

Interfaces: Design by Contract

2.9.2019

A good interface?

- **Complete**
- **Beautiful**
- **Cute**



Fig: clement127 (CC BY-NC-ND 2.0)

The purpose of an interface?



Fig: Clement127 (CC BY-NC-ND 2.0)

Where do the interfaces come from?

Main steps in designing a program:

Component

- identification
- responsibilities
- connections
- **Specifying interfaces**

Interface specification

- What is the use **allowed** by an interface?
- What do the functions behind an interface **promise** to do?
- What kind of **errors** are possible in the functions?
- How to **test** an interface?

Design by Contract

- A clean metaphor to guide the design process

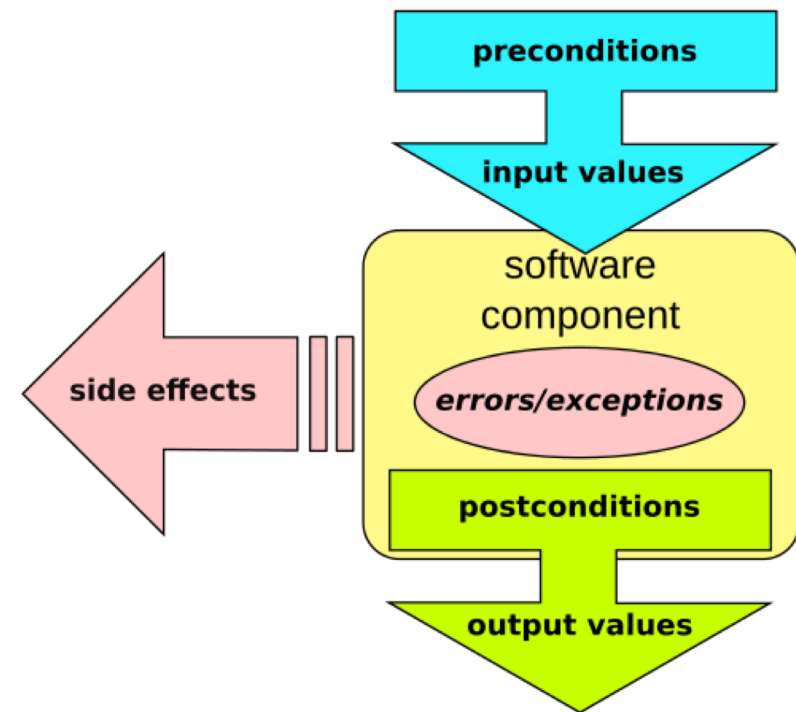


Fig: Fabuio (Own work) [CC0], via Wikimedia Commons

Design by contract

Interface specification given as a **contract**

- Client (caller) and supplier (implementor)
- Mutual obligations and benefits

Contract between an interface and its user

- Responsibility of a caller: how the interface is allowed to be used? (precondition)
- Responsibility of an implementor: what does an interface promise to do? (postcondition)
- Specification of errors

```
{P} o.service() {Q}
```

Precondition

- Must be true **before** a service
- **Caller** is responsible on fulfillment
- “When/how to call a service?” or “What does the service expect?”
- E.g. $\{a < 10 \wedge b < 20\}$

Postcondition

- Must be true **after** the service
- **Implementor** is responsible on fulfillment
- “What does a service promise to do” or “What does it guarantee?”
- Violation of postcondition leads to an exception
- E.g. $\{j < 30 \wedge a < 10 \wedge b < 20\}$

Obligations

Client

- Takes care of fulfilling the precondition
- Preconditions can be checked during testing, not in the final program

Supplier

- Takes care of fulfilling the postcondition
- No more checks of the obligations of the caller
- If the service fails (violation of the postcondition) \Rightarrow exception, to be informed about

(Class) invariant

Logical assertion that is held to always be true (during a certain phase of the execution)

- Tests if an object is valid or “in its right mind”
- Must be true between the calls

$\{\text{CLASS_INVARIANT} \wedge P\} o.\text{service}() \{\text{CLASS_INVARIANT} \wedge Q\}$

- Useful for the implementor, not for the caller

Example

```
class Date
{
public
    setDay( int day );
private
    int d_;
    int m_;
    int y_;
};
```

- Invariant?
- Precondition?
- Postcondition?

C++20: Contracts

- C++20 enables writing pre and post conditions and invariants as part of the code

```
double sqrt(double x) [[expects: x >= 0]];  
void sort(vector<emp>& v) [[ensures audit:  
is_sorted(v)]];
```

C++20: Contracts

```
int push(queue& q, int val)
    [[ expects: !q.full() ]]
    [[ ensures: !q.empty() ]]
    {
        ...
        [[assert: q.is_ok() ]]
        ...
    }
```

Conditions in C++ standard

25.4.3.4 `binary_search`

[`binary.search`]

```
template<class ForwardIterator, class T>
    bool binary_search(ForwardIterator first, ForwardIterator last,
                      const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
    bool binary_search(ForwardIterator first, ForwardIterator last,
                      const T& value, Compare comp);
```

- 1 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expressions `e < value` and `!(value < e) or comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` implies `!(value < e) or comp(e, value)` implies `!comp(value, e)`.
- 2 *Returns:* `true` if there is an iterator `i` in the range `[first,last)` that satisfies the corresponding conditions: `!(*i < value) && !(value < *i) or comp(*i, value) == false && comp(value, *i) == false`.
- 3 *Complexity:* At most $\log_2(\text{last} - \text{first}) + \mathcal{O}(1)$ comparisons.

Conditions in C++ standard

25.4.1.1 sort

[sort]

```
template<class RandomAccessIterator>
    void sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
    void sort(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

- 1 *Effects:* Sorts the elements in the range `[first,last)`.
- 2 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).
- 3 *Complexity:* $\mathcal{O}(N \log(N))$ (where $N == \text{last} - \text{first}$) comparisons.

Design by Contract: documentation

Documentation

```
mapped_type& operator [] (const key_type& k);  
mapped_type& operator [] (key_type&& k);
```

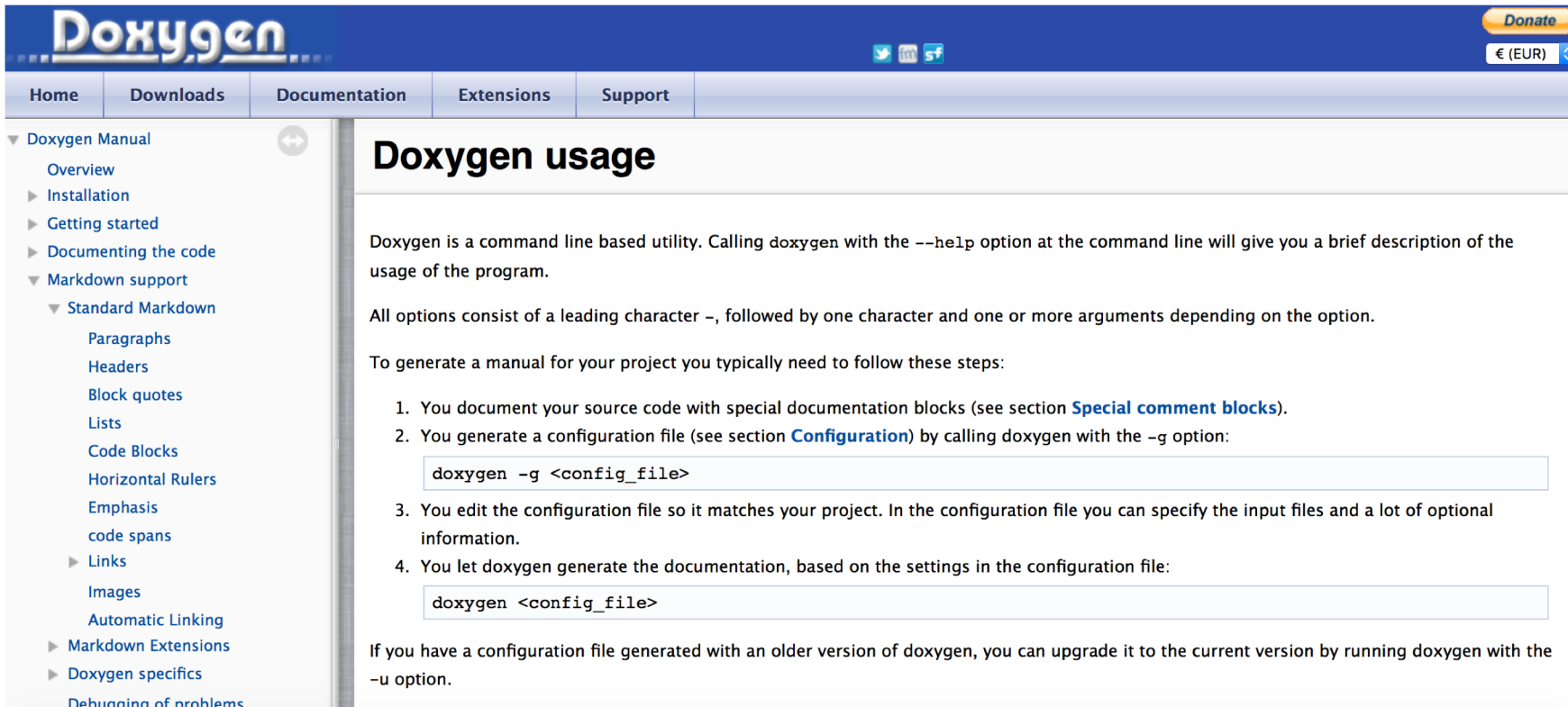
- 1 *Requires:* `mapped_type` shall be `DefaultInsertable` into `*this`. For the first operator, `key_type` shall be `CopyInsertable` into `*this`. For the second operator, `key_type` shall be `MoveConstructible`.
- 2 *Effects:* If the `unordered_map` does not already contain an element whose key is equivalent to `k`, the first operator inserts the value `value_type(k, mapped_type())` and the second operator inserts the value `value_type(std::move(k), mapped_type())`.
- 3 *Returns:* A reference to `x.second`, where `x` is the (unique) element whose key is equivalent to `k`.
- 4 *Complexity:* Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\text{size}())$.

What do you know about the behavior of the operator `[]` based on documentation?

Hiding the implementation

- Pre- and postconditions will be *documented* in a form that can be understood by users
- If a class **tests** the conditions or invariant, test will be written based on internal implementation

Documentation: Doxygen



The screenshot shows the Doxygen website's documentation page. The top navigation bar includes 'Home', 'Downloads', 'Documentation', 'Extensions', and 'Support'. A 'Donate' button and a currency selector (€ (EUR)) are also visible. The left sidebar contains a tree view of the documentation structure, with 'Doxygen Manual' expanded to show 'Standard Markdown' and its sub-items: Paragraphs, Headers, Block quotes, Lists, Code Blocks, Horizontal Rulers, Emphasis, code spans, Links, Images, Automatic Linking, Markdown Extensions, Doxygen specifics, and Debugging of problems. The main content area is titled 'Doxygen usage' and contains the following text:

Doxygen is a command line based utility. Calling `doxygen` with the `--help` option at the command line will give you a brief description of the usage of the program.

All options consist of a leading character `-`, followed by one character and one or more arguments depending on the option.

To generate a manual for your project you typically need to follow these steps:

1. You document your source code with special documentation blocks (see section [Special comment blocks](#)).
2. You generate a configuration file (see section [Configuration](#)) by calling `doxygen` with the `-g` option:

```
doxygen -g <config_file>
```
3. You edit the configuration file so it matches your project. In the configuration file you can specify the input files and a lot of optional information.
4. You let `doxygen` generate the documentation, based on the settings in the configuration file:

```
doxygen <config_file>
```

If you have a configuration file generated with an older version of `doxygen`, you can upgrade it to the current version by running `doxygen` with the `-u` option.

Documentation: Doxygen

KDE API Reference

KDE API Reference

Navigation

- [Main Page](#)
- [Old KDE4 Versions](#)


Related

- [API Doc Tutorial](#)
- [KDE TechBase](#)
- [KDE CMake Modules](#)
- [Extra CMake Modules](#)

Search


API Reference Index

The reference guides for the KDE APIs -- for KDE2 all the way to the current development version -- are collected here. We assume you are already familiar with the excellent [Qt4](#) documentation. [TechBase](#) is the right place to start looking for general development information for KDE. There are only reference guides here.

To obtain a gzip compressed tar file containing the documentation, click on the  images, which are immediately adjacent to many of the listed items.

To obtain a version of the documentation for use in [Digia Qt Assistant](#), click on the "[qch]" links, which are immediately adjacent to some of the listed items. (In Qt assistant, go into Edit->Preferences->Documentation and [Add] the .qch file.)

Man pages are also provided for some modules. Click on the "[man]" links, also immediately adjacent to some of the listed items to download a bzip2 compressed tar file containing the man pages for the corresponding module. (Uncompress and untar these files into a standard MANPATH directory.)

Frameworks	Others
 frameworks5 [qch][man]	<ul style="list-style-type: none">Other KDE SoftwareKDE4 VersionsKDE3 and older versions

Documentation: Doxygen

```
/**  
... text ... */  
or  
/*!  
... text ...  
*/
```

Documentation: Doxygen

```
\pre { precondition }
```

```
\post { postcondition }
```

```
\throw <exception_object> { exception }
```

```
\invariant { invariant }
```

Using contracts

- Lectures typically give only simple examples
- In practice:
 - Do not scale well
 - Specification that is mathematically exact is difficult (and usually unnecessary)
 - Inheritance brings difficulties

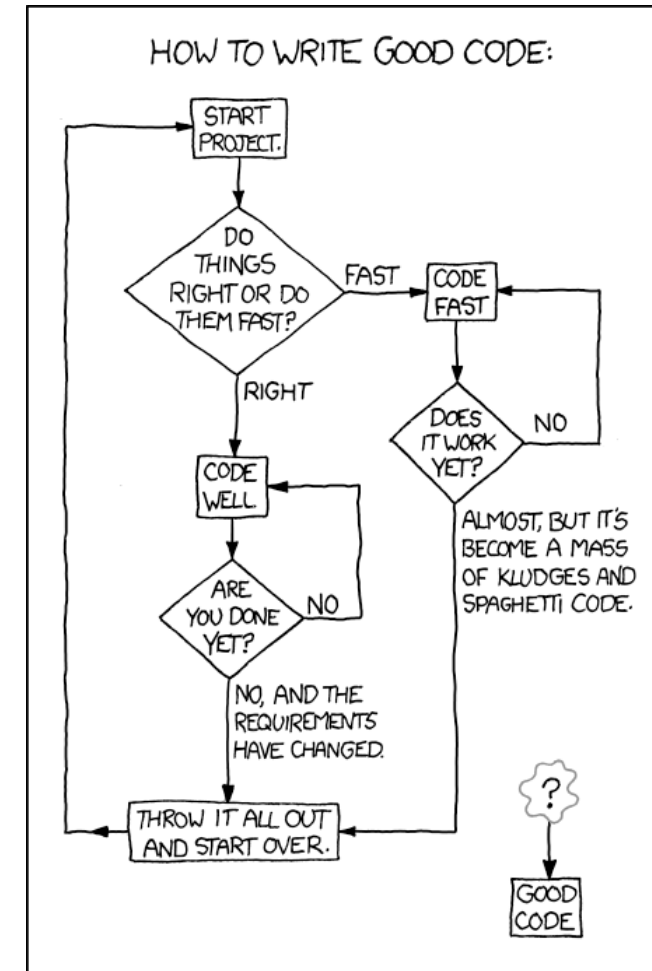
Other contracts and practices

Coding conventions

Improve readability

Things to be agreed

- Comments (e.g. Doxygen)
- Indentations
- Length of rows
- Naming
- Coding practices and principles, rules of thumb
- Style issues



Pair programming

- Originates from eXtreme Programming
- Two programmers: controller and observer
 - Improves quality: decreases errors
 - Team work and communication
 - Learning

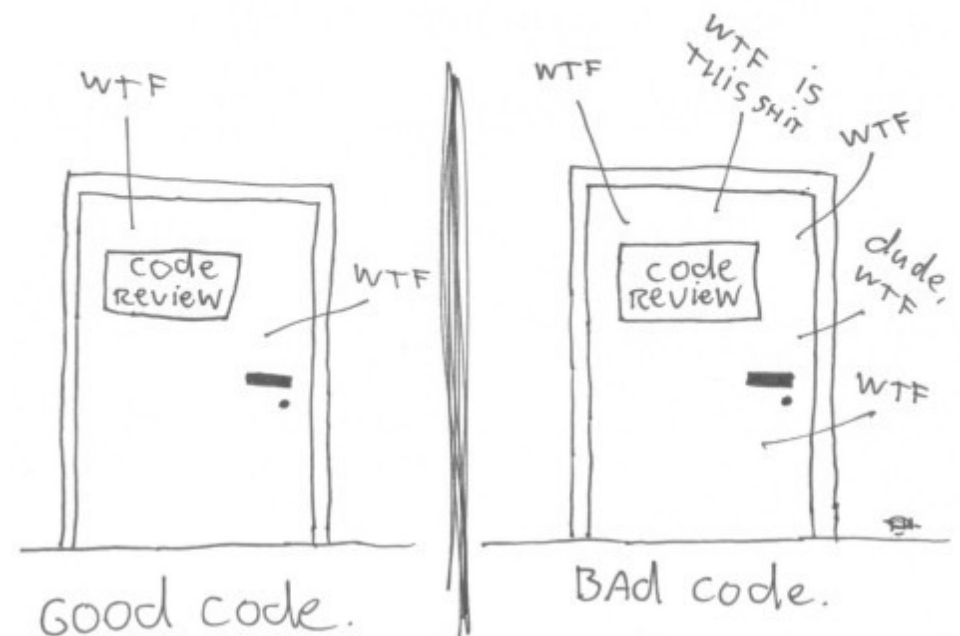
Code review

Reading code and examining:

- Does it do what it should do?
- Does it follow coding conventions?
- Does it have errors?

Fig: osnews.com/Thom Holwerda

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

Design by Contract: testing

Design by contract and testing

- Complements regular testing strategies: unit testing, integration testing, system testing
- Testing also preconditions
- Integration testing for free
- Support for debugging: contract violations help locating errors

Testing conditions: C++

Assert macro in C++ tests conditions and crashes the program if the condition is false

```
#include <cassert>
void f( int i )
{ assert( i >= 0 );}
```

NDEBUG ⇒ assert does nothing

Invariant

```
inline void OrderedTable::Invariant()  
{  
#ifndef NDEBUG  
    // Invariant: items are always ordered such that index 1 contains the  
    smallest item and  
    // index SIZE contains the largest one  
    for( int i = 1; i < SIZE; i++ )  
    {  
        if( item[ i ] > item[ i+1 ] )  
            throw OrderedTable::InvariantBroken();  
    }  
#endif  
}  
  
void OrderedTable::searchAndChange( Item const& wanted, Item const&  
substitution )  
{  
    Invariant();  
    // Actual implementation  
    Invariant();  
}
```


Testing conditions: Qt

Qt provides more functions/macros

- `#include <QtGlobal>`
- `Q_ASSERT` macro

- Can be taken off: `QT_NO_DEBUG`

```
void Q_ASSERT(bool test)  
void Q_ASSERT(bool test, const char* where, const  
char* what)
```

Testing conditions: Qt

```
int divide(int a, int b)
{
    Q_ASSERT(b != 0);
    return a / b;
}
```

⇒ ASSERT: "b == 0" in file div.cpp, line 7

Testing conditions: Qt

Correspondingly:

```
    Q_ASSERT_X(b != 0, "divide", "division by  
zero");  
    return a / b;
```

⇒ ASSERT failure in divide: "division by zero",
file div.cpp, line 7