MEDEVRV
Qualité logiciel : métriques et documentation

basé sur des cours de Théo Combe

Olivier Augereau
Maitre de conferences ECN/LS2N

REINVENT ENGINEERING
CENTRALE NANTES

Course Content

- Version control / Git
- Unified Modeling Language (UML)
- **Agile methods**
- Code metrics / Code documentation
- Group projects

Métriques du code

Métrique du code

Ensemble de mesures qui permettent d'avoir des indices de qualité et complexité sur le code développé

- Permet de savoir quelle partie du code retravailler en priorité
- Permet d'identifier certains risques

Ce ne sont que des nombres ! A vous de construire la connaissance associée…

Pour être utiles, ces métriques doivent être faciles à calculer, si possible automatiquement.

Métrique du code – Source Monitor

File Name	Lines	Statements	% Branches	% Comments	Class Defs	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complexity	Functions
Carte.cpp	4*	3	0,0	0,0	0	1,00	1,0	1	1	0,33	1,00	0
Carte.h	17*	13	0,0	0,0	1	1,00	1,0	1	2	0,77	1,00	0
Carte_action.cpp	13*	10	10,0	0,0	0	2,00	3,5	2	2	0,80	1,50	0
Carte_action.h	21*	19	0,0	0,0	1	3,00	1,3	1	2	0,95	1,00	0
Carte_point.cpp	9*	7	0,0	0,0	0	2,00	2,0	1	1	0,57	1,00	0
Carte_point.h	16*	17	0,0	0,0	1	3,00	1,0	1	2	0,88	1,00	0
Joueur.cpp	52*	32	28,1	9,6	0	5,00	5,2	9	4	1,41	2,80	0
Joueur.h	20*	19	0,0	0,0	1	3,00	1,0	1	2	1,00	1,00	0
JoueurAttaque.cpp	47*	26	57,7	12,8	0	1,00	23,0	16	4	1,92	16,00	0
JoueurAttaque.h	6*	3	0,0	0,0	1	0,00	0,0	0	1	0,33	0,00	0
JoueurPoint.cpp	47*	26	57,7	12,8	0	1,00	23,0	16	4	1,92	16,00	0
JoueurPoint.h	6*	3	0,0	0,0	1	0,00	0,0	0	1	0,33	0,00	0
LesCartes.cpp	72*	43	20,9	6,9	0	5,00	6,6	6	3	1,23	2,80	0
LesCartes.h	20*	14	0,0	0,0	1	1,00	1,0	1	2	0,71	1,00	0
main.cpp	96*	61	32,8	5,2	0	0,00	0,0	23	5	2,11	23,00	1
SetDeCartes.cpp	29*	18	22,2	0,0	0	4,00	3,2	3	3	1,06	2,00	0
SetDeCartes.h	20*	13	0,0	0,0	1	1,00	1,0	1	2	0,69	1,00	0

https://www.derpaul.net/SourceMonitor/

https://jacobfilipp.com/DrDobbs/articles/DDJ/2000/0003/0003i/0003i.htm

Statements: in C++, computational statements are terminated with a semicolon character. Branches such as **if**, **for**, **while** and **goto** are also counted as statements. The exception control statements **try** and **catch** are also counted as statements. Preprocessor directives **#include**, **#define**, and **#undef** are counted as statements. All other preprocessor directives are ignored. In addition all statements between each **#else** or **#elif** statement and its closing **#endif** statement are ignored, to eliminate fractured block structures.

Percent Branch Statements: Statements that cause a break in the sequential execution of statements are counted separately. These are the following: **if**, **else**, **for**, **while**, **break**, **continue**, **goto**, **switch**, **case**, **default**, and **return**. Note that **do** is not counted because it is always followed by a **while**, which is counted. The try block statement **catch** is also counted as a branch statement.

Percent Lines with Comments: The lines that contain comments, either C style (/*...*/) or C++ style (//...) are counted and compared to the total number of lines in the file to compute this metric. If you select the project option to ignore headers and footers (see create project, step 3 or the project properties dialog), contiguous comment lines at the beginning and end of files are **not** counted as comments.

Classes: Classes (including structs) are counted on the basis of their definitions. In general, classes are defined in header files while methods are usually implemented in separate files. Therefore the number of classes may be different from the number of classes for which a biggest method has been measured (see below). Template definitions are counted as class definitions.

Methods per Class: Both inline and non-inline class, struct, and template class implementations are counted. This metric is an overall average for all class, struct, and template method implementations in a file or checkpoint, computed as the total number of methods divided by the total number of classes and templates for which method implementations are found.

Functions: Number of functions found.

Average Statements per Method: The total number of statements found inside of methods found in a file or checkpoint divided by the number of methods found in the file or checkpoint.

Maximum Method or Function Complexity: The complexity value of the most complex method or function in a file. (Note: when the Modified Complexity option is enabled, average complexity values are displayed with a trailing asterisk. Example: "4.56*".) The Project and Checkpoint Views display only the method or method with maximum complexity; however, in the Details View for a file (double click on a file in Checkpoint View) the complexities of all methods and functions are listed, along with all method metrics. All method complexities are shown in the Method View as well.

Calls: The total number of calls to other methods or functions found inside of all methods or functions in a file or checkpoint are displayed in the Method View.

Métrique du code – Source Monitor

Nombre total de lignes : compte tout ce qui est dans le fichier, y compris les espaces et commentaires.

Statements : compte les instructions exécutables

Un nombre élevé peut signaler un projet complexe ou mal structuré, mais il faut le relativiser selon le contexte. Une augmentation brutale peut indiquer une duplication de code.

% Branches : les branches sont des instructions qui modifie le flux d'exécution (if, else, for, while, switch, case, try, catch, etc.). Une valeur > 30% peut indiquer un code très conditionnel, avec beaucoup de logique de contrôle, ce qui peut le rendre plus complexe à comprendre, tester et maintenir.

% Comments

Un bon ratio commentaires/code (généralement entre 10% et 50%) suggère un code bien documenté. Trop peu de commentaires peut rendre le code difficile à maintenir, tandis qu'un excès peut indiquer un code trop complexe ou mal écrit.

Métrique du code – Source Monitor

Class defs : nb de classes et struct

Method/class : Une grande valeur (>20) indique qu'une classe assume trop de responsabilités. Cela rend le code plus difficile à maintenir, à tester et à comprendre.

Avg stmts / method : idem (conseillé < 25)

Max complexity : complexité cyclomatique. Nombre de chemins indépendants dans le code (boucles, conditions, etc.). Recommandé < 10 en général et < 20 pour une partie complexe

Max depth : niveau maximal d'imbrication des blocs de contrôle (if, for, while, switch, etc.) dans une méthode. Recommandé < 3 (<1.8 pour la moyenne)

Métrique du code – Source Monitor

Profondeur d'imbrication

Indique le niveau d'imbrication des structures de contrôle (boucles, conditions imbriquées). Une profondeur élevée (> 4-5) rend le code difficile à lire et à déboguer. Il est conseillé de refactoriser pour réduire cette valeur.

Nombre de paramètres par fonction/méthode

Un grand nombre de paramètres (> 4-5) peut rendre une fonction difficile à utiliser et à tester. Cela peut indiquer qu'une fonction fait trop de choses et devrait être divisée.

Nombre de points de contrôle

SourceMonitor permet de comparer les métriques à différents points de contrôle (versions, dates). Cela aide à suivre l'évolution de la qualité du code dans le temps, par exemple après une refactorisation ou l'ajout de nouvelles fonctionnalités

Métriques de code

- Aucune métrique ne doit être interprétée isolément.
- Par exemple une complexité cyclomatique élevée peut être acceptable pour une fonction critique et bien testée, mais doit alerter si elle est combinée à un manque de commentaires et une profondeur d'imbrication importante.
- L'objectif est d'utiliser ces indicateurs pour identifier les zones à risque et prioriser les efforts de refactorisation ou de documentation.

Documentation

Documentation

Une forme de contradiction entre écrire une doc et écrire du code commenté

Certains outils proposent de générer la doc à partir du code lui-même

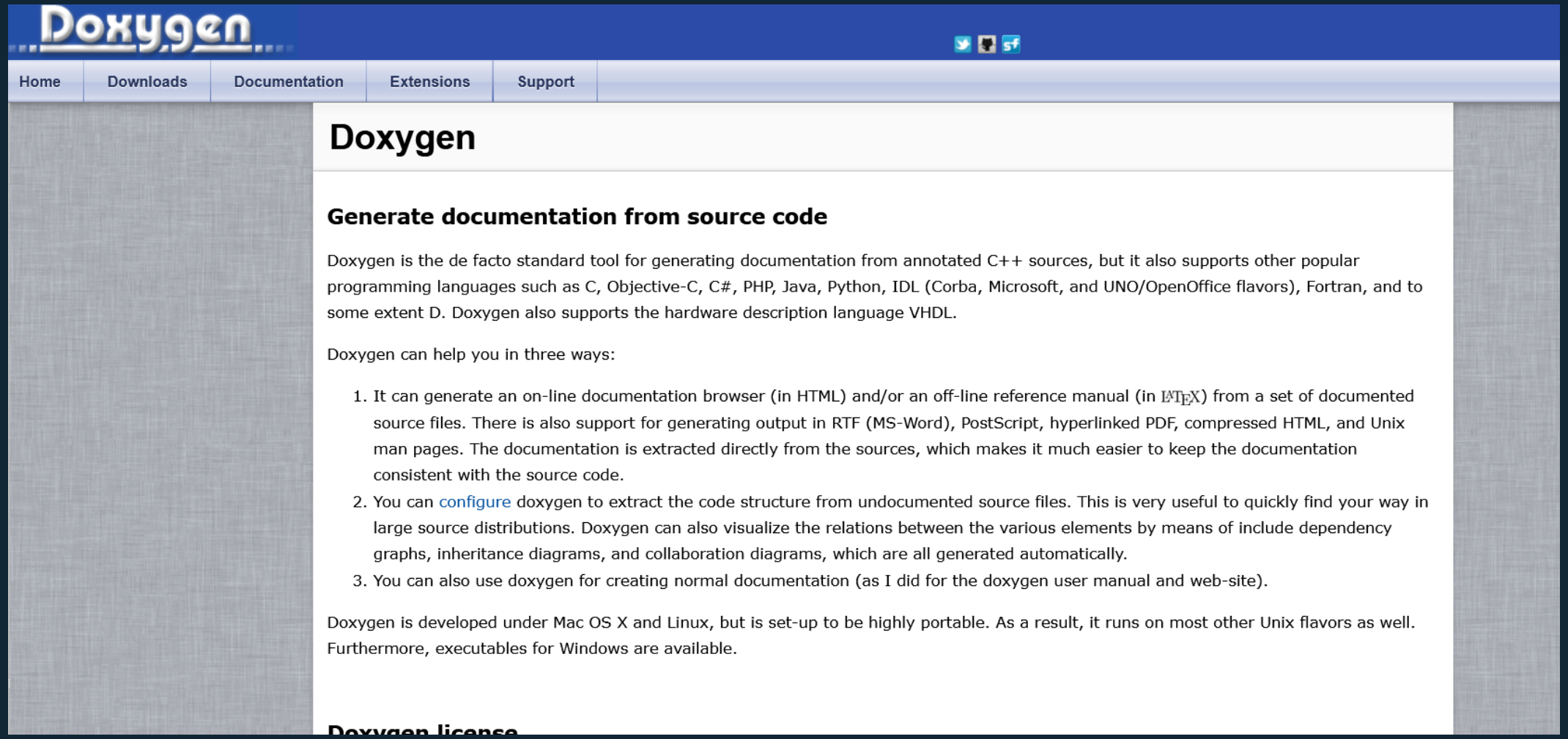
Avantages :

- On ne fait qu'une fois le travail
- Mise à jour de la doc automatique

Inconvénients

- Ne dispense pas de docs complémentaires
- Mettre des commentaires pertinents dans le code

Documentation



https://www.doxygen.nl/download.html

Documentation

Méthode de documentation des entités : placez un bloc de documentation spécial devant la déclaration ou la définition du membre, de la classe ou de l'espace de nom

Style C avec deux *

```
/**
*    ... documentation
*/
```

Style C avec un !

```
/*!
*    ... documentation
*/
```

Style C++ avec trois /

```
///
///    ... documentation
///
```

Style C++ avec un !

```
//!
//!    ... documentation
//!
```

Documentation - Exemple

```
#ifndef CPLAYER_H_
#define CPLAYER_H_

/*!
 * \file CPlayer.h
 * \brief Lecteur de musique de base
 * \author hiko-seijuro
 * \version 0.1
 */
#include <string>
#include <list>

/*! \namespace player
 *
 * espace de nommage regroupant les outils composants
 * un lecteur audio
 */
namespace player
{
  /*! \class CPlayer
   * \brief classe representant le lecteur
   *
   *  La classe gere la lecture d'une liste de morceaux
   */
  class CPlayer
  {
  private:
    std::list<string> m_listSongs; /*!< Liste des morceaux*/
    std::list<string>::iterator m_currentSong; /*!< Morceau courant */
```

Démo : extension VS code

https://marketplace.visualstudio.com/items?itemName=cschlosser.doxdocgen

Peut générer la doc en HTML ou PDF

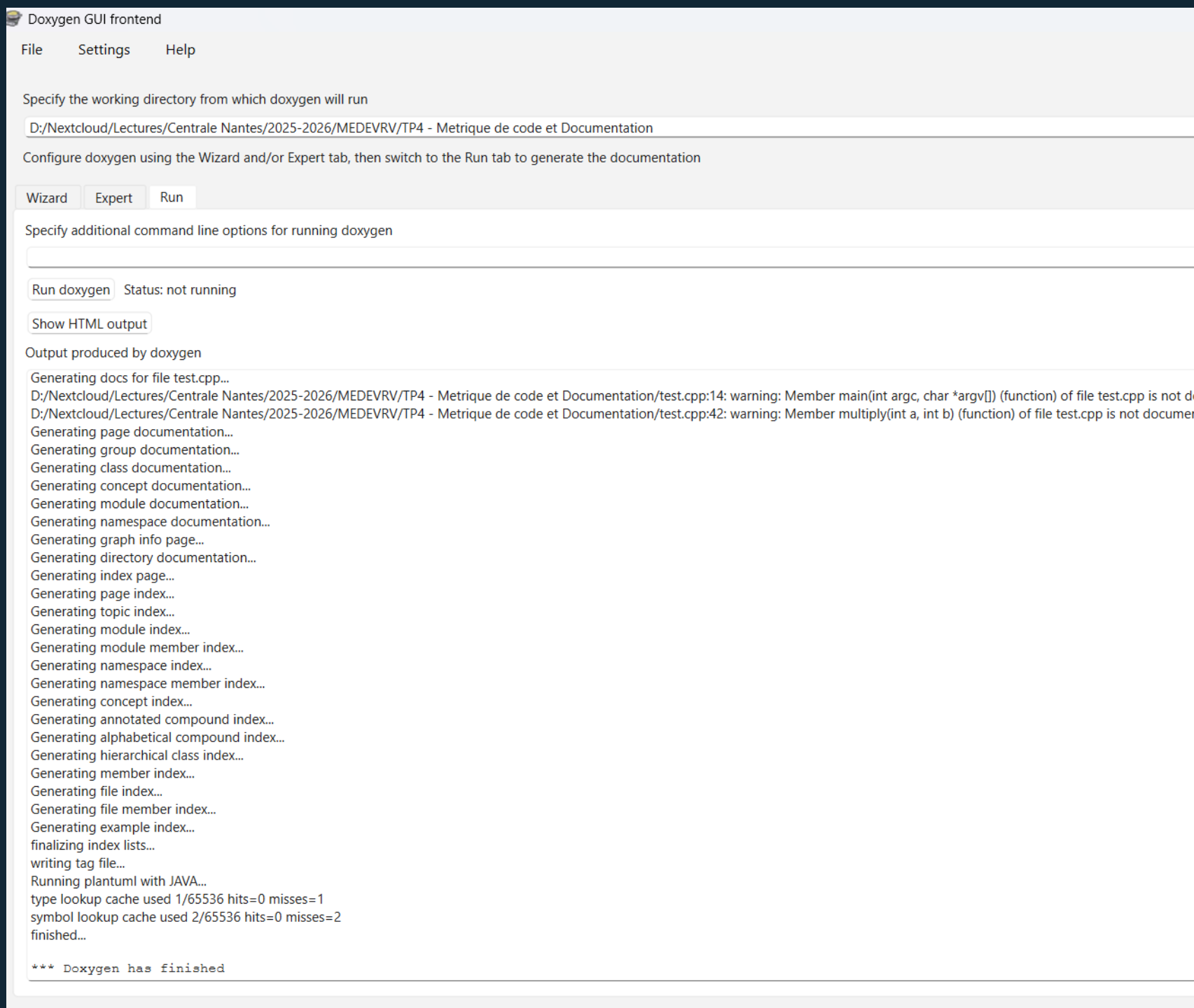
Documentation - Exemple

```
public:
  /*!
   *  \brief Constructeur
   *
   *  Constructeur de la classe CPlayer
   *
   *  \param listSongs : liste initial des morceaux
   */
  CPlayer(std::list<string> listSongs);

  /*!
   *  \brief Destructeur
   *
   *  Destructeur de la classe CPlayer
   */
  virtual ~CPlayer();

public:
  /*!
   *  \brief Ajout d'un morceau
   *
   *  Methode qui permet d'ajouter un morceau a liste de
   *  lecture
   *
   *  \param strSong : le morceau a ajouter
   *  \return true si morceau deja present dans la liste,
   *  false sinon
   */
  bool add(std::string strSong);
```

```
/*!
 *  \brief Morceau suivant
 *
 *  Passage au morceau suivant
 */
void next();

/*!
 *  \brief Morceau precedent
 *
 *  Passage au morceau precedent
 */
void previous();

/*!
 *  \brief Lecture
 *
 *  Lance la lecture de la liste
 */
void play();

/*!
 *  \brief Arret
 *
 *  Arrete la lecture
 */
void stop();
```

Doc

Géneration HTML PDF...



My Project

Main Page Files

- My Project
  - Files
    - File List
      - test.cpp
    - File Members

test.cpp File Reference

Minimal main for Doxygen documentation. More...

```
#include <iostream>
```

Functions

```
int main (int argc, char *argv[])
int add (int a, int b)
```

calcule la somme de deux entiers a et b

```
int multiply (int a, int b)
```

Detailed Description

Minimal main for Doxygen documentation.

Author

your name (you@domain.com)

Version

0.1

Date

2025-11-27

Copyright

Copyright (c) 2025

Documentation

Autres possibilités :

Métrique de code : Visual Studio (Analytics window, pour C#)

Documentation : Sphinx, Javadocs…

Qualité logicielle

v · m	Gestion de la qualité logicielle	[masquer]
Indicateurs de qualité (ISO/CEI 9126)	Capacité fonctionnelle (réponse aux exigences) · Fiabilité · Maintenabilité · Performance · Portabilité · Utilisabilité	
Compréhension et contrôle du code source	Automatisation de test · Commentaires · Documentation · Inspection de produit · Programmation en binôme ou en groupe · Règles de codage · Revue de code	
	Tests	Acceptation · Intégration · Performance · Régression · Unitaire · Utilisateur · Validation
	Métriques	Cohésion · Couplage · Couverture de code · Halstead · Indépendance fonctionnelle · Indice de maintenabilité · Ligne de code · Nombre cyclomatique · Point de fonction
Remaniements	Maintenance · Optimisation de code · Réusinage de code (Règle de trois)	
Principes de programmation	Encapsulation · GRASP · KISS · Loi de Déméter · Masquage de l'information · Ne vous répétez pas (DRY) · Patron de conception · Séparation des préoccupations · YAGNI	
	SOLID	Responsabilité unique · Ouvert/fermé · Substitution de Liskov · Ségrégation des interfaces · Inversion des dépendances
Mauvaises pratiques	**Antipatterns**	Attente active · Grosse boule de boue · Programmation spaghetti (syndrome) · Réinventer la roue
	Code smells	Duplication de code · God object
Voir aussi : Génie logiciel, Software craftsmanship, Dégradation logicielle		