



JOHANNES KEPLER UNIVERSITÄT LINZ
INSTITUT FÜR WIRTSCHAFTSINFORMATIK
CD-LABOR FÜR SOFTWARE ENGINEERING
O. UNIV.-PROF. DR. G. POMBERGER



R. Plösch

Design by Contract for Python

Copyright

Copyright 1997 IEEE. Proceedings of the Joint Asia Pacific Software Engineering Conference (APSEC97/ICSC97), Hongkong, December 2-5, 1997. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions/IEEE Service Center/445 Hoes Lane/P.O. Box 1331/Piscataway, NJ08855- 1331, USA. Telephone: + Intl. 908-562-3966.

TR-SE-97.24

Design by Contract for Python

Reinhold Plösch

Ch. Doppler Laboratory for Software Engineering
Johannes Kepler University of Linz
Altenbergerstr. 69, A-4040 Linz, Austria
ploesch@swe.uni-linz.ac.at

Abstract

The idea of design by contract (DBC), realized in the statically typed object-oriented programming language Eiffel, can be viewed as a systematic approach to specifying and implementing object-oriented software systems. We believe that a statically typed programming language is not suitable in the analysis and design phase of a prototyping-oriented software life cycle. For this purpose, dynamically typed interpreted programming languages are better suited. Unfortunately, dynamically typed programming languages usually do not support the concept of DBC. Therefore we integrated DBC into the programming language Python by using a metaprogramming approach, i.e., without changing the language or the run-time system. We adopted the DBC concept by adding mechanisms for dynamic type checking for method parameters and instance variables. The proposed combination of a more formal approach with a slim programming language provides a good basis for elicitation and documentation tasks in the analysis and design phase, especially in cases of a prototyping-oriented software development approach. Although the approach presented provides basic tool support for the analysis and design phase, further tool support, especially for browsing assertions, is desirable.

1. Motivation

Understanding, using and customizing object-oriented class libraries are major tasks in object-oriented software development. These tasks can be supported by appropriately written documentation. An example of a documentation scheme for object-oriented systems can be found in [11].

In statically typed object-oriented programming languages like C++, Eiffel and Java, designers and programmers are both supported and restricted by the underlying type system; i.e., overriding methods must

meet the type rules imposed by the base classes (static type of parameters and return values).

In Eiffel additional support is available by using assertions for classes and methods. Assertions are elements of formal specifications and express correctness conditions for classes and/or methods. Assertions are part of the implementation and are checked at run time.

The underlying theory, design by contract (DBC), can be viewed as a systematic approach to specifying and implementing object-oriented software systems. Meyer enumerates the primary benefits of this approach [7]:

- A better understanding of object-oriented software construction in general
- A systematic approach to building correct software systems
- An effective framework for quality assurance
- A method for documenting software
- Better control of the inheritance mechanism
- A technique for dealing with abnormal cases, leading to a proper use of exception handling mechanisms

The benefits of DBC are not limited to statically typed languages but can also be valuable for dynamically (weakly) typed programming languages like Smalltalk, Self and Python. Enriching the DBC concept with a type-checking mechanism for method parameters and instance variables (as described in Section 3) leads to a better understanding of the relationships between classes. Additionally, type safer and thus more reliable software systems can be produced.

We implemented the enriched DBC model for the programming language Python [5], [9], [10] to illustrate the feasibility of the approach. Conceptually this work enhances the programming language Python in terms of correctness support. The main reason for implementing DBC support for Python was the need for a system that provides support in the analysis and design phase:

- Modeling domain knowledge in an object-oriented fashion
- Partial specification of semantics of the domain model by means of formal techniques
- Experimentation (i.e., prototyping) with the software architecture of the evolving domain model

We believe that the proposed combination of an enriched DBC model and the programming language Python provides a good basis for handling these tasks.

The ideas presented in this paper are applicable for any object-oriented, dynamically typed programming language. The effort necessary to implement these ideas depends on the specific language mechanisms available.

2. Underlying Theory

In this section we give an overview of the underlying theory as documented in [6], [7]. We describe DBC only as far as it is relevant for understanding our approach for dynamically typed programming languages. The original literature quoted above may serve as basis for clarifying certain aspects of the theory. As described in [6], assertions may appear

- as part of a method (preconditions and postconditions)
- as class invariants
- as check instructions
- as loop invariants

Any of the assertions above consists of one or more assertion clauses, each based on a boolean expression. The value of an assertion is true if and only if every assertion clause in the assertion yields the value true. An assertion violation at run time triggers an exception (see [6]).

Preconditions/Postconditions. The specification of a method is made up of a pair of assertions, i.e., preconditions and postconditions.

```

put (v: T) is
  -- add item v to queue
  require
    not full
  do
    ...
  ensure
    count = old count + 1;
    (old empty) implies (item() = v)
    not empty
    item((last - 1 + capacity) \ capacity) = v
  end -- put

```

Figure 1. Preconditions and postconditions

Figure 1 shows a method *put* that adds an item *v* of type *T* to a *queue*. Adding an item requires that the queue is not yet full.

For postconditions, *old*-expressions are available. The value of the expression on method exit is the value of the expression as evaluated on method entry. In Figure 1 *count = old count + 1* means that after addition of an item the number of elements in the queue is increased by 1.

Assertion clauses may also comprise expressions with boolean implications using the keyword *implies*. In Figure 1 *(old empty) implies (item() = v)* indicates that if the queue was initially empty, the value at cursor position will be the one just inserted.

Method *put(v: T)* can be replaced by any method that satisfies the obligation defined by the original *contract* (by means of preconditions and postconditions). In object-oriented programming, method substitution is usually achieved by subclassing and overriding methods. In Eiffel, preconditions and postconditions must be defined in subclasses using *require else* and *ensure then* specification, expressing that the new assertions are variations of the base-class assertions. Variation means that the preconditions are *or*-ed with the original preconditions and that the postconditions are *and*-ed with the original postconditions. This rule (also known as subcontracting) is valid not only for direct ancestors of a class but also for the entire inheritance path. This has the following implications:

- Weakening the preconditions or retaining the preconditions of the inherited method.
- Strengthening the postconditions or at least retaining the postconditions of the inherited method. Strengthening is very common, as overriding methods usually do more (e.g., changing the value of additional instance variables).

Class invariants. Class invariants specify properties any object of the class must satisfy at every instant. The invariant of a class is obtained by concatenating the classes own invariants with the invariants of all parent classes. This means that a class inherits not only interfaces and implementation but also their invariants of base classes.

```

deferred class Person feature
  ...
  invariant
    age >= 0;
    age < 100;
    len(name) > 0;
    socialSecurityNumber > 1000;
  end

```

Figure 2. Class invariants

Figure 2 shows the invariants of a class *Person*, implying a number of consistency conditions on every object of class *Person*.

Class invariants are checked at method invocation jointly with the preconditions and on leaving a method jointly with the postconditions. A class is said to be *consistent* when these checks yield true for every method of a class. Class consistency is an import aspect of correctness. A class is said to be correct if it is consistent and every routine of the class is check-correct (i.e., no check instruction (see next section) in any method yields an error) and loop-correct (i.e., no loop invariant (see section “Loop invariants”) in any method yields an error).

Check Instructions. A check instruction defines a property that must be satisfied at the point of definition. Check instructions are comparable to assert Macros in C/C++ [1].

Loop Invariants. Correctness conditions on loops may be specified by means of loop invariants and loop variants. The invariant determines the properties ensured by the loop on exit, whereas the variant guarantees the termination of the loop’s execution.

3. DBC Model for Python

Correctness is defined by Pomberger [8] as “the software system’s fulfillment of the specification underlying its development”. Thus DBC is a more powerful approach than static type systems (as supported by programming languages like C++ and Java), as assertions are formal specifications of properties of a software system.

From this viewpoint it is legitimate to support the DBC approach in dynamically typed programming languages like Python. Nevertheless, static typing also contributes to the correctness of a software system. We therefore also provide simple mechanisms to support type checks on parameters and instance variables at least at run time.

The model described here is illustrated by the programming language Python. From a conceptual perspective the model is valid for any programming language. Section 4 discusses some architectural issues on implementing DBC for Python.

3.1 DBC Elements

Preconditions/Postconditions. In our Python model every method has its associated documentation. The basic idea of our Python DBC model is, to specify preconditions and postconditions in the documentation section of

a method. Preconditions and postconditions are checked according to the theory (see Section 2). Figure 3 shows a method *SetProperties* of class *Person*.

There may be an arbitrary number of *req:* and *ensure:* clauses but only one *param:* clause per method. The *param:* assertion is a feature beyond standard DBC concepts. The *param:* clause describes the type of parameters (not including the first parameter *self*) in the order of their occurrence. The *param:* clause leads to a run-time type-check of the parameters. Type checking is possible not only for Python built-in types but for arbitrary class types. We believe that this addition and its methodical use leads to better understandable Python Code and that it is a mechanism for higher correctness of the developed software.

```
def SetProperties(self, age,ssn):
    """
    param: IntType, IntType;
    olds.age: self.age;

    req: age > 0;
    req: self.age > 0 and self.age < 100;

    ensure: self.olds.age = self.age - 1;
    ensure: if age > 65: self.pensioner == 1;
    """
    self.age= age
    self.ssn=ssn
```

Figure 3. Python preconditions and postconditions

There are no special keywords (like *require else* and *ensure else* in Eiffel) necessary for overridden methods. The preconditions of the overridden method are automatically *or*-ed with the original preconditions, and the postconditions of the overridden method are automatically *and*-ed with the original postconditions. This rule is valid not only for direct ancestors of a class but also for the entire inheritance path.

Although Python supports multiple inheritance, inheritance of preconditions and postconditions is restricted to single inheritance. We impose this limitation for the following reasons:

- We do not want to encourage the use of multiple inheritance, as it is often difficult to understand [12]. In addition multiple inheritance can be reproduced by means of single inheritance and by use of the Adapter-Pattern [3].
- The model for solving name conflicts is poorly developed in Python. If an attribute is defined in more than one place in the hierarchy, Python takes the *closest* and *leftmost* version from the perspective of the object qualified [5]), which makes understanding multiple inheritance architectures even harder.

As shown in Figure 3, the specification of *old* expressions is possible. The right side of the *old* expression (*self.age* in our case) may consist not only of elementary data types but of arbitrary class types. In the latter case, a deep copy of the object is created at method entry.

Boolean implications (as described in Section 2) can be simulated by writing if statements *inreq:* or *ensure:* clauses (see *ensure: if age>100: self.pensioner == 1* in Figure 3).

Class invariants. In Python every class has, as does every method, its associated documentation. The basic idea is to specify class invariants in the documentation section of a class. Class invariants are checked according to the theory (see Section 2). Figure 4 shows class *Person*, a subclass of *LivingThing*.

```
class Person(LivingThing):
    """
    inv: self.age >= 0;
    inv: self.age < 100;
    inv: len(self.name) > 0;
    inv: self.socialSecurityNumber > 1000;
    """
    def __init__(self, a, n, ssn):
        """
        type: age: IntType,
              name: StringType,
              socialSecurityNumber: IntType;
        """
        self.age = a
        self.name = n
        self.socialSecurityNumber = ssn
```

Figure 4. Python invariants

Figure 4 shows the invariants of a class *Person*, defining a number of consistency conditions on every object of class *Person*.

Method *__init__*, which semantically is comparable to C++ constructor operations, shows an example of a *type:* clause. The *type:* clause provides a mechanism for describing the types of instance variables of a class. What we already stated for the *param:* clause is also valid for the *type:* clause. The only difference is that the *type:* clause is treated like a class invariant; i.e., the types of instance variables are always checked before and after invocation of a method.

Check Instructions. Check instructions can be placed anywhere in method implementations. A check instruction may contain one boolean expression. Figure 5 shows an example of method *SetAge* in class *Person*, which is a subclass of class *LivingThing*.

This check instruction could be replaced by a precondition, which would have a slightly different semantics, as the preconditions of the base class

LivingThing would have to be checked on program execution, too.

```
class Person(LivingThing):
    ...
    def SetAge(self, age):
        self.Check("age > 10 and age < 100")
```

Figure 5. Python check instructions

Loop Invariants. Loop invariants are currently not considered in our DBC model for Python, as they are a lower level kind of specification compared to invariants, preconditions and postconditions and check instructions. In this context we point out again that our emphasis is not on correct software but on providing mechanisms to combine formal approaches with object-oriented technology. For this purpose we do not consider loop invariants to be essential.

3.2 Handling Violations

Currently any failed assertion leads to an entry in a logfile that clearly records, the reason of the violation.

We consider this approach to be sufficient, as we see the application of our DBC/Python model in the analysis and design phase, where it suffices, to know that an exception occurred that contradicts the domain model developed.

Nevertheless, an exception handling mechanism could be incorporated, as Python provides language mechanisms for exception handling.

4. Alternatives for Implementing DBC

In principle there are two possible solutions to establish DBC support for a programming language. One is by means of directly changing the syntax of the language and by changing the semantics of the run-time system. The other approach is to use metaprogramming techniques.

4.1 Changing the Language

In order to support purely and elegantly DBC it would be necessary to include the mechanism in the language. Nevertheless, this approach would probably conflict with general language design principles of Python.

Python's main goal, from the viewpoint of language design, is simplicity; adding any feature to the language has to be carefully considered. In the case of DBC, a number of keywords would have to be added and the run-time system would have to be changed considerably.

Python is, in principle, an interpreted language where, comparable to Java, interim code is generated and executed. When a module is imported by another module,

Python internally compiles its code to an intermediate, portable form and stores it in a separate file. Subsequent imports of the module lead to a direct import of the byte-code file, thus avoiding the byte-code compile step. Thus to support DBC on the language level, the byte-code compiler had to be changed significantly, too; i.e., on passing flags to the interpreter (indicating whether DBC checking is enabled) corresponding code generation and recompilation had to take place.

The business of code generation and recompilation gets even more complicated when DBC checking can be enabled and disabled selectively, i.e., for certain classes, which is a desirable feature in prototyping-oriented development.

4.2 Metaprogramming Approach

As we saw in the previous section, considerable changes are necessary to support DBC on the language level. We believe that we can support DBC without changing the language and without imposing restrictions on the user of the system.

In order to implement the DBC support, we used metaprogramming techniques, where metaprogramming refers to “programming at the level of program interpretation, or in other words, to extending the interpreter of a given programming language in an application-specific way. Traditionally, this concept is available only in dynamically typed and interpreted languages” [13].

Execution Model. The basic idea of our execution model is to wrap every instance of a class for which DBC checking is required in a *Wrapper* object. Sending messages to this class results in the invocation of the `__getattr__` (`__getattr__` is comparable to `doesNotUnderstand:` in Smalltalk [4]) method. This mechanism allows us to intercept any method call and to make the appropriate DBC checks. Figure 6 sketches the implementation of the *Wrapper* class.

```
class Wrapper:
    def __init__(self, object):
        self.object= object
        object.dbc= self

    def __getattr__(self, name):
        attribute = getattr(self.object, name)
        if type(attribute) == types.MethodType:
            return Wrapper.Method(self.object, attribute)
        return attribute

class Method:
    def __init__(self, object, method):
        self.obj= object
        self.m= method

    def __call__(self, *args):
```

```
        dbcManager.CheckBefore(self.obj, self.m)
        retValue= apply(self.method, args)
        dbcManager.CheckAfter(self.obj, self.m)
        return retValue
```

Figure 6. Intercepting method calls

In the `__call__` method, the method call (`retValue = apply(self.method, args)`) is embedded in *CheckBefore* and *CheckAfter* statements, which perform the necessary DBC checks. We also have access to the arguments of the method by means of the *args* parameter. It is therefore possible to evaluate *param:* clauses if defined for the method currently executing.

```
class Foo:
    def foo:
        ...
    def foo1:
        self.dbc.foo()
```

Figure 7. Accessing the wrapper

The execution model presented here has some drawbacks:

- It is not possible to use local variables of a method in preconditions, postconditions or check instructions, as checking is done outside the method scope.
- If a method calls another method of the same object, DBC checking can only take place if and only if the message is sent via the wrapper object. To provide support for this, the wrapper object registers itself in the wrapped object (see Method `__init__` in Figure 6). The assignment `object.dbc= self` in Method `__init__` (see Figure 6) dynamically adds the instance variable *dbc* to the wrapped object. Without this capability it would be necessary to subclass every class with DBC support from a common base class *DBC*. The *DBC* class would maintain the reference to the wrapper object. Figure 7 shows an example of how to access the wrapping object.
- The mechanism works due to the fact that instance variables in Python are not protected and therefore can be accessed from outside. Nevertheless, this is not a drawback of our DBC model but a Python language feature that eases the implementation of DBC.

We believe that the restrictions imposed and the programming style forced (especially for calling methods of the same class) do not substantially affect programmers.

The process of DBC checking is controlled by an environment variable (to switch on and off checking) and by a configuration file. The configuration file consists of a list of module/class pairs that have to be checked. Thus by altering the configuration file, DBC checking can selectively be enabled and disabled for certain classes. In

addition, it is possible to exclude certain objects of a class to be checked, simply by not wrapping them with a *Wrapper* object. From the perspective of correctness, this freedom is not desirable, but from the perspective of the proposed use, i.e., in the analysis and design phase, it is a very convenient feature.

Parsing. We provide support for *traditional parsing* (before or at application startup) and a *late parsing* approach. Both parsing techniques heavily rely on the reflective capabilities of Python. This means that not the source code of files is parsed, but the metainformation provided by the run-time system. This approach was applicable, as the necessary metainformation (e.g., class/base class relationships, methods of a class, documentation of a class or method) is available at run time. This availability reduces the parsing effort substantially, as parsing is based on prestructured symbol information.

In Python, the metainformation of a class (and its base classes) is available automatically after class loading; thus to implement the traditional parsing approach, dynamic class loading based on a textual specification (a configuration file) had to be used. An appropriate mechanism is provided by Python.

In case of the late parsing approach, interception of the class loading mechanism is necessary to be able to parse the metainformation without having to change the code to be parsed. Whether a class had already been parsed and whether DBC support should be provided by a class is checked at loading time. The configuration file already mentioned controls the parsing activities, i.e., whether a class has to be parsed. If a class has to be parsed (according to the configuration file), all its base classes are parsed, too. This is necessary to avoid conflicts with the theory of subcontracting (see Section 2).

We encourage the use of the late parsing technique as it guarantees that the currently relevant assertions are considered. We believe that the traditional parsing technique is an additional feature to provide symbol information, i.e., DBC information, in a persistent form that can be used by other tools.

Metaprogramming concepts. In the above section we showed how to build an execution model and a parser by means of metaprogramming techniques. The approach presented here is also suitable for other dynamically typed programming languages.

<i>Technique</i>	<i>Purpose</i>
<i>Method call interception</i>	Necessary to perform DBC checks before and after method invocation
<i>Instance variable extension</i>	To be able to maintain a reference to the wrapper object in the wrapped object without having to

<i>Technique</i>	<i>Purpose</i>
	inherit this feature from a base class
<i>Intercepting module (class) loading</i>	Necessary for late parsing technique to automatically gain the relevant DBC information
<i>Reflection</i>	Provides access to class objects, method objects, and documentation of a method or a class; allows the implementation of parsers on the basis of the metainformation provided

Table 1. Metaprogramming techniques

The effort necessary depends on the metaprogramming support available in the respective language. Table 1 lists the metaprogramming techniques used.

5. Benefits

We do not discuss the benefits of DBC in Python from the perspective of correctness; instead, we relate it to the phases of the software life cycle.

Figure 8 illustrates our idea of a software life cycle [8]. The emphasis of this prototyping-oriented life cycle is on the two planned iterations in the analysis and design phase (user interface prototyping, architecture and component prototyping). We see the main benefits of a DBC solution with Python within these two iterations.

- There is a general need for formalization (see [2]). We believe that the proposed combination of an object-oriented, dynamically typed, interpreted programming language and a formal method like DBC is a perfect combination of informal and formal approaches in the first phases of the software life cycle.
- In prototyping-oriented development, an interpretable and lean language far outperforms a statically typed programming language like Eiffel. This makes our DBC/Python approach especially valuable for prototyping-oriented development, no matter whether the DBC technique is used during requirements elicitation and specification or in the design phase, where the emphasis is on evaluating different design approaches.

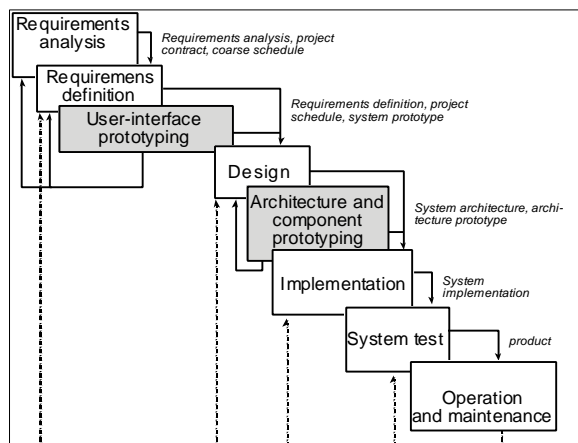


Figure 8. Prototyping-oriented software life cycle

Support in the implementation and/or test phase is possible in cases where the software system is implemented in Python (and not just prototyped with it). In this case the assertions developed in the analysis and design phase can be of direct benefit in the implementation and especially testing phase, as they are part of the software system and reflect parts of the specification.

In cases where the target language is different from the prototyping language, direct benefit from the DBC technique can only be drawn if appropriate tool support is available (see next section).

6. Tool Support

We regard tool support to be essential in the analysis and design phase. Furthermore tool support is required to make use of our approach in the implementation and test phases.

Especially with the emphasis on using this technique for requirements elicitation and documentation as well as in the design phase (architecture prototyping), there is a need for an interactive tool that allows easy inspection and editing of assertions, inherited assertions, etc.

To support this approach throughout the entire software life cycle, adapters are necessary to transform Python code (especially the class structure, method signatures and the assertions) to the target language of the project, e.g., C++, Java. This needs smart bi-directional transformational tools, as most statically typed programming languages do not have the amount of metaprogramming support available that would be necessary for a smooth and straightforward transformation process.

7. Conclusion

In this paper we showed that the concept of DBC can be integrated with a dynamically typed object-oriented

programming language. We believe that this approach is interesting for the analysis and design phase of a prototyping-oriented software life cycle. We also showed that the implementation is possible without directly changing the language syntax but by means of metaprogramming techniques. This implementation approach can also be used for other dynamically typed object-oriented programming languages. We also believe that further tool support (especially interactive browsers) is necessary to fully benefit from this approach in a prototyping-oriented procedure.

Bibliography

- [1] Banahan M., Brady D., Doran M.: "The C Book - Featuring the ANSI C Standard (Second Edition)", Addison-Wesley, The Instruction Set Series, 1991
- [2] Bowen J. P., Hinchey M. G.: "Ten Commandments of Formal Methods", IEEE Computer, IEEE Computer Society, April 1995, pp 56-63
- [3] Gamma E., Helm R., Johnson R., Vlissides J.: "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, Professional Computing Series, Reading, Massachusetts, 1994
- [4] Goldberg A., Robson D.: "Smalltalk-80 - The Language and its Implementation", Addison Wesley series in Computer Science, Reading, Massachusetts, 1983
- [5] Lutz M.: "Programming Python", O'Reilly & Associates, Sebastopol, 1996
- [6] Meyer B.: "Eiffel - The Language", Prentice Hall, Object-Oriented Series, Hemel Hempstead, 1992
- [7] Meyer B.: "Building bug-free O-O software: An introduction to Design by Contract", Object Currents, SIGS Publication, Vol. 1, No. 3, March 1996
- [8] Pomberger G., Blaschek G.: "Object-Oriented Prototyping in Software Engineering", Prentice Hall, The object-oriented series, Hemel Hempstead, 1996
- [9] Rossum van, G.: "Python Language Reference", electronic hypertext document, available from <http://www.python.org>, 1996
- [10] Rossum van, G.: "Python Library Reference", electronic hypertext document, available from <http://www.python.org>, 1996
- [11] Sametinger J., Stritzinger A.: "A Documentation Scheme for Object-oriented Software Systems", OOPS Messenger, ACM Press, Vol. 4, No. 3, July 1993, pp. 6-17
- [12] Shang D. L.: "Multiple Inheritance - A Critical Comparison on Transframe, Java & C++", Object Currents, SIGS Publication, Vol. 1, No. 11, November 1996
- [13] Templ J.: "Metaprogramming in Oberon", ETH Dissertation No. 10655, Zurich, 1994