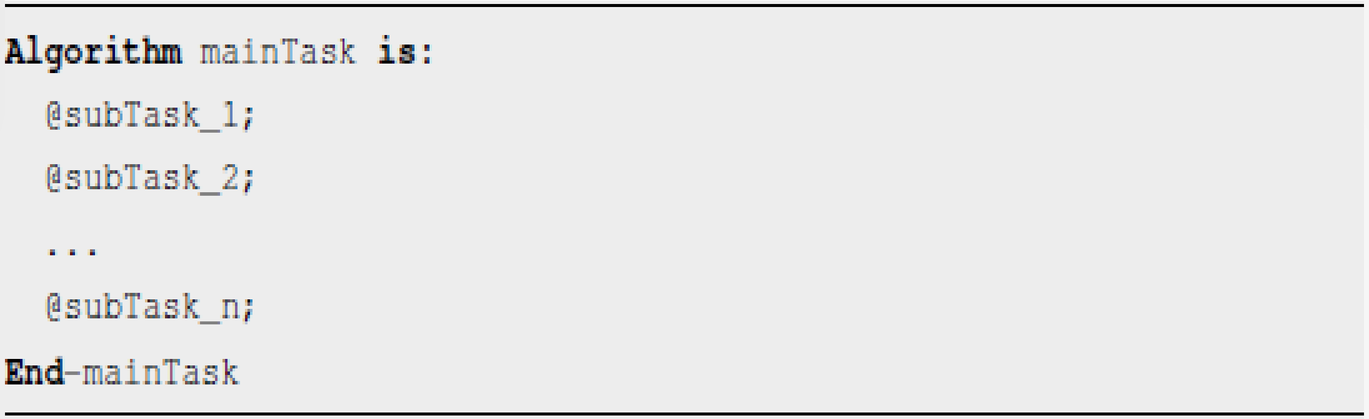
Design by Contract (DbC) [1,2]

- The Design by Contract (DbC) methodology has entered software development due to Bertrand Meyer, along with the Eiffel language
- It proposes a contractual approach to the development of object-oriented software components, based on the use of *assertions*
- The approach has been aimed at increasing the *reliability* of object-oriented software components - a critical requirement in the context of large-scale software reuse, as promoted by the object-oriented paradigm
  - *reliability* = *correctness* (software's ability to behave according to the specification) + *robustness* (the ability to properly handle situations outside of the specification)
- Expected to positively contribute to
  - the develoment of correct and robust OO systems
  - a deeper understanding of inheritance and related concepts (overriding, polymorphism, dynamic bynding), by means of the *subcontracting* concept
  - a systematic approach to *exception handling*

Software Contracts. Assertions

- A generic algorithm to solve a non-trivial task



```
Algorithm mainTask is:
  @subTask_1;
  @subTask_2;
  ...
  @subTask_n;
End-mainTask
```

- Each subtask may be either inlined or triggering the call of a subroutine
- Analogy: calling a subroutine to solve a subtask vs. a real-life situation with a person (client) requiring the services of a third-party (provider) to accomplish a task that he cannot / would not do personally
  - e.g. contracting the services of a fast courier to deliver a package to a particular destination in a foreign city

Software Contracts. Assertions

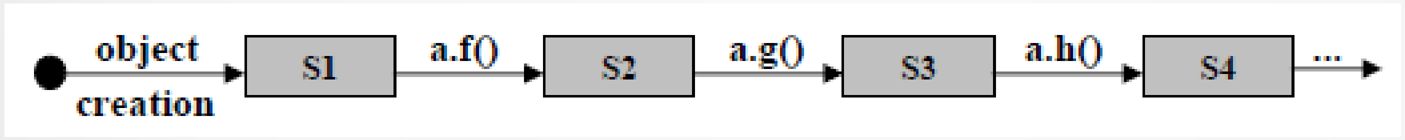
- Characteristics of human contracts involving two parts
  - stipulate the benefits and obligations of each part; a benefit for one part is an obligation for the other
  - may also reference general laws that should be obeyed by both parts
- *Same principles should apply to software development: each time a routine depends on the call of a subroutine to accomplish a subtask, there should be a contractual specification among the client (caller) and supplier (calee)*
- The clauses of software contracts are formalized by means of *assertions*
  - *assertion = expression involving a number of software entities, which state a property that these entities should fulfill at runtime*
  - *closest concept – predicate, implementation – boolean expressions*
  - *some apply to individual routines (pre/post-conditions), other to the class as a whole (invariants)*

Pre/post-condition assertions

- A <pre, post> pair for a method expresses the software contract that the method in question (provider of some service) publishes for its callers (clients of the service)
- Characteristics of software contracts
  - the precondition defines the situation when a call is legitimate - obligation for the client (caller) and benefit for the provider (method)
  - the postcondition states which properties should be fulfilled once the execution has ended - obligation for the provider (method) and benefit for the client (caller)
- The major contribution brought by DBC in the field of software reliability: precondition = benefit of the service provider
- *DBC non-redundancy principle: The body of a mehod should never check for a precondition* (as opposed to defensive programming)
- In Eiffel, assertions are part of the language, allowing for runtime monitoring
  - precondition violation = bug in the client, postcondition violation = bug in the supplier

Invariant assertions

- In addition to pre/post-conditions, that capture the behavior of individual methods, it is possible to express global properties of a class' instances, that should be preserved by all its public methods
- An *invariant* encloses all the semantic constraints and integrity rules applying to the class in question
- Lifecycle of an object



- An assertion *I* is a correct invariant for a class *C* if and only if the following conditions hold
  - each constructor of *C*, apply to arguments that fulfill its precondition, in a state in which the class attributes have default values, leads to a state in which *I is fulfilled*
  - each public method of *C*, applied to a set of arguments and to a state satisfying both *I* and the method precondition, leads to a state satisfying *I*

Correctness of a class

- Informally, a class is said to be correct with respect to its specification if and only if its implementation, as given by the method bodies, is consistent with its preconditions, postconditions and invariant
- **Definition:** The class *C* is said to be correct with respect to its assertions (pre/post-conditions and invariant) if and only if the following conditions hold:

    (1) [*default_C* and *pre_p(x_p)*] *body_p* [*post_p(x_p)* and *INV*]

    for each class constructor *p* and each set of valid arguments of *p* – *x_p* and

    (2) [*pre_r(x_r)* and *INV*] *body_r* [*post_r(x_r)* and *INV*]

    for each public class method r and each set of valid arguments of *r* – *x_r*, where

    default_C is an assertion stating that the attributes of *C* have default values for their type, *INV* is the invariant of *C*, *pre_m*, *post_m*, *body_m* are the precondition, postcondition and body of an arbitrary method *m* of *C*.

The purpose of using assertions

- Support in writting correct software, including the means to formally define correctness
  - The writing of explicit contracts comes as a prerequisite of their enforcement in software
- Support for a better software documentation
  - Essential when it comes to reusable assets, see the case of Ariane!
- Support for testing, debugging and quality assurance
  - Levels of runtime assertion monitoring:
    - 1.preconditions only
    - 2.preconditions and postconditions
    - 3.all assertions
  - While testing, enforce level 3, in production, there is a tradeoff between trust in the code, efficiency level desired and critical nature of the application
- Support for the development of fault tolerant systems

Defensive Programming [3]

- Analogy to *defensive driving*
  - *Defensive driving*: You can never be sure what the others might do, so take responsibility of protecting yourself, such that another driver's mistake won't hurt you!
  - *Defensive programming*: If a routine is passed bad data, it should not be hurt, even if the bad data is someone else's fault (humans, software).
- The core idea of defensive programming is guarding against unexpected errors
- Acknowledges that errors happen and invites programmers to write code accordingly
- Comprises a set of techniques that make errors easier to detect, easier to repair and less damaging in production code
- Should serve as a complement to defect-prevention techniques (iterative design, pseudocode first, design inspections, etc.)
- Protecting from invalid input involves
  - Checking all data received from the outside (from users, files, network, etc.)
    - numeric values should be between tolerances, strings – short enough to handle and obeying to their intended semantics
  - Checking all input parameters
  - Deciding on how to deal with bad data

References

[1] Meyer, B., *Object-Oriented Software Construction (2nd ed.)*, Prentice-Hall, 1997. (Chapter 11 – Design by Contract: building reliable software)

[2] Meyer, B., *Applying „Design by Contract*”, IEEE Computer 25(10):40-51, 1992.

[3] McConnel, S., *Code Complete (2nd ed.)*, Microsoft Press, 2004. (Chapter 8 – Defensive Programming)