Génie Logiciel

Introduction au génie logiciel

BAPTISTE PESQUET

Support de cours

https://ensc.gitbook.io/genie-logiciel/

Sommaire

- Définition du génie logiciel
- Architecture logicielle
- Principes de conception logicielle
- Production du code source

Définition du génie logiciel

Le génie logiciel

Application de principes d'ingénierie au domaine de la création de logiciels

Mise en œuvre de **méthodes**, de **pratiques** et **d'outils** pour maximiser les chances de réussite du projet

Réponse aux défis posés par la complexification croissante des logiciels et de leur processus de création

« *Software engineering is what happens to programming when you add time and other programmers.* » (Russ Cox)

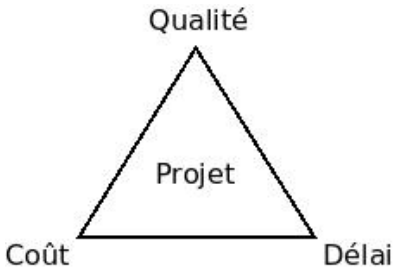
Enjeux du génie logiciel

Adéquation aux besoins du client

Respect des délais de réalisation prévus

Maximisation des performances et de la fiabilité

Facilitation de la maintenance et des évolutions ultérieures



La qualité d'un logiciel

Le logiciel répond-il aux besoins exprimés ?

Le logiciel demande-t-il peu d'efforts pour évoluer aux regards de nouveaux besoins ?

Le logiciel peut-il facilement être transféré d'une plate-forme à une autre ?

... (voir norme ISO 9126)

Dimensions du génie logiciel

Le génie logiciel couvre l'ensemble du **cycle de vie** d'un logiciel :

- Analyse des besoins du client
- Définition de l'architecture du logiciel
- Choix de conception
- Règles et méthodes de production du code source
- Gestion des versions
- Test du logiciel
- Documentation
- Maintenance et évolutions

La (longue) vie d'un projet logiciel

Projets scolaires

- Durée de vie courte
- Aspect « kleenex » (une seule version livrée, puis archive/poubelle)

Projets d'entreprise

- Durée de vie longue (parfois plus de 10 ans)
- Nombreuses évolutions : plusieurs versions livrées et ensuite maintenues
- Phase de maintenance en général beaucoup plus longue que la phase de création initiale
- Changements dans l'équipe en charge du projet

Architecture logicielle

Définition

Architecture : « art de construire des édifices »

Phase pendant laquelle sont faits les **choix structurants** pour une application : langages, technologies, outils, décomposition en sous-parties et définition de leurs interactions, etc

Souvent associée à des **diagrammes** qui décrivent l'application

L'architecture logicielle peut désigner :

- L'activité d'architecture
- Le résultat de cette activité

Parfois confondue avec la **conception logicielle**

Importance

Le logiciel créé doit :

- Répondre aux besoins présents
- Résister aux évolutions futures (probablement nombreuses)

Une architecture mal pensée fera augmenter les coûts de mise au point et de maintenance

Il est parfois utile de repenser l'architecture (**refactorisation**) au moment où le projet évolue

Principes de conception logicielle

Séparation des responsabilités

Separation of concerns

Le logiciel est conçu sous la forme d'un ensemble d'éléments (procédures/fonctions, classes, etc) ayant chacun une responsabilité bien définie

- Augmentation de la modularité
- Maintenance et évolutions facilitées
- Meilleure réutilisation

Principe de responsabilité unique (*Single Responsibility Principle*) : chaque élément a une et une seule responsabilité, ou bien est décomposé en sous-éléments plus atomiques

Réutilisation

Inclusion de briques de bases externes : librairies, modules ou composants

Facteur-clé de succès dans les projets logiciels
- Baisse des coûts
- Augmentation de la qualité (composants déjà testés et éprouvés)

Plateformes de partage disponibles dans la majorité des environnements
- C# : NuGet
- JavaScript : npm
- PHP : composer

Efficacité du modèle ***Open Source*** et des services de partage de code en ligne comme GitHub

Couplage faible, cohésion forte

« Une entité (sous-partie, composant, classe, méthode) est **couplée** à une autre si elle dépend d'elle pour fonctionner »
- Couplage faible => souplesse de mise à jour
- Couplage fort => « code spaghetti »

Cohésion forte : regrouper des éléments ayant des rôles similaires ou gérant la même problématique

« Program to interfaces, not implementations »

```
// Classe1 et classe2 sont couplées

public class Classe1 {
  public Classe2 Classe2
    { get; set; }
}

public class Classe2 { }
```

```
// Découplage : Classe1 utilise une interface sans connaître son implémentation

public class Classe1 {
  public IClasse2 Classe2
    { get; set; }
}

public interface IClasse2 { … }

public class Classe2 : IClasse2 { … }
```


DRY: Don’t Repeat Yourself

Objectif : éviter la redondance et ses effets néfastes :

- Augmentation du volume de code
- Diminution de sa lisibilité
- Risque d'apparition de bogues dûs à des modifications incomplètes

Un peu de redondance est acceptable si son élimination conduit à trop de complexité

Eliminer la redondance de code

```
// Avant
function A() {
  // ...
  // Code dupliqué
  // ...
}
function B() {
  // ...
  // Code dupliqué
  // ...
}
```

```
// Après
function C() {
  // Code auparavant dupliqué
}
function A() {
  // ...
  // Appel à C()
  // ...
}
function B() {
  // ...
  // Appel à C()
  // ...
}
```


KISS: Keep It Simple, Stupid

Complexité => surcoûts de développement puis de maintenance

Il faut privilégier la simplicité !

Ordre des priorités sur un projet logiciel :

1. *Make it work*
2. *Make it right*
3. *Make it fast*

« Simplicity is the ultimate sophistication »

The Evolution of a Software Engineer

The Best Code is No Code At All

YAGNI: You Ain’t Gonna Need It

Ne pas se baser sur d’hypothétiques besoins futurs (passage à l’échelle, nouveaux utilisateurs, etc) pour faire les choix du présent

Sinon, risque d’***over-engineering*** : application inutilement complexe par rapport aux besoins réels

Commencer au plus simple (principe KISS), puis **refactoriser** en fonction des nouveaux besoins

Production du code source

Importance du code source

Code source = cœur d'un projet logiciel

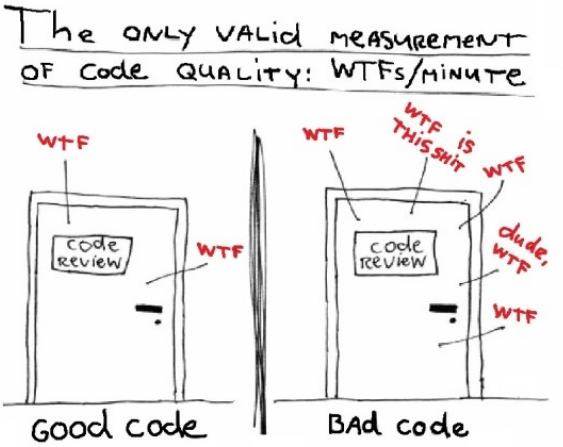
Un développeur passe en moyenne beaucoup plus de temps à lire qu'à écrire du code

=> La production de code source doit obéir à certaines **règles** afin d'obtenir une **qualité** maximale

La qualité du code source

Facteurs de qualité :

- Utilisation *idiomatique* du langage de programmation
- Respect des principes de conception (DRY, KISS, etc)
- Homogénéité
  - Nommage
  - Formatage
- Commentaires



Nommage

Adoption d'une **convention** pour l'ensemble du projet

Exemple : *camelCase*

```
class UneNouvelleClasse {
  private int unAttribut;
  private float unAutreAttribut;
  public void UneMethode(int monParam1, int monParam2) {
    // ...
  }
  public void UneAutreMethode(string encoreUnParametre) {
    // ...
  }
}
```

25 sur 34

Formatage

Les éditeurs et IDE offrent des fonctionnalités de formatage automatique à utiliser sans modération

Ce formatage est **paramétrable**

- Utilisation de tabulations ou espaces pour indenter
- Nombre d'espaces pour l'indentation (souvent 2 ou 4)
- Placement des accolades ouvrantes et fermantes
- …

26 sur 34

Commentaires

Commentaires = forme de documentation du projet

Utile pour les parties **complexes** ou **essentielles**

Eviter de paraphraser le code

Bonne pratique : commenter tous les en-têtes de classes et de fonctions/méthodes

```
/// <summary>
/// Description de la classe
/// </summary>
public class MaClasse { … }
```

```
/// <summary>
/// Description de la méthode
/// </summary>
/// <param name="s"> Description du paramètre s
/// </param>
public void MaMethode(string s) { … }
```


Versionnage

Objectifs :

- Assurer la pérennité du code source d'un logiciel.
- Faciliter le travail collaboratif.
- Fournir une gestion de l'historique du logiciel.

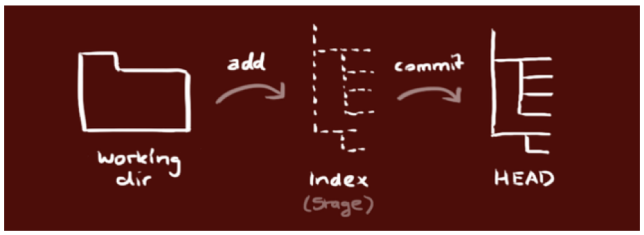
Solution : utiliser des outils dédiés

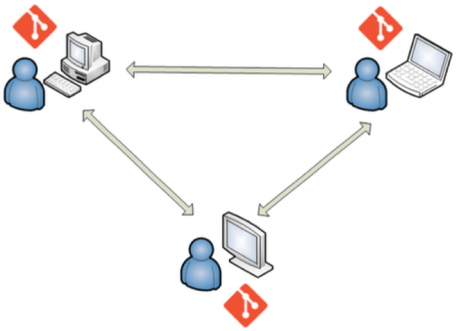
- Logiciels de versionnage : Git, Mercurial, SVN…
- Services de partage et collaboration en ligne : GitHub, GitLab, BitBucket…

Git

Logiciel de versionnage *décentralisé* créé pour gérer les versions du noyau Linux.

Standard actuel en entreprise.

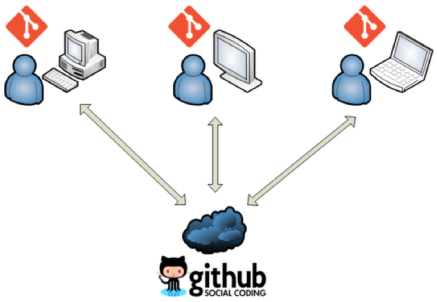




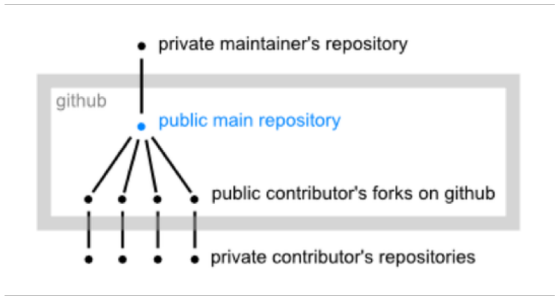
29 sur 34

GitHub

Plate-forme web d'hébergement et de partage de code via des dépôts Git. Deux modèles de collaboration possibles :



Dépôt partagé



Pull Requests

30 sur 34

Documentation

Importance de la documentation

En plus de son code, un ou plusieurs documents doivent accompagner un projet logiciel.

Tout comme le code, ces documents doivent être maintenus régulièrement pour ne pas devenir obsolète et donc inutile.

Différentes formes possibles : documents bureautique, web, Wiki, etc.

Documentation technique

Décrit comment *fonctionne* le logiciel.

Destinée aux membres de l'équipe de développement/maintenance.

Langage technique.

Contenu typique :

- Modélisation des données (diagrammes MCD et/ou UML)
- Découpage en composants
- Environnement requis : OS, langages, outils, dépendances
- Processus de génération et de déploiement
- Processus de test
- …

33 sur 34

Documentation utilisateur

Décrit comment *utiliser* le logiciel.

Destinée à ses utilisateurs.

Langage non technique.

Formes possibles (combinables) :

- Manuel d'utilisation
- Guide de référence
- Tutoriel
- FAQ
- Aide contextuelle
- …

34 sur 34