49. Correctness

This is the first chapter in the lecture about contracts and assertions. We all want to write correct programs. But what is correctness? Program correctness is always relative to something else. In this lecture we will discuss program correctness relative to a *program specification*. In Chapter 50, (the next chapter) we will take a closer look at a particular approach to program specification, on which the rest of this lecture will be based.

49.1. Software Qualities

Lecture 13 - slide 2

Program correctness is one of several *program qualities*. A software quality is a positive property of program. There are many different software qualities that may be considered and promoted. In Table 49.1 we list a number of important program qualities.

Quality	**Description**	**Contrast**
Correct	Satisfies expectations, intentions, or requirements	Erroneous
Robust	Can resist unexpected events	Fragile
Reusable	Can be used in several contexts	Application specific
Simple	Avoids complicated solutions	Complex
Testable	Constructed to ease revelation of errors	-
Understandable	Mental manageability	Cryptic

Table 49.1 *Different program qualities listed by name, description, and (for selected qualities) a contrasting, opposite quality*

Of all software qualities, correctness play a particular important role. Program correctness is in a league of its own. Who would care about robustness, reusability, and simplicity of an incorrect program?

49.2. Correctness

Lecture 13 - slide 3

Software correctness is only rarely an absolute concept. Correctness should be seen relative to something else. We will distinguish between program correctness relative to

- The programmers own, immediate comprehension
  - Not formulated - not documented - volatile - easily forgotten
  - Sometimes incomplete
- A program specification
  - Formulated - written
  - Well-considered and agreed upon
  - Formal or informal
  - Part of the program

At the time the program is written, it may be tempting to rely on the comprehension and specification in the mind of the programmer. It is not difficult to understand, however, that such a specification is volatile. The specification may slide away from the original understanding, or it may totally fade away. In a software house it may also easily be the case that the programmer is replaced. Of these reasons it is attractive to base correctness on written and formal specifications.

In the following section we will discuss written and formal specifications that are based on mathematical grounds.

49.3. Specifications

Lecture 13 - slide 4

We will introduce the following straightforward definition of a specification:

> A *program specification* is a definition of what a computer program is expected to do [Wikipedia].
>
> *What - not how.*

Notice that specifications answer *what questions*, not *how questions*.

In the area of formal mathematically-oriented specifications, the following two variants are well-known:

- **Algebraic specifications**
  - Equations that define how certain operations work on designated constructors
- **Axiomatic specifications**
  - Logical expressions - assertions - associated with classes and operations
  - Often divided into invariants, preconditions, and postconditions

We will first study an algebraic specification of a stack, see Program 49.1. We have already encountered this specification earlier in the material, namely in the context of our discussion of abstract data types in Section 1.5. From line 4-11 we declare the syntax of the operations that work on stacks. The operations are categorized as constructors, destructors, and selectors. As the name suggests, constructors are operations that constructs a stack. Both `push` and `pop` are *functions* that return a stack. This is different from the imperative stack *procedures* we experienced in Program 30.1, which mutate the stack without returning any value.

An arbitrary stack can be constructed in terms of one or more constructors. Destructors are operations that work on stacks. (The term "destructor" may be slightly misleading). Any stack can be constructed without use of destructors. As an example, the expression `pop(push(5, pop (push (6, push (7, new ()))))))` is equivalent with `push(7, new ())`. The selectors extract information about the stack.

```
1  Type stack [int]
2    declare
3          constructors
4                  new () -> stack;
5                  push (int, stack) -> stack;
6          destructors
7                  pop (stack) -> stack;
8          selectors
9                  top (stack) -> int;
10                 isnew (stack) -> bool;
11   for all
12         i in int;
13         s in stack;
14   let
15         pop (new()) = error;
16         pop (push (i,s)) = s;
17         top (new()) = error;
18         top (push (i,s)) = i;
19         isnew (new()) = true;
20         isnew (push(i,s)) = false;
21   end
22 end stack.
```

Program 49.1 *An algebraic specification of a stack.*

The lines 12-21 define the *meaning* (also known as the *semantics*) of the stack. It tells us what the concept of a stack is all about. The idea is to define equations that express how each destructor and each selector work on expressions formulated in terms of constructors. The equation in line 16 specifies that it is an error to pop the empty stack. The equation in line 17 specifies that pop applied on stack `s` on which we have just pushed the integer `i` is equivalent with `s`. Please consider the remaining equations and make sure that you understand their meaning relative to your intuition of the stack concept.

The specification in Program 49.1 tells us what a stack is. It is noteworthy that the specification in Program 49.1 defines the stack concept without any binding to a concrete representation of a stack. The specification gives very little input to the programmer about how to implement a stack with use of a list or an array, for instance. A good specification answers *what questions*, not *how questions*.

If you wish to see other similar specifications of abstract datatypes, you may review our specifications of natural numbers and booleans in Program 1.9 and Program 1.10 respectively.

Below, in Program 49.2 we show an axiomatic specification of a single function, namely the square root function. An axiomatic specification is formulated in terms of a precondition and a postcondition. The precondition specifies the prerequisite for activation of the square root function. It states that it is only

possible to calculate the square root of non-negative numbers. The precondition constrains the output of the function. In case of the square root function, the square of the result should be very close to the input.

```
1 sqrt(x: Real) -> Real
2
3   precondition: x >= 0;
4
5   postcondition: abs(result * result - x) <= 0.000001
```

Program 49.2 *An axiomatic specification of the squareroot function.*

In the rest of this lecture we will study object-oriented programming, in which methods can be specified with preconditions and postconditions.

50. Specification with preconditions and postconditions

As exemplified at the end of the previous chapter, preconditions and postconditions can be used to specify the meaning of a function. In this chapter we will study preconditions and postconditions in more details.

50.1. Logical expressions

Lecture 13 - slide 6

Logical expressions and assertions form the basis of preconditions and postconditions. Consequently, we define the concepts of logical expressions and assertions before preconditions and postconditions:

A *logical expression* is an expression of type boolean

An *assertion* is a logical expression, which, if false, indicates an error [Foldoc]

A *precondition* of an operation is an assertion which must be true just before the operation is called

A *postcondition* of an operation is an assertion which must be true just after the operation has been completed

We have worked with logical expressions numerous times during this course. Logical expressions are formed by relational, equational, conjunctional (and) and disjunctional (or) operators. You find these operators at level 3, 4, 8, and 9 in Table 6.1.

Assertions are also used in the context of program testing. In Section 55.7 we surveyed a large collection of assertions, which are available in the NUnit testing tools for C#. As stated in Section 55.8, an assertion in a test case, which returns the value false, causes a failure. A failed testcase signals that the unit under test is incorrect. Assertions used in test cases are similar to assertions found in postconditions.

We can now characterize a precondition in the following way:

- A precondition states if it makes sense to call an operation
- The precondition is a *prerequisite* for the activation

The precondition is typically formulated in terms of the formal parameters of the operation.

Similarly, a postcondition can be characterized as follows:

- A postcondition states if the operation returns the desired result, or has the desired effect, relative to the given parameters that satisfy the precondition
- The postcondition defines the *meaning* of the operation

The postcondition of a procedure or function F must be fulfilled if the precondition of F holds, and if F terminates (F runs to its completion).

50.2. Examples of preconditions and postconditions

Lecture 13 - slide 7

We will now study preconditions and postconditions of the operations in a circular list. A circular list is a linked list, in the sense we discussed in Section 45.14. However, the circular list discussed in this section is only single-linked. The distinctive characteristics of a circular list are the following:

1. The last `LinkedListNode` is linked to the first `LinkedListNode` of the list
2. The `CircularList` object refers to the `LinkedListNode` of the last element instead of the `LinkedListNode` of the first element.

We show a circular list with five elements in Figure 50.1. The idea of referring the last element instead of the first element from the `CircularList` object means that both the front and the rear of the list can be reached in constant time. In many context, this is a very useful property. Notice also, that it is possible to deal with double-linked circular lists as well.

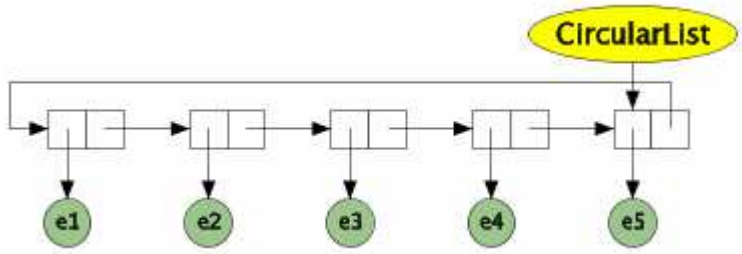

Figure 50.1 *A circular list. The large yellow object represents the circular list as such. The circular green nodes represent the elements of the list. The rectangular nodes are instances of a class akin to* `LinkedListNode`*, which connect the constituents of the list together.*

Below we specify the operations of the circular list with preconditions and postconditions. The specification in Program 50.1 defines the meaning of operations of the yellow object in Figure 50.1. In the program listing, the preconditions are marked with keyword **require**, and shown in **red**. The postconditions are marked with the keyword **ensure**, and shown in **blue**. The names of the keywords stem from the object-oriented programming language Eiffel [Meyer97, Meyer92, Switzer93], which is strong in the area of assertions. Apart from that, the syntax used in Program 50.1 is C# and Java like.

```
1  class CircularList {
2
3    // Construct an empty circular list
4    public CircularList()
5      require true;
6      ensure Empty();
7
8    // Return my number of elements
9    public int Size()
10     require true;
11     ensure size = CountElements() && noChange;
12
```

```
13    // Insert el as a new first element
14    public void InsertFirst(Object el)
15      require !Full();
16      ensure !Empty() && IsCircular() && IsFirst(el);
17
18    // Insert el as a new last element
19    public void InsertLast(Object el)
20      require !Full();
21      ensure !Empty() && IsCircular() && IsLast(el);
22
23    // Delete my first element
24    public void DeleteFirst()
25      require !Empty();
26      ensure
27        Empty() ||
28        (IsCircular() && IsFirst(old RetrieveSecond()));
29
30    // Delete my last element
31    public void DeleteLast()
32      require !Empty();
33      ensure
34        Empty() ||
35        (IsCircular() && isLast(old RetrieveButLast()));
36
37    // Return the first element in the list
38    Object RetrieveFirst()
39      require !Empty();
40      ensure IsFirst(result) && noChange;
41
42    // Return the last element in the list
43    Object RetrieveLast()
44      require !Empty();
45      ensure IsLast(result) && noChange;
46  }
```

Program 50.1 *Circular list with preconditions and postconditions.*

The precondition `true` of the constructor says that there are no particular requirements to call the constructor. This is natural and typical. The postcondition of the constructor expresses that the constructor makes an empty circular list.

The operation `Size` returns an integer corresponding to the counted number of elements in the list. `CountElements` is an operation, which counts the elements in the list. In a particular implementation of `CircularList`, the operation `Size` my return the value of a private instance variable which keeps track of the total number of elements in the list. `nochange` is a special assertion, which ensures that the state of the list has not changed due the execution of the `Size` operation. We see that the postcondition expresses the consistency between the value returned by `Size`, and the counted number of elements in the list.

The operation `InsertFirst` is supposed to insert an element, to become the first element of the list (the one shown at the left hand side of Figure 50.1). The precondition expresses that the list must not be full before the insertion. The postcondition expresses that the list is not empty after the insertion, that it is still circular, and that `el` indeed is the first element of the list. The specification of the operation `InsertLast` is similar to `InsertFirst`.

The operation `DeleteFirst` requires as a precondition a non-empty list. The postcondition of `DeleteFirst` expresses that the list either is empty or circular. If the list is non-empty (and therefore circular) after the deletion, the second element before the deletion must be the first element after the deletion. Notice the

modifier `old`. The value of `Old(expression)` is the value of `expression`, as evaluated in the state just before the current operation is executed. `DeleteLast` is symmetric to `DeleteFirst`.

`RetrieveFirst` returns the first element of the list. The precondition of `RetrieveFirst` says that the list must be non-empty in order for this operation to make sense. The postcondition says that the result is indeed the first element, and that `RetrieveFirst` is a pure function (it does not mutate the state of the circular list). `RetriveLast` is symmetrical to `RetrieveFirst`.

What about the operations `Empty`, `Full`, `CountElements`, `IsCircular`, `IsFirst`, `IsLast`, `RetrieveSecond`, and `RetrieveButLast`? They are intended to be auxiliary, public boolean operations in the circular list. In an implementation of `CircularList` we must implement these operations. They are supposed to be implemented as simple as possible, and they are not supposed to carry preconditions and postconditions. In order to be operational (meaning that the specification can be confirmed at run-time) these auxiliary operations be must be implemented. Nothing comes for free! In reality, we check the consistency between the operations listed in Program 50.1 and the auxiliary boolean operations. An inconsistency reveals an error in either the circular list operations, or in the auxiliary operations. The necessary auxiliary operations are typically much simpler than the circular list operations, and therefore an inconsistency most often will reveal an error in the way we have implemented a circular list operation.

50.3. An Assertion Language

Lecture 13 - slide 8

We are now about to focus on the language in which we formulate the assertions (preconditions and postconditions). In the previous section we have studied examples, in which we have met several features in the assertion language.

As it will appear, we are pragmatic with respect to the assertion language. The reason is that we allow programmed, boolean functions to be used in the assertion language. These boolean function are siblings to the functions that we are about to specify.

It is an important goal that the preconditions and the postconditions should be checkable at program execution time. Thus, the it should be possible and realistic to evaluation the assertions at run-time.

The following items characterize the assertions language:

- Logical expressions - as in the programming language
- Programmed assertions - via boolean functions of the programming language
    - Should be simple functions
    - Problems if there are errors in these
- Universal (*"for all..."*) and existential (*"there exists..."*) quantifiers
    - Requires programming - iteration - traversal
    - It may be expensive to check assertions with quantifiers
- Informal assertions, written in natural language
    - Cannot be checked
    - Much better than nothing
- Special means of expression
    - `old Expr` - The value of the expression at the beginning of the operation
    - `nochange` - A simple way to state that the operation has not changed the state of the object

Use of universal and existential quantifiers, known from mathematical formalisms, makes it hard to check the assertions. Therefore such means of expressions do not exist directly in the assertion language. If we wish to express *for all* ... or *there exists* ... it must be programmed explicitly in boolean functions.

We may easily encounter elements of a specification that we cannot (or will no) check by programmed exceptions. It may be too expensive, or too complicated to program boolean functions which represent these elements. In such situations we may wish to fall back on informal assertions, similar to comments.

50.4. References

[Switzer93] Robert Switzer, *Eiffel and Introduction*. Prentice Hall, 1993.

[Meyer92] Bertrand Meyer, *Eiffel the Language*. Prentice Hall, 1992.

[Meyer97] Bertrand Meyer, *Object-oriented software construction, second edition*. Prentice Hall, 1997.

51. Responsibilities and Contracts

This section is about responsibilities and contracts, and their connection to preconditions and postconditions. Recall from Section 2.2 in the initial lecture that we already touched on responsibilities in the slipstream of the pizza delivery example, see Figure 2.1. At the end of the chapter, in Section 51.8 we briefly discuss Design by Contract, which broadens the scope for applicability of contracts in the development process.

51.1. Division of Responsibilities

Lecture 13 - slide 10

A class encapsulates some description of state, and some operations. A subset of the operations make up the interface between the class and other classes. All together, the class manages a certain amount of *responsibility*. Internally, the class is responsible for keeping the state consistent and sound. Externally, the operations of the class are responsible for delimitation of the messages that they handle, and the quality of the work (results) the operations deliver.

It is bad if a class is irresponsible. Class irresponsibility may occur if a pair classes both expect the other class to be responsible.

It is also bad if a class is too responsible. A pair of over-responsible classes redundantly care about the same properties. This is not necessary, and it bloats the amount of program lines in the implementation of the classes.

This leads us to the essence of this and the following sections, namely division of responsibilities. Let us first enumerate the consequences of well-defined and ill-defined division of responsibilities:

- Without well-defined division of responsibilities
  - All classes accept a large responsibility
  - All program parts check all possible conditions (defensive programming)
  - *Makes a large program even larger*
- With well-defined division of responsibilities
  - Operations can safely operate under given assumptions
  - It is well-defined which parts should check which conditions
  - *Simplifies the program*

51.2. The highly responsible program

Lecture 13 - slide 11

Before we proceed to the role of preconditions and postconditions in relation to responsibility, we will study an example of an object-oriented program with two classes that altogether are over-responsible.

We make our points with yet another version of class `BankAccount`, see Program 51.2, in relation to a client of class `BankAccount`, see Program 51.1. As you will realize below, the illustration of over-responsibility is slightly exaggerated in relation to a real-life program.

The `Main` method in Program 51.1 withdraws and deposits money on the bank account referred by the variable `ba`, which is declared and initialized in line 5. Before withdrawing money in line 8, `Main` checks the soundness of the account (with `AccountOK`), and it checks if there are enough money available. After the withdrawal `Main` checks if the account is still sound. It also deals with the situation where `Main` withdraws an amount of money, which is greater then the balance of the account. Similar observations apply to `Deposit` in line 19.

```
1  public class Client{
2
3    public static void Main(){
4
5      BankAccount ba = new BankAccount("Peter");
6
7      if (ba.AccountOK && ba.EnoughMoney(1000))
8        ba.WithDraw(1000);
9      else
10       WithdrawingProblems("...");
11     if (!ba.AccountOK)
12       MajorProblem("...");
13     if (ba.Balance <= 0)
14       BankAccountOverdrawn(ba);
15
16     ...
17
18     if (ba.AccountOK)
19       ba.Deposit(1500);
20     if (!ba.AccountOK)
21       MajorProblem("...");
22
23   }
24 }
```

Program 51.1 *Excerpt of highly responsible class Client of BankAccount.*

In class `BankAccount` below, the `Withdraw` method in line 9-16 check the soundness of the bank account, and it deals with insufficient funds, before the actual withdrawal takes place in line 15.

The `Deposit` method in line 18-24 cares about the situation where clients deposit very large amounts. In such cases the bank account attempts to check if the money comes from illegal or criminal sources.

```
1  public class BankAccount {
2
3     private double interestRate;
4     private string owner;
5     private double balance;
6
7     // ...
8
9     public void Withdraw (double amount) {
10       if (!AccountOK)
11         ComplainAboutNonValidAccount();
12       else if (!this.EnoughMoney(amount))
13         ComplainAboutMissingMoney();
14       else
15         balance -= amount;
16    }
17
18    public void Deposit (double amount) {
19       if (amount >= 10000000)
20         CheckIfMoneyHaveBeenStolen();
```

```
21        else if (!AccountOK)
22          ComplainAboutNonValidAccount();
23        alse balance += amount;
24     }
25  }
```

Program 51.2 *Excerpt of highly responsible class BankAccount.*

Seen altogether, the amount of code in Program 51.1 and Program 51.2 is much larger than desired. The checks that happen more than once should be eliminated. In addition, some of the responsibilities should be delegated to third party objects.

51.3. Responsibility division by pre and postconditions

Lecture 13 - slide 12

Preconditions and postconditions can be used to divide the responsibility between classes in an object-oriented program. The idea is to make it the responsibility of particular objects to fulfill the precondition of a method, and to make it the responsibility of other objects to fulfill the postcondition of a method. The rules are as follows:

- Fulfillment of the precondition
  - The responsibility of the caller
  - The responsibility of the *client* in an object-oriented program
- Fulfillment of the postcondition
  - The responsibility of the called operation
  - The responsibility of the *server* in an object-oriented program

Client and *server* are roles of objects relative to the message passing in between them. The client and server roles were discussed in Section 2.1. In some books, the server is called a supplier.

Let us recall the precondition and the postcondition of the square root function `sqrt`, as shown in Program 49.2. A function that calls `sqrt` is responsible to pass a non-negative number to the function. If a negative number is passed, the square root function should do nothing at all to deal with it. If, on the other hand, a non-negative number is passed to `sqrt`, it is the responsibility of `sqrt` to deliver a result which fulfills the postcondition. Thus, the caller of `sqrt` should do nothing at all to check or rectify the result.

Now we know who to blame if an assertion fails:

Blame the caller if a precondition of an operation fails
Blame the called operation if the postcondition of an operation fails

51.4. Contracts

Lecture 13 - slide 13

In everyday life, a contract is an enforceable agreement between two (or more) parties. Often, contracts are regulated by law. In relation to programming in general, we define a contract in the following way:

> A *contract* expresses the mutual obligations in between parts of a program that cooperate about the solution of some problem

In object-oriented programming it is natural that the program parts are classes.

The preconditions and the postconditions of the public methods in a class together form a contract between the class and its clients.

It can be a serious matter if a contract is broken. A broken contract is tantamount to an inconsistency between the specification and the program, and it is usually interpreted as an error in the program. The error is usually fatal. A broken contract should raise and throw an exception. Unless the exception is handled, the broken contract will cause the program to stop.

51.5. Everyday Contracts

Lecture 13 - slide 14

Contracts are all around us in our everyday life

When we do serious business in our everyday life, we are very much aware of contracts. When we accept a new job or when we buy a house, the mutual agreement is formulated in a contract.

Below we list some additional everyday contracts:

- Student and University
  - The student enrolls some course
  - The university offers a teacher, a room, supervision and other resources
- Citizen and Tax office
  - The citizen does a tax return
  - The tax office calculates the taxes, and regulates the paid amount of money
- Football player and Football club
  - The player promises to play 50 games per season
  - The football club pays 10.000.000 kroner to the player pr. month
- Citizen and Insurance company
  - The insurance holder pays the insurance and promises to avoid insurance fraud
  - In case of a damage or accident, the insurance company pays compensation

51.6. Contracts: Obligations and Benefits

Lecture 13 - slide 15

Contracts in object-oriented programs, specified by preconditions and postconditions of certain methods, express obligations and benefits.

In Figure 51.1 we personalize the obligations and benefits of a client and server. In the context of Figure 51.1 the server is called a supplier. This terminology, as well as the syntax used in the illustration, come from the object-oriented programming language Eiffel [Meyer97, Meyer92, Switzer93].

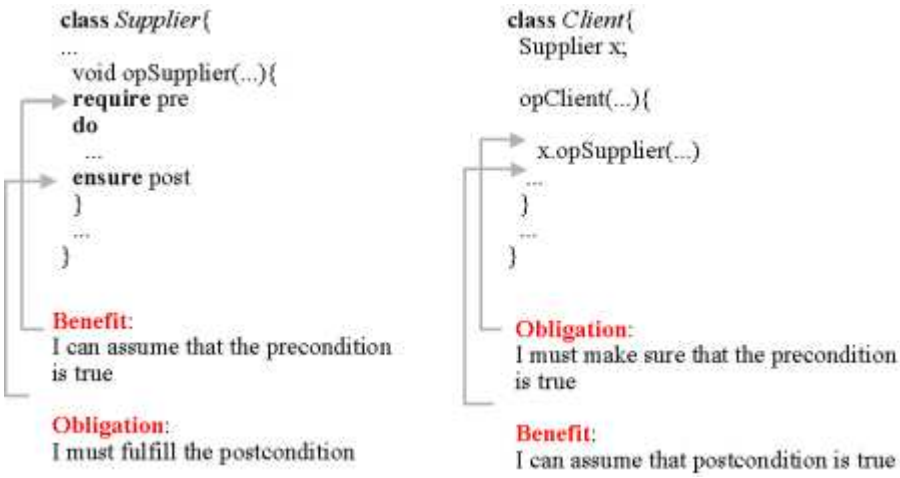

Figure 51.1 *A give-and-take situation involving a client and a server (supplier) class.*

The Client, shown to the right in Figure 51.1 must make an effort to arrange, that everything is prepared for calling `opSupplier` in the class `Supplier`. These efforts can be enjoyed by the supplier, because he can take for granted that required precondition of `opSupplier` is fulfilled.

The roles are shifted with respect to the rest of the game. The supplier must make an effort to ensure that the postcondition of `opSupplier` is fulfilled when the operation terminates. This reflects the fact the operation has done the job, as agreed on in the contract. In return, the client can take for granted that the opposite party (the supplier) delivers an appropriate and correct result.

The obligations and benefits of the contract can be summarized as follows:

Obligation - May involve hard work
Benefit - A delight. No work involved

If you feel that the discussion in this section is too abstract, we will rephrase the essence in the next section relative to the squareroot function.

51.7. Obligations and Benefits in Sqrt

Lecture 13 - slide 16

In Program 49.2 of Section 49.3 we exemplified axiomatic specifications with a squareroot function. Let us, of convenience, rephrase the specification here.

```
1 sqrt(x: Real) -> Real
2
3   precondition: x >= 0;
4
5   postcondition: abs(result * result - x) <= 0.000001
```

Program 51.3 *An axiomatic specification of the squareroot function.*

The obligations and benefits of `sqrt`, relative to its callers, are summarized in the following table:

-	**Obligation**	**Benefit**
Client	Must pass a non-negative number	Receives the squareroot of the input
Server	Returns a number r for which r * r = x	Take for granted that x is non-negative

Table 51.1 *A tabular presentation of the obligations and benefits of the squareroot function (in a server role) and its callers (in a client role).*

Notice in particular the obligation of the client and the benefit of the server, as emphasized using the **red** color in the table.

51.8. Design by Contract

Lecture 13 - slide 27

As presented in Section 51.4, a contract of a class is the sum of the assertions in the class. Thus, a contract is formed by concrete artifacts in the source program.

As part of the Eiffel efforts [Meyer97, Meyer92, Switzer93], the use and benefits of contracts have been broadened such that contracts affects both design, implementation, and testing. The broad application of contract is known as *Design by Contract* (DBC). Design by Contract is a trademark of the company Eiffel Software, and as such it may be problematic to use the term, at least in commercial contexts.

> Design by Contract™ (DBC) represents the idea of designing and specifying programs by means of assertions

The following summarizes the use of contracts in the different phases of the software development process, and beyond.

- **Design:** A pragmatic approach to program specification
- **Documentation:** Adds very important information to interface documentation of the classes
- **Implementation:** Guides and constrains the actual programming
- **Verification:** The program can be checked against the specification every time it is executed
- **Test:**
    - Preconditions limit the testing work
    - The check of postconditions and class invariants do part of the testing work
- **End use:** Trigger exception handling if assertions are violated

The use of contracts for design purposes is central. The contract of a planned class serves as the *specification* of the class. We have discussed program specifications in Section 49.3 of this material.

Interface documentation - as pioneered by JavaDoc - includes signatures of methods and informal explanations found in so-called documentation comments. It is very useful to include both preconditions, postconditions, and class invariants in such documentation.

During program execution - both in the testing phase and in the end use phase - the actual state of the program execution can be compared with the assertions. As such, it is possible to verify the implementation against the specification at program run-time. If an inconsistency is discovered during testing, we have located an error. This is always a pleasure and a success. If an inconsistency is discovered during end use, an exception is thrown. This is clearly less successful. Exceptions have been treated in Chapter 33 - Chapter 36 of this material.

51.9. References

[Switzer93] Robert Switzer, *Eiffel and Introduction*. Prentice Hall, 1993.

[Meyer92] Bertrand Meyer, *Eiffel the Language*. Prentice Hall, 1992.

[Meyer97] Bertrand Meyer, *Object-oriented software construction, second edition*. Prentice Hall, 1997.

52. Class Invariants

In this chapter we will study yet another kind of assertions called class invariants. The class invariant serves as a strengthening of both the preconditions and the postconditions of all operations in the class. As we will see in the first section of this chapter, a good class invariant makes it easier to formulate both preconditions and postconditions of the operations in the class.

52.1. General aspects of contracts

Lecture 13 - slide 18

When we do computations in general, the values of the variables in the running programs are modified throughout the computation. In an object-oriented program the states of the involved objects will vary as the program execution progresses. This variation of the program state is not arbitrary, however. There is usually some rules that control and constrain the variations. Such rules can be formulated as *invariants*. An invariant describes some properties and relationships that remain constant (do not vary) during the execution of a program.

A *class invariant* is an assertion that captures the properties and relationships, which remain stable throughout the life-time of instances of the class.

> A *class invariant* expresses properties of an object which are stable in between operations initiated via the public client interface

The following characterizes a class invariant:

- acts as a general strengthening of both the precondition and postcondition
- expresses a "*health criterion*" of the object
- must be fulfilled by the constructor
- must be maintained by the public operations
- must not necessarily be maintained by private and protected operations

The class invariant is an assertion, which should be true at *every stable point in time* during the life of an object. In this context, a stable point in time is just after the completion of the constructor and in between executions of public operations on the class. At a stable point in time, the object is in rest - the object is not in the middle of being updated. The unstable points in time are, for instance, in the middle of the execution of a constructor, or in the middle of the execution of a public operation. In addition, a non-public operation may leave the object in a state, which does not satisfy the class invariant. The reason is that a public operation may need to activate several non-public operations, and it may need to carry out additional state changes (assignments) in order to reach a stable state that satisfies the class invariant. A non-public operation may be responsible for only a fraction of the updating of an object.

You can think of the class invariant as a health criterion, which must be fulfilled by all objects in between operations. As a precondition of every public operation of the class, it can therefore be assumed that the class invariant holds. In addition, it can be assumed as a postcondition of every public operation that the class invariant holds. In this sense, the class invariant serves as a general strengthening of both the precondition and the postcondition of public operations in the class. The *effective precondition* is the formulated

precondition in conjunction with the class invariant. Similarly, the *effective postcondition* is the formulated postcondition in conjunction with the class invariant.

> A class invariant expresses some constraints that must be true at every stable point in time during the life of an object

Our primary interest in this chapter is class invariants. Invariants are, however, also useful and important in other contexts.

52.2. Everyday invariants

Lecture 13 - slide 19

Before we proceed to a programming example, we will draw the attention to useful everyday invariants.

- **Coffee Machine**
  - *In between operations there is always at least one cup of coffee available*
- **Toilet**
  - *In between "transactions" there is always at least 0.75 meter of toilet paper on the roll*
- **Keys and wallet**
  - *In between using keys and/or wallet*
    - *During daytime: Keys and wallet are in the pocket*
    - *During nighttime: Keys are wallet are located on the bedside table or underneath the pillow*

The coffee machine invariant ensures that nobody will go for coffee in vain. If you happen to fill your jug with the last cup of coffee from the coffee pot, your operation on the coffee machine is not completed until you have brewed a new pot of coffee.

The toilet paper invariant should be broadly appreciated. As a consequence of the invariant, the operation of emptying the toilet paper reel is not completed before you have found and mounted an extra, full reel of paper.

The last everyday invariant is - in my experience - often broken by women and children, because they do not always wear practical cloth with pockets suitable for wallets and keys. As a consequence, these important items tend to be forgotten or misplaced, such that they are not available when needed. If the proposed key and wallet invariant is observed, you either use the key or wallet, or you will be confident where to find them.

> Adherence to invariants is the key to order in our daily lives

52.3. An example of a class invariant

Lecture 13 - slide 20

It is now time to study the invariant of the circular list. Recall that we introduced preconditions and postconditions of the circular list in Program 50.1 of Section 50.2.

The class invariant of a circular lists expresses that the list is circular whenever it is non-empty. In Program 52.1 the invariant is formulated at the bottom of the program, in line 43-45. In the same way as the preconditions and postconditions, the class invariant involves subexpressions that are realized by programmed operations (`empty`, `isCircular`, and `size`) of the class.

```
1  class CircularList {
2
3    // Construct an empty circular list
4    public CircularList()
5      require true;
6      ensure empty();
7
8    // Return my number of elements
9    public int size()
10     require true;
11     ensure (size = countElements) && noChange;
12
13   // Insert el as a new first element
14   public void insertFirst(Object el)
15     require !full();
16     ensure !empty() && isFirst(el);
17
18   // Insert el as a new last element
19   public void insertLast(Object el)
20     require !full();
21     ensure !empty() && isLast(el);
22
23   // Delete my first element
24   public void deleteFirst()
25     require !empty();
26     ensure (empty() || isFirst(old retrieveSecond));
27
28   // Delete my last element
29   public void deleteLast()
30     require !empty();
31     ensure (empty() || isLast(old retrieveButLast()));
32
33   // Return the first element in the list
34   Object retrieveFirst()
35     require !empty();
36     ensure isFirst(result) && noChange;
37
38   // Return the last element in the list
39   Object retrieveLast()
40     require !empty();
41     ensure isLast(result) && noChange;
42
43   invariant
44     !empty() implies isCircular()  and
45     empty() implies (size() = 0);
46 }
```

Program 52.1 *Circular list with a class invariant.*

If we compare Program 52.1 with Program 50.1 it is worth noticing that the preconditions and postconditions become simpler and shorter, because they implicitly assumes that the class invariant is true. Thus, relative to (a slightly idealised version of) Program 50.1, the invariant is factored out of all preconditions and postconditions.

53. Inheritance is Subcontracting

In this chapter we will review inheritance - including specialization - in the light of contracts. Specialization was discussed in Chapter 25 and inheritance was discussed in Chapter 27. The concept of contracts was introduced in Chapter 51.

Stated briefly, we understand a subclass as a *subcontractor* of its superclass. Being a subcontractor, it will not be possible to carry out arbitrary redefinitions of operations in a subclass, relative to the overridden operations in the superclass.

53.1. Inheritance and Contracts

Lecture 13 - slide 22

The following question is of central importance to the discussion in this chapter.

How do the assertions in a subclass relate to the similar assertions in the superclass?

Figure 53.1 illustrates a class B which inherits from class A. Both class A and B have invariants. In addition, operations in class A that are redefined in class B have preconditions as well as postconditions.

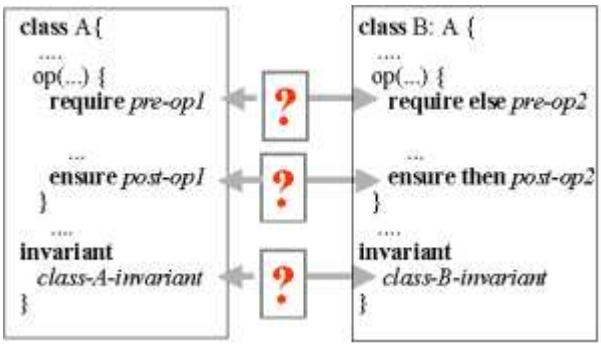

Figure 53.1 *The relationship between inheritance and contracts*

The question from above can now to refined as follows:

- How is the invariant in class B related to the invariant of class A?
- How is the precondition of the operation `op` in class B related to the precondition of the overridden operation `op` from class A?
- How is the postcondition of the operation `op` in class B related to the postcondition of the overridden operation `op` from class A?

Each of the three questions are symbolized with a red question mark in Figure 53.1.

53.2. Subcontracting

Lecture 13 - slide 23

Due to polymorphism, an instance of a subclass can act as a *stand in* for - or *subcontractor* of - an instance of the superclass. Consequently, the contract of the subclass must comply with the contract of the superclass. The contract of a subclass must therefore be a *subcontract* of the superclass' contract. This is closely related to the principle of substitution, which we discussed in Section 25.7.

The notion of subcontracting is realized by enforcing particular requirements to preconditions, postconditions, and class invariants across class hierarchies. In order to understand inheritance as subcontracting, the following rules must apply for assertions in a subclass:

- The precondition must not be stronger than the precondition in the superclass
- The postcondition must not be weaker than the postcondition in the superclass
- The class invariant must not be weaker than the invariant in the superclass

As discussed in Section 50.1, a precondition of an operation states the prerequisites for calling the operation. If the precondition is evaluated to the value *true*, the operation can be called. It is the responsibility of the caller (the client) to fulfill the precondition. The postcondition of the operation states the meaning of the operation, in terms of requirements to the returned value and/or requirements to the effect of the operation. It is the responsibility of the operation itself (the server) to fulfill the postcondition. The postcondition must be *true* if the precondition is satisfied and if the operation terminates normally (without throwing an exception).

If we assume that the precondition of a redefined operation in a subclass is stronger than the precondition of the original operation in the superclass, then the subclass cannot be used as a subcontractor of the superclass. Consequently, the preconditions of redefined operations in subclasses must be equal to or weaker than the preconditions of corresponding operations in superclasses.

In case the postcondition of a redefined operation in a subclass is weaker than the postcondition of the operation in the superclass, the redefined operation does not solve the problem as promised by the contract in the superclass. Therefore, the postconditions of redefined operations must be equal to or stronger than the postconditions of corresponding operations in superclasses.

The superclass has promised to solve some problem via the virtual operations. Redefined and overridden operations in subclasses are obliged to solve the problem under the same, or possible weaker conditions. This causes the weakening of preconditions. The job done by the redefined and overridden operations must be as least as good as promised in the superclass. This causes the strengthening of postconditions.

The invariant of the superclass expresses requirements to instance variables in the superclass, at stable points in time. These instance variables are also present in subclasses, and the requirements to these persist in subclasses. Consequently, class invariants cannot be be weakened in subclasses.

Relative to Figure 53.1 the formulated precondition `pre-op2` in `B.op` serves as a weakening of `pre-op1` of `A.op`. The effective precondition of `B.op` is `pre-op1` **or** `pre-op2`. Similarly, the effective postcondition of `B.op2` is `post-op1` **and** `post-op2`. The use of the Eiffel keywords **require else** and **ensure then** signals this understanding.

Operations in subclasses cannot arbitrarily redefine/override operations in superclasses

In our discussion of redefinition of methods in Section 28.9 we came up with some technical and syntactical requirements to redefinitions. The contributions outlined above in terms of subcontracting constrain the meaning (the semantics) of redefined operations in subclasses in relation to the original operations in superclasses. This is very satisfactory!

53.3. Class invariants in the triangle class hierarchy

Lecture 13 - slide 24

We studied the specialization hierarchy of polygons in Section 25.5. In Figure 53.1 below we revisit the five triangle classes. It is our interest to understand how the class invariants are strengthened in subclasses of the most general triangle class.

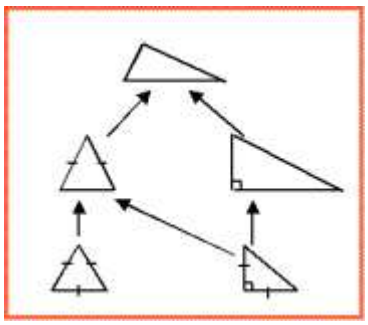

Figure 53.2 *The hierarchy of triangle classes. The root class represents the most general triangle. The son to the left represents an isosceles triangle (where to sides are of same lengths). The son to the right represents a right triangle, where one of the angles is 90 degrees. The triangle at the bottom left is an equilateral trianlge (where all three sides are of equal lengths). The triangle at the bottom right is both an isosceles triangle and a right triangle.*

The invariants of the five types of triangles can be described as follows:

- **Most general triangle:**
  3 angles, 3 edges
  Sum of angles: 180 degrees
- **Isosceles triangle**
  Invariant of general triangle
  2 edges of equal length
- **Equilateral triangle:**
  Invariant of isosceles triangle
  3 edges of equal length
- **Right triangle:**
  Invariant of general triangle
  Pythagoras
- **Isosceles right triangle:**
  Invariant of isosceles triangle
  Invariant of right triangle

Notice that the *italic contributions* above describe the strengthenings relative to the invariant of the superclass.

53.4. Assertions in Abstract classes

Lecture 13 - slide 25

Abstract classes where discussed in Section 30.1. An abstract method in an abstract class defines the name and parameters of the method - and nothing more. The intended meaning of the method is an informal matter. In Chapter 30 we did not encounter any means to define or constrain the actual result or effect of abstract methods. In this section we will see how the meaning of an abstract method can be specified.

In Program 30.1 we studied an abstract class `Stack`. Below, in Program 53.1 we show a version of the abstract stack with contractual elements - preconditions and postconditions. Possible future non-abstract subclasses of `Stack` will be subcontractors. It means that such subclasses will have to fulfill the contract of the abstract stack, in the way we have discussed in Section 53.2.

```
using System;

public abstract class Stack{

  abstract public void Push(Object el);
    require !full;
    ensure  !empty  &&  top() = el  &&  size() = old size() + 1 &&
            "all elements below el are unaffected";

  abstract public void Pop();
    require !empty();
    ensure  !full()  &&  size() = old size() - 1 &&
            "all elements remaining are unaffected";

  abstract public Object Top
    require !empty();
    ensure  nochange  &&  Top = "the most recently pushed element";    {
    get; }


  abstract public bool Full
    require true;
    ensure nochange  &&  Full = (size() = capacity);   {
    get; }


  abstract public bool Empty
    require true;
    ensure nochange  &&  Empty = (size() = 0);   {
    get;}

  abstract public int Size
    require true;
    ensure nochange  &&  Size = "number of elements on stack";   {
    get;}

  public void ToggleTop()
    require size() >= 2;   {
    if (Size >= 2){
      Object topEl1 = Top;  Pop();
      Object topEl2 = Top;  Pop();
      Push(topEl1); Push(topEl2);
    }
    ensure size() = old size() &&
    "top and element below top have been exchanged"  &&
    "all other elements are unaffected";
  }

  public override String ToString(){
    return("Stack");
  }
}
```

Program 53.1 *An abstract class with preconditions and postconditions.*

As we have seen before, **require** clauses are preconditions and **ensure** clauses are postconditions. Notice the use of **old** and **nochange**, which we introduced in Section 50.2. The *"italic strings"* represent informal preconditions. Alternatively, and more rigidly, we may consider to implement these parts of the assertions as private boolean functions. Notice, however, that it would be quite demanding to do so, at least compared with the remaining implementation efforts involved.