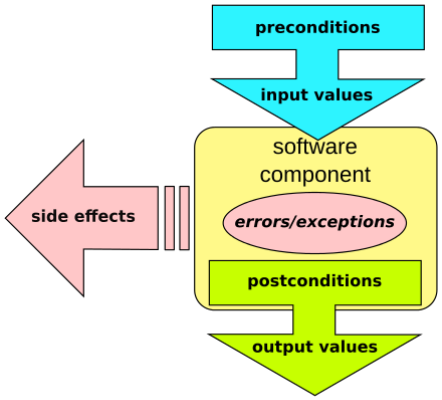
Design by contract

Design by contract (**DbC**), also known as **contract programming**, **programming by contract** and **design-by-contract programming**, is an approach for designing software. It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts.



A design by contract scheme

The DbC approach assumes all client components that invoke an operation on a server component will meet the preconditions specified as required for that operation. Where this assumption is considered too risky (as in multi-channel client-server or distributed computing) the opposite *"defensive design"* approach is taken, meaning that a server component tests (before or while processing a client's request) that all relevant preconditions hold true, and replies with a suitable error message if not.

Contents

History

The term was coined by Bertrand Meyer in connection with his design of the Eiffel programming language and first described in various articles starting in 1986[1][2][3] and the two successive editions (1988, 1997) of his book *Object-Oriented Software Construction*. Eiffel Software applied for trademark registration for *Design by Contract* in December 2003, and it was granted in December 2004.[4][5] The current owner of this trademark is Eiffel Software.[6][7]

Design by contract has its roots in work on formal verification, formal specification and Hoare logic. The original contributions include:

- A clear metaphor to guide the design process
- The application to inheritance, in particular a formalism for redefinition and dynamic binding
- The application to exception handling

- The connection with automatic software documentation

Description

The central idea of DbC is a metaphor on how elements of a software system collaborate with each other on the basis of mutual *obligations* and *benefits*. The metaphor comes from business life, where a "client" and a "supplier" agree on a "contract" that defines, for example, that:

- The supplier must provide a certain product (obligation) and is entitled to expect that the client has paid its fee (benefit).
- The client must pay the fee (obligation) and is entitled to get the product (benefit).
- Both parties must satisfy certain obligations, such as laws and regulations, applying to all contracts.

Similarly, if a routine from a class in object-oriented programming provides a certain functionality, it may:

- Expect a certain condition to be guaranteed on entry by any client module that calls it: the routine's precondition—an obligation for the client, and a benefit for the supplier (the routine itself), as it frees it from having to handle cases outside of the precondition.
- Guarantee a certain property on exit: the routine's postcondition—an obligation for the supplier, and obviously a benefit (the main benefit of calling the routine) for the client.
- Maintain a certain property, assumed on entry and guaranteed on exit: the class invariant.

The contract is semantically equivalent to a Hoare triple which formalises the obligations. This can be summarised by the "three questions" that the designer must repeatedly answer in the contract:

- What does contract expect?
- What does contract guarantee?
- What does contract maintain?

Many programming languages have facilities to make assertions like these. However, DbC considers these contracts to be so crucial to software correctness that they should be part of the design process. In effect, DbC advocates writing the assertions first. Contracts can be written by code comments, enforced by a test suite, or both, even if there is no special language support for contracts.

The notion of a contract extends down to the method/procedure level; the contract for each method will normally contain the following pieces of information:

- Acceptable and unacceptable input values or types, and their meanings
- Return values or types, and their meanings
- Error and exception condition values or types that can occur, and their meanings
- Side effects
- Preconditions
- Postconditions
- Invariants
- (more rarely) Performance guarantees, e.g. for time or space used

Subclasses in an inheritance hierarchy are allowed to weaken preconditions (but not strengthen them) and strengthen postconditions and invariants (but not weaken them). These rules approximate behavioural subtyping.

All class relationships are between client classes and supplier classes. A client class is obliged to make calls to supplier features where the resulting state of the supplier is not violated by the client call. Subsequently, the supplier is obliged to provide a return state and data that does not violate the state requirements of the client. For instance, a supplier data buffer may require that data is present in the buffer when a delete feature is called. Subsequently, the supplier guarantees to the client that when a delete feature finishes its work, the data item will, indeed, be deleted from the buffer. Other design contracts are concepts of "class invariant". The class invariant guarantees (for the local class) that the state of the class will be maintained within specified tolerances at the end of each feature execution.

When using contracts, a supplier should not try to verify that the contract conditions are satisfied; the general idea is that code should "fail hard", with contract verification being the safety net. DbC's "fail hard" property simplifies the debugging of contract behavior, as the intended behaviour of each routine is clearly specified. This distinguishes it markedly from a related practice known as defensive programming, where the supplier is responsible for figuring out what to do when a precondition is broken. More often than not, the supplier throws an exception to inform the client that the precondition has been broken, and in both cases —DbC and defensive programming—the client must figure out how to respond to that. DbC makes the supplier's job easier.

Design by contract also defines criteria for correctness for a software module:

- If the class invariant AND precondition are true before a supplier is called by a client, then the invariant AND the postcondition will be true after the service has been completed.
- When making calls to a supplier, a software module should not violate the supplier's preconditions.

Design by contract can also facilitate code reuse, since the contract for each piece of code is fully documented. The contracts for a module can be regarded as a form of software documentation for the behavior of that module.

Performance implications

Contract conditions should never be violated during execution of a bug-free program. Contracts are therefore typically only checked in debug mode during software development. Later at release, the contract checks are disabled to maximize performance.

In many programming languages, contracts are implemented with assert. Asserts are by default compiled away in release mode in C/C++, and similarly deactivated in C#[8] and Java. This effectively eliminates the run-time costs of contracts in release.

Relationship to software testing

Design by contract does not replace regular testing strategies, such as unit testing, integration testing and system testing. Rather, it complements external testing with internal self-tests that can be activated both for isolated tests and in production code during a test-phase. The advantage of internal self-tests is that they can detect errors before they manifest themselves as invalid results observed by the client. This leads to earlier and more specific error detection.

The use of assertions can be considered to be a form of test oracle, a way of testing the design by contract implementation.

Language support

Languages with native support

Languages that implement most DbC features natively include:

- Ada 2012
- Ciao
- Clojure
- Perl6
- Cobra
- D[9]
- Eiffel
- Fortress
- Lisaac
- Mercury
- Nice
- Oxygene (formerly Chrome and Delphi Prism[10])
- Racket (including higher order contracts, and emphasizing that contract violations must blame the guilty party and must do so with an accurate explanation[11])
- RPS-Obix

- Sather
- SPARK (via static analysis of Ada programs)
- Spec#
- Vala
- VDM

Languages with third-party support

Various libraries, preprocessors and other tools have been developed for existing programming languages without native Design by Contract support:

- Ada, via GNAT pragmas for preconditions and postconditions.
- C and C++, via the *DBC for C* preprocessor, *GNU Nana*, *eCv* and *eCv++* formal verification tools, or the Digital Mars C++ compiler, via CTESK extension of C. Loki Library provides a mechanism named ContractChecker that verifies a class follows design by contract.
- C# (and other .NET languages), via *Code Contracts* (a Microsoft Research project integrated into the .NET Framework 4.0)
- Groovy via GContracts
- Java:
  - Active:
    - OVal (http://oval.sourceforge.net) with AspectJ
    - Contracts for Java (https://github.com/nhatminhle/cofoja) (Cofoja)
    - Java Modeling Language (JML)
    - Bean Validation (only pre- and postconditions)[12]
    - valid4j (http://www.valid4j.org)
  - Inactive/unknown:
    - Jtest (active but DbC seems not to be supported anymore)[13]
    - iContract2/JContracts
    - Contract4J
    - jContractor
    - C4J
    - Google CodePro Analytix
    - SpringContracts for the Spring Framework
    - Jass (http://csd.informatik.uni-oldenburg.de/~jass/)
    - Modern Jass (http://modernjass.sourceforge.net) (successor is Cofoja)[14][15]
    - JavaDbC using AspectJ
    - JavaTESK using extension of Java
    - chex4j using javassist
    - highly customizable java-on-contracts
- JavaScript, via AspectJS (specifically, AJS_Validator), Cerny.js, ecmaDebug, jsContract, dbc-code-contracts (https://www.npmjs.com/package/dbc-code-contracts) or jscategory.
- Common Lisp, via the macro facility or the CLOS metaobject protocol.
- Nemerle, via macros.
- Nim, via macros (https://github.com/Udiknedormin/NimContracts).
- Perl, via the CPAN modules Class::Contract (by Damian Conway) or Carp::Datum (by Raphael Manfredi).
- PHP, via PhpDeal, Praspel or Stuart Herbert's ContractLib.
- Python, using packages like PyContracts, Decontractors, dpcontracts, zope.interface, PyDBC or Contracts for Python. A permanent change to Python to support Design by Contracts was proposed in PEP-316, but deferred.
- Ruby, via Brian McCallister's DesignByContract, Ruby DBC ruby-contract or contracts.ruby.
- Rust via the Hoare (https://crates.io/crates/hoare) library
- Tcl, via the XOTcl object-oriented extension.

See also

- Component-based software engineering
- Correctness (computer science)
- Defensive programming
- Fail-fast
- Formal methods
- Hoare logic
- Modular programming
- Program derivation
- Program refinement
- Test-driven development

Notes

1. Meyer, Bertrand: *Design by Contract*, Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986
2. Meyer, Bertrand: *Design by Contract*, in *Advances in Object-Oriented Software Engineering*, eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991, pp. 1–50
3. Meyer, Bertrand: *Applying "Design by Contract"*, in Computer (IEEE), 25, 10, October 1992, pp. 40–51, also available online (http://se.ethz.ch/~meyer/publications/computer/contract.pdf)
4. "United States Patent and Trademark Office registration for "DESIGN BY CONTRACT"" (http://tess2.uspto.gov/bin/showfield?f=doc&state=4010:lsqmmo.2.2).
5. "United States Patent and Trademark Office registration for the graphic design with words "Design by Contract"" (http://tess2.uspto.gov/bin/showfield?f=doc&state=4010:lsqmmo.2.1).
6. "Trademark Status & Document Retrieval" (http://tarr.uspto.gov/servlet/tarr?regser=serial&entry=78342277). *tarr.uspto.gov*.
7. "Trademark Status & Document Retrieval" (http://tarr.uspto.gov/servlet/tarr?regser=serial&entry=78342308). *tarr.uspto.gov*.
8. "Assertions in Managed Code" (https://msdn.microsoft.com/en-us/library/ttcc4x86.aspx). *msdn.microsoft.com*.
9. Bright, Walter (2014-11-01). "D Programming Language, Contract Programming" (http://dlang.org/contracts.html). Digital Mars. Retrieved 2014-11-10.
10. Hodges, Nick. "Write Cleaner, Higher Quality Code with Class Contracts in Delphi Prism" (http://edn.embarcadero.com/article/39398). Embarcadero Technologies. Retrieved 20 January 2016.
11. Findler, Felleisen Contracts for Higher-Order Functions (http://www.eecs.northwestern.edu/~robby/pubs/papers/ho-contracts-icfp2002.pdf)
12. "Bean Validation specification" (http://beanvalidation.org/1.1/spec/). *beanvalidation.org*.
13. https://www.parasoft.com/wp-content/uploads/pdf/JtestDataSheet.pdf
14. "Archived copy" (https://web.archive.org/web/20160328164727/http://cofoja.googlecode.com/files/cofoja-2011011 2.pdf) (PDF). Archived from the original (https://cofoja.googlecode.com/files/cofoja-20110112.pdf) (PDF) on 2016-03-28. Retrieved 2016-03-25. p. 2
15. "No chance of releasing under Apache/Eclipse/MIT/BSD license? · Issue #5 · nhatminhle/cofoja" (https://github.com/nhatminhle/cofoja/issues/5). *GitHub*.

Bibliography

- Mitchell, Richard, and McKim, Jim: *Design by Contract: by example*, Addison-Wesley, 2002
- A wikibook describing DBC closely to the original model.
- McNeile, Ashley: *A framework for the semantics of behavioral contracts* (https://dx.doi.org/10.1145/1811147.1811150). Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications (BM-FA '10). ACM, New York, NY, USA, 2010. This paper discusses generalized notions of **Contract** and **Substitutability**.

External links

- The Power of Design by Contract(TM) (https://www.eiffel.com/values/design-by-contract/) A top-level description of DbC, with links to additional resources.
- Building bug-free O-O software: An introduction to Design by Contract(TM) (http://archive.eiffel.com/doc/manuals/technology/contract/) Older material on DbC.
- Benefits and drawbacks; implementation in RPS-Obix (http://www.rps-obix.com/docs/manuals/design_by_contract_contract_programming.html)
- Bertrand Meyer, *Applying "Design by Contract"* (http://se.ethz.ch/~meyer/publications/computer/contract.pdf), IEEE Computer, October 1992.
- Using Code Contracts for Safer Code (http://buksbaum.us/2011/04/20/using-code-contracts-for-safer-code/)
