Eiffel: The Language*. Prentice Hall International, Hemel Hemstead, 1992.

4. Meyer, Bertrand. *Object Oriented Software Construction*. Second Ed. Prentice Hall, Upper Saddle River, NJ, 1997.

5. Garrido, José M. *The OOSimL Simulation Language Reference*. Technical Report, Department of Computer Science and Information Systems, Kennesaw State University, June 2008.

6. Warmer, Jos and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, Upper Saddle River, NJ, 1999.

7. Plösch, Reinhold. "Evaluation of Assertion Support for the Java Programming Language". *Journal of Object oriented Technology*. Vol 1, No. 3, Special issue: TOOLS USA 2002 proceedings, pages 5-17.