

Bonnes pratique logicielles, test unitaires etc.

2022-11-21

Inria Rennes

Constat sur les codes en recherche

Constat sur les codes en recherche

On n'écrit pas de code pour des clients, on écrit du code pour des reviewers (pas la même notion de durée).

- on part d'une idée pré-existante
- on code
- on modifie l'idée de départ
- on modifie le code
- on répète
- on publie
- (on oublie)

Conséquence de la méthodologie

Pas de mises à jour \implies un bug a tendance à rester.

De nouveaux bugs peuvent aussi apparaître, ex:

- python 3.10 n'autorise plus l'affichage de grand nombre
- le passage de PHP 7.3 à 7.4 introduit un nouveau mot-clef et n'installe plus PEAR par défaut

Architecture logicielle

La plus facile à mettre en oeuvre sur les langage orienté objet est la l'architecture "orientée objets" .

Un des principe clef en est l'**encapsulation**.

Chaque composant n'expose que le minimum nécessaire.

Pensez à bien nommer vos composants!

Ne pas faire:

```
class Parent {  
    public:  
        Enfant* _enfant;  
        int truc;  
  
    Parent() {  
        _enfant = new Enfant(this);  
    }  
};
```

Ne pas faire:

```
class Enfant {  
    public:  
        Parent* _parent;  
  
        Enfant(Parent* parent) {  
            _parent = parent;  
        }  
  
        // this->parent->truc  
};
```


Faire:

```
class Parent {  
    public:  
        Enfant* _enfant;  
        int truc;  
  
    Parent() {  
        _enfant = new Enfant(truc);  
    }  
};
```

Faire:

```
class Enfant {  
    public:  
        int _truc;  
  
    Enfant(int truc) {  
        _truc = truc;  
    }  
  
    // this->truc  
};
```

Utilisation des langages

Un peu de python

```
def add_nb_lines(d, filenames):  
    for filename in filenames:  
        with open(filename, 'r') as fichier:  
            n = 0  
            for _ in fichier:  
                n += 1  
            d[filename] = n
```

Seems OK, but:

- no type annotation
- no error handling
- no documentation

Un peu de python

```
def add_nb_lines(d: dict[str, int], filenames: list[str]):  
    for filename in filenames:  
        with open(filename, 'r') as fichier:  
            n :int = 0  
            for _ in fichier:  
                n += 1  
            d[filename] = n
```

Seems OK, but:

- type-annotation
- no error handling
- no documentation

Un peu de python

```
def add_nb_lines(d: dict[str, int], filenames: list[str]):  
    for filename in filenames:  
        try:  
            with open(filename, 'r') as fichier:  
                n :int = 0  
                for _ in fichier:  
                    n += 1  
                d[filename] = n  
        except FileNotFoundError as e:  
            print(f"{filename} could not be opened")
```

Seems OK, but:

- type-annotation
- no error-handling
- no documentation

Un peu de python

```
def add_nb_lines(d: dict[str, int], filenames: list[str]):  
    """Add number of line of every file in d."""  
    for filename in filenames:  
        try:  
            with open(filename, 'r') as fichier:  
                number_of_line :int = 0  
                for _ in fichier:  
                    number_of_line += 1  
                d[filename] = number_of_line  
        except FileNotFoundError as e:  
            print(f"{filename} could not be opened")
```

Perfection !

Des ribambelles de tests

Pourquoi tester ? Ça se voit que ça marche

```
def assign_integer(t: tuple[int]):  
    try:  
        t[0] += 1  
    except TypeError as e:  
        print(e)
```

```
def assign_list(t: tuple[list[str]]):  
    try:  
        t[0] += ["truc"]  
    except TypeError as e:  
        print(e)
```

Pourquoi tester ? Ça se voit que ça marche

```
from tuple_assign import assign_integer, assign_list

tuple_integer = (0,)
print("tuple_integer before += :", tuple_integer)
assign_integer(tuple_integer)
print("tuple_integer after += :", tuple_integer)

print()

tuple_list : tuple[list[str]] = ([],)
print("tuple_list before += :", tuple_list)
assign_list(tuple_list)
print("tuple_list after += :", tuple_list)
```

Pourquoi tester ? Ça se voit que ça marche

tuple_integer before + =: (0,)

'tuple' object does not support item assignment

tuple_integer after + =: (0,)

tuple_list before + =: ([],)

'tuple' object does not support item assignment

tuple_list after + =: (['truc'],)

Pourquoi tester ? Ça se voit que ça marche

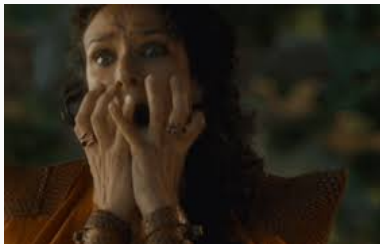
```
def f(l=[]):  
    l.append(3)  
    return l  
  
print(f([0, 1, 2])) # [0, 1, 2, 3]  
print(f()) # [3]
```

Pourquoi tester ? Ça se voit que ça marche

```
def f(l=[]):  
    l.append(3)  
    return l  
  
print(f([0, 1, 2])) # [0, 1, 2, 3]  
print(f()) # [3]  
print(f()) # [3, 3]
```

Pourquoi tester ? Ça se voit que ça marche

```
def f(l=[]):  
    l.append(3)  
    return l  
  
a = f()  
a.append(5)  
print(f())  # [3, 5, 3]
```



OK, OK... Comment je vérifie mon programme ?

Par des tests !

- tests unitaires
- tests d'intégration
- tests de validation

Le but est de vérifier le comportement de chaque unité de code, indépendamment des autres unités.

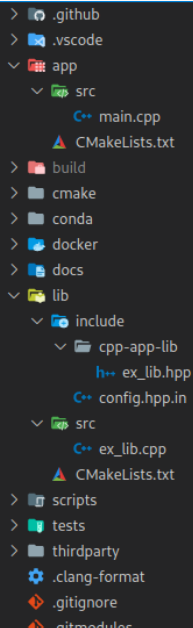
Il existe généralement des solutions pour chaque langage.

Tests unitaires

L'utilisation de bout de code "inutiles" (les mocks) est possible pour isoler les unités à tester.

```
class AMQ {
public:
    virtual int get(const std::string& kmer);
};
You, il y a 26 secondes | 1 author (You)
class MockAMQ : public AMQ {
public:
    MOCK_METHOD(int, get, ());
};
{
    // in a test
    MockAMQ mockAMQ;
    EXPECT_CALL(mockAMQ, get())
    |>>>.Times(AtLeast(1));
}
```

Tests unitaires (C++)



A screenshot of a file explorer window showing a project structure. The root directory contains several folders and files. The 'app' folder is expanded, showing a 'src' subfolder with 'main.cpp' and 'CMakeLists.txt'. The 'lib' folder is also expanded, showing an 'include' subfolder with 'cpp-app-lib' (containing 'ex_lib.hpp' and 'config.hpp.in') and a 'src' subfolder with 'ex_lib.cpp' and 'CMakeLists.txt'. Other folders visible include '.github', '.vscode', 'build', 'cmake', 'conda', 'docker', 'docs', 'scripts', 'tests', 'thirdparty', '.clang-format', '.gitignore', and 'gitmodules'.

- > .github
- > .vscode
- ▼ app
 - ▼ src
 - main.cpp
 - CMakeLists.txt
- > build
- > cmake
- > conda
- > docker
- > docs
- ▼ lib
 - ▼ include
 - cpp-app-lib
 - ex_lib.hpp
 - config.hpp.in
 - ▼ src
 - ex_lib.cpp
 - CMakeLists.txt
- > scripts
- > tests
- > thirdparty
- .clang-format
- .gitignore
- gitmodules

Le but est de vérifier le comportement du programme total, en utilisant un maximum d'unités.

Cela permet de détecter des erreurs de communication entre les unités ou de détecter des API cassées.

Tests d'intégration

The screenshot displays the Beeceptor web interface for configuring and testing an API endpoint. The interface is divided into a left sidebar, a top navigation bar, and a main workspace.

Top Navigation Bar: Includes buttons for 'New', 'Import', 'Runner', and 'Invite'. The current project is 'Practical Programming'.

Left Sidebar: Contains a 'Filter' search bar and tabs for 'History', 'Collections', and 'APIs'. Under 'Collections', there is a '+ New Collection' button and a 'Trash' section. The 'First Collection' contains 3 requests, and 'Order management' contains 3 requests.

Main Workspace:

- POST CREATE OBJECT:** The active tab for editing the request.
- HTTP METHOD:** A dropdown menu currently showing 'POST'.
- URL:** A text input field containing '(url)/'.
- Buttons:** 'Send' (blue) and 'Save' (grey) buttons.
- Tabs:** 'Params' (selected), 'HEADERS', 'BODY', and 'PARAMS'. Below these are 'Authorization', 'Headers (0)', and 'Body'.
- Query Params:** A table with columns 'KEY' and 'VALUE'. The first row has 'Key' in the key column and 'Value' in the value column. A 'DESCRIPTION' column is also present.
- Body:** The selected tab for the request body. It shows a JSON object:

```
{  "id": "1",  "fizz": "bar"}
```

. The status bar indicates 'Status: 201 Created', 'Time: 169 ms', and 'Size: 231 B'. A 'Save Response' button is visible.
- Response View:** At the bottom, there are tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. The 'Pretty' tab is selected, showing the formatted JSON response.

Bottom Bar: Includes a 'Bootcamp' link, 'Build' and 'Browse' buttons, and a help icon.

- scénarios d'utilisation concrète
- simule ce qu'un reviewer ferait pour rapidement voir que "ça marche"

C'est aussi à cette étape que sont réalisés les tests de performance par exemple.

Quelques petites remarques:

Tester est chronophage (en particulier, l'utilisation de mock).

Préférer tester les plus petites unités d'abord, puis remonter l'arbre de dépendance.

⇒ fusion des tests unitaires et d'intégration.

Quelques petites remarques:

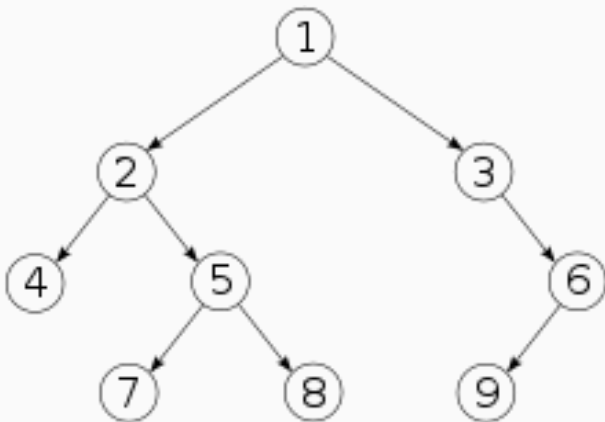


Figure 1: proposition d'ordre: 4, 7, 8, 9, puis 2, 5, 6, puis 3 puis 1

Outil permettant:

- de compiler le code à chaque mises à jour
- d'exécuter les tests
- de mettre à disposition des rapports