Assertions and contracts

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University Montreal, Canada

7 January, 2020

Contracts in human affairs

- In human affairs we form legally binding documents specifying an agreement between parties, where each party is expected to satisfy certain obligations and is entitled to expect certain benefits.
- The legality factor implies that if one of the parties does not hold up to his or her obligations, one (perhaps a third party) should be able to assign blame.

Booking a theater seat: An analogy example

- Assume that you wish to book a seat for a theater play and you contact the theater's booking office in order to purchase a ticket.
- The ticket that you purchase constitutes a legally binding document between the theater booking office ("office") that supplies a service and you ("the client") who are purchasing the service.

Booking a theater seat: Obligations of the client

- There are certain obligations that you have as a client before this service can be provided.
- First, you must make a full payment. You must appear at the theater on time.
- You must also present a valid ticket, together with any other supporting documentation.
- For example, if you are a minor or if you are a student and you are eligible for a reduced price, you must also supply valid identification.
- Furthermore, you may also be required to follow some dress code.

Booking a theater seat: Obligations of the client /cont.

- Note that it is not the responsibility of the office to do any of these for you.
- For example, if you do not show up on time or if you do not have a valid ticket and identification, you will not be allowed in.

Booking a theater seat: Obligations of the client /cont.

- In formal logic, your obligations can be captured by a logical expression that must hold before you may enter the auditorium, i.e. before the service is provided.
- We refer to this expression as a *precondition*.
- The office benefits from your obligations: First, it collects a payment and second, it does not need to worry about you being on-time.
- Thus, the precondition constitutes the obligation of the client but also the benefit of the supplier.

Booking a theater seat: Obligations of the supplier

- On the other hand, the office has certain obligations of its own: The play to be shown must be the one advertised (the one which you have purchased the ticket for), and the play must start on-time.
- Furthermore, the rooms (auditorium, reception, bar, etc.) of the building must be well maintained (air-conditioned, heated).
- You may see the obligations of the office as your benefits.

Booking a theater seat: Obligations of the supplier

- In formal logic, the office's obligations can be captured by a logical expression that must hold once the play has finished (i.e. by the successful termination of the service).
- We refer to this expression as a *postcondition*. Thus the postcondition constitutes the obligation of the supplier but also the benefit of the client.

Booking a theater seat: Obligations and benefits /cont.

- It is important to note that a party that acts as a supplier for a given service, may also act as a client for another service that is needed to perform its obligations as a supplier.
- For example, the theater would normally act as a client to a number of suppliers in the city, such as vendors, catering services, security services, etc.

Booking a theater seat: General obligations

- There are also certain general obligations that must always be met such that the building must meet safety standards,that you cannot vandalize the rooms or assault the staff or other clients, etc.
- If any of these general obligations are violated, then the agreement between you and the office is also violated.
- In formal logic, these general obligations are captured by a logical expression called the *invariant*.
- The invariant must always hold from the moment you purchase the service until the moment the service has been successfully completed.

Programming with assertions

- Definition: An assertion is a declarative statement that affirms that a certain premise is true.
- In a computer program, an assertion is a predicate that is assumed to be true. If found false, the assumption made by the assertion does not hold.

Programming with assertions /cont.

- Prior to the execution of any modular unit, there is a condition on the expected state of the computation. This may for example be the type of the arguments passed to the module. This condition is referred to as the *precondition* of the module.
- Furthermore, the successful termination of the module will change the state of the computation.
- The expected new state is referred to as the *postcondition* of the module.

Programming with assertions /cont.

- We can express this by the triple

$$\{P\}\ S\ \{Q\}$$

  where S denotes the routine, P denotes the precondition and Q denotes the postcondition.
- P and Q are the module's assertions.
- The triple reads "If the client can ensure the precondition, then upon termination of the module the provider guarantees the postcondition."

Programming with assertions /cont.

- If no precondition is imposed, we write

$$\mathit{true}\ S\ \{Q\}$$

Clients of the routine have the obligation to uphold the precondition. The routine has the obligation to uphold the postcondition upon successful termination.

Example: Factorial

- Function *factorial* : $\mathbb{N}_0 \to \mathbb{N}_1$ is defined for non-negative integers by two guarded expressions as follows:

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times factorial(n-1) & \text{if } n > 0 \end{cases}$$

Example: Factorial /cont.

- For a routine S that computes this function, we can define the following triple:

$$\{n \in \mathbb{N} \geq 0\}\ S\ \{S(n) = \Pi_{i=1}^{n}(i)\ \textit{if}\ n > 0,\ \textit{else}\ 1\}$$

where the precondition states that the input parameter must be a non-negative integer and the postcondition states that the return value of the routine must be the product of $n \times n-1 \times \ldots \times 1$ in the case that $n > 0$, and it must be 1 otherwise.

Example: Fibonacci number

- Function $fib : \mathbb{N}_0 \to \mathbb{N}_0$ is defined for non-negative integers by three guarded expressions as follows:

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases}$$

Example: Fibonacci number /cont.

- For a routine S that computes this function, we can define the following triple:

$$\{n \in \mathbb{N} \ \geq \ 0\}$$

$$S$$

$$\{fib(0) = 0, fib(1) = 1, \forall(n \geq 2)\ fib(n) = fib(n-1) + fib(n-2)\}$$

where the precondition states that the input parameter must be a non-negative integer and the postcondition states that the return value of the routine for any Fibonacci number must be the sum of the two previous Fibonacci numbers.

Example: Sum of first n natural numbers

- The sum of the first n natural numbers is given by

$$\sum_{i=1}^{n}(i) = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

or a routine S that computes this function, we can define the following triple:

$$\{n \in \mathbb{N} \geq 1\}\ S\ \{S(n) = \sum_{i=1}^{n}(i)\}$$

where the precondition states that the input parameter n must be a positive integer and the postcondition states that the return value of the routine must be the sum of $1 + 2 + \ldots + n$.

Example: The sum of the first n odd natural numbers

- The sum of the first n odd natural numbers is given by

$$\sum_{i=1}^{n}(2i-1) = 1+3+5+\ldots+(2n-1) = n^2$$

For a routine S that computes this function, we can define the following triple:

$$\{n \in \mathbb{N} \geq 1\}\ S\ \{S(n) = n^2\}$$

where the precondition states that the input parameter n must be a positive integer and the postcondition states that the return value of the routine must be n^2.

Importance of assertions

- What is the importance of assertions? After all, we say that they are assumptions about the state of the system. What happens if they end up being false?
- There are two benefits:

1. They explicitly state and distinguish between the obligations and benefits of each party, i.e. what each party must provide and what each party should expect to receive in return.
2. They direct us in assigning blame.

Importance of assertions /cont.

- In the factorial example above, who is to blame if the argument is a negative number or a non-integer?
- To answer the question we need to see whose responsibility is to uphold the precondition.
- The client of the function is clearly to blame in this case.
- Who is to blame if the return value is not the expected value?
- The routine itself is to blame in this case, and we can safely assume that the routine has some error.

Obligations and benefits

- We can generalize and tabulate this discussion illustrating how clients and suppliers relate to preconditions and postconditions of a given service.

	Obligations	Benefits
Client	Precondition	Postcondition
Supplier	Postcondition	Precondition

Partial vs total correctness

- In the triple $\{P\}\ S\ \{Q\}$ we assume that S will successfully terminate.
- Thus, the triple reads *"Whenever P is true before the execution of S, then Q will be true afterwards, otherwise S does not terminate."*
- This is referred to as *partial correctness* as it merely asserts that if an answer is returned then it will be correct.
- Partial correctness is contrasted with *total correctness* which additionally asserts that a statement will terminate.
- Total correctness falls outside of the scope of this discussion.

Defensive programming

- We use the term *defensive programming* to refer to the inclusion of code inside a routine that checks whether the specification of a routine is met, such as correct argument values.

Example: Defensive programming

- The code below demonstrates the implementation of the factorial function using defensive programming to specify the precondition.

```
public static int factorial (int n) {
 if (n < 0)
   return -1;
 else {
   if (n == 0)
     return 1;
   else
     return (n * factorial (n - 1)); }}
```

Example: Defensive programming /cont.

- We can call the method with different arguments:

```
public static void main(String[] args) {
  System.out.println(factorial(-3));
  System.out.println(factorial(5));
}
```

and we get the following output:

```
-1
120
```

Defensive programming: discussion

- Defensive programming has certain consequences:
- The responsibilities between caller and callee are not separated but they are placed in the method body.
- The contract code is tangled with the code of the core operation of the method.

Contract programming

- In programming we adopt the metaphor of contracts for collaborating software elements.
- The agreement involved is an operation to be performed by a software element such as a method.
- The method acts as a *supplier* to any *client* wishing to make use of its services.
- The method of *design by contract (dbc)* (or *contract programming*) recommends that every software element should be associated with a contract as a specification of its interaction with the rest of the world.
- Note that nothing prevents a software element from acting as both a supplier and a client with different elements.

Contract programming /cont.

- Even though contract programming is not confined to object-oriented programming (OOP), we will use OOP as our underlying programming paradigm.
- In OOP we can build contracts by associating each method (which forms part of the interface of the class) with a set of preconditions and a set of postconditions, and by associating an invariant with the class.

Defining assertions on methods

- A contract for a software element is specified by *assertions*: *preconditions* and *postconditions*.
- Precondition: Logical expression describing the state of the system before the execution of an operation.
- Postcondition: Logical expression describing the state of the system after the execution and successful termination of an operation.

Defining assertions on classes

- Since a contract specifies the relation between an element with the outside world, we should then specify assertions for all public (and protected) methods, i.e. those which form part of the interface of the class.

Defining a class invariant

- In OOP we can associate each class with an invariant.
- The class invariant specifies the valid states for every instance of the class.
- An instance of a class which does not uphold the invariant is not a valid instance.