Bug Predictor

Outil de Prédiction de Fichiers à Risque

Boutadghart Imran
Fazaz Houssam

7 décembre 2025

https://github.com/imranboutadghart/software_engineering

Table des matières

1 Introduction Générale

Dans le domaine du développement logiciel moderne, la qualité du code et la prévention des bugs constituent des enjeux majeurs pour garantir la fiabilité et la maintenabilité des applications. Les bugs non détectés peuvent entraîner des coûts importants, tant en termes de temps de développement que de qualité du produit final.

1.1 Problématique

Les équipes de développement font face à plusieurs défis concernant la qualité du code :
— Identification tardive des zones à risque dans le code
— Difficulté à prioriser les efforts de refactoring et de tests
— Absence d'indicateurs préventifs sur la qualité du code
— Manque d'outils automatisés pour l'analyse prédictive des bugs

1.2 Objectif du Projet

Ce projet vise à développer **Bug Predictor**, un outil automatisé capable d'identifier les fichiers à risque dans un projet logiciel en combinant l'analyse de l'historique Git et l'analyse de la complexité du code. L'objectif principal est de fournir aux développeurs un tableau de bord interactif permettant de visualiser les fichiers les plus susceptibles de contenir des bugs.

1.3 Contributions

Les principales contributions de ce projet sont :
— Development d'un moteur d'analyse de risque combinant plusieurs métriques
— Création d'un service d'analyse de l'historique Git pour identifier les fichiers fréquemment modifiés
— Implémentation d'un calculateur de complexité cyclomatique
— Développement d'une interface web moderne pour visualiser les résultats

2 Contexte Général du Projet

2.1 Contexte Scientifique

La prédiction de bugs est un domaine de recherche actif dans le génie logiciel. Plusieurs études ont démontré que :

— Les fichiers fréquemment modifiés sont plus susceptibles de contenir des bugs
— La complexité cyclomatique élevée est corrélée avec un taux de bugs plus important
— L'historique des modifications (churn) peut servir d'indicateur de qualité du code

2.2 Technologies Utilisées

Le projet **Bug Predictor** s'appuie sur un ensemble de technologies modernes :

Catégorie	**Technologie**	**Utilisation**
Backend	Python 3.x	Langage principal de développement
Framework Web	FastAPI	API REST et serveur web asynchrone
Analyse Git	GitPython	Extraction des métriques d'historique
Analyse Code	Radon	Calcul de la complexité cyclomatique
Frontend	HTML/TailwindCSS	Interface utilisateur moderne
Templating	Jinja2	Génération dynamique des pages HTML

Table 1 – Stack technologique du projet

2.3 Fonctionnalités Principales

Le système offre les fonctionnalités suivantes :

1. **Analyse de l'Historique Git** : Identification des fichiers fréquemment impliqués dans les corrections de bugs
2. **Analyse de Complexité** : Calcul de la complexité cyclomatique pour évaluer la maintenabilité
3. **Calcul du Score de Risque** : Combinaison des métriques de churn et de complexité
4. **Dashboard Web** : Interface interactive pour visualiser les résultats d'analyse

3 Analyse et Conception

3.1 Analyse des Besoins

Le backlog produit a identifié les user stories suivantes :

ID	Description	Priorité
US-01	Analyser l'historique Git pour identifier les fichiers instables	Haute
US-02	Calculer la complexité cyclomatique des fichiers	Haute
US-03	Calculer un score de risque unique par fichier	Haute
US-04	Fournir une interface web pour visualiser les fichiers à risque	Moyenne
US-05	Exporter les résultats d'analyse	Basse

TABLE 2 – Backlog produit - User Stories

3.2 Architecture Logique

L'architecture du système est organisée en plusieurs couches :
— **Couche Présentation** : Interface web utilisant FastAPI et Jinja2
— **Couche Métier (Services)** :
  — GitAnalysisService : Analyse de l'historique Git
  — ComplexityService : Calcul de la complexité
  — RiskScorer : Calcul du score de risque
— **Couche Modèle** : Structures de données (FileAnalysis, ProjectAnalysis)
Le système suit une architecture MVC (Model-View-Controller) adaptée au contexte FastAPI.

3.3 Architecture Physique

L'application est déployée en tant qu'application web monolithique :
— **Serveur Web** : FastAPI avec Uvicorn (serveur ASGI)
— **Port d'écoute** : 8000 (par défaut)
— **Interface** : http ://localhost :8000

3.4 Conception Détaillée

3.4.1 Modèles de Données

Le système définit deux modèles principaux :

```
class FileAnalysis:
    filename: str
    churn_count: int
    complexity_score: float
    risk_score: float

class ProjectAnalysis:
```

```
    project_path: str
    total_files: int
    files: List[FileAnalysis]
```

3.4.2 Services

GitAnalysisService : Extrait les métriques de churn à partir de l'historique Git du projet.

ComplexityService : Utilise Radon pour calculer la complexité cyclomatique de chaque fichier.

RiskScorer : Combine les métriques de churn et de complexité pour générer un score de risque unifié.

4 Approche Proposée

4.1 Méthodologie de Développement

Le projet a adopté une approche agile avec les éléments suivants :
— **Méthodologie** : Scrum adapté pour un mini-projet
— **Sprints** : Organisation en sprints courts
— **Documentation** : Product Backlog et Sprint Backlog
— **Livraison** : Approche itérative et incrémentale

4.2 Algorithme de Calcul du Risque

L'approche proposée pour calculer le score de risque combine deux métriques principales :

1. **Churn Count** : Nombre de modifications du fichier dans l'historique Git
2. **Complexity Score** : Complexité cyclomatique moyenne du fichier

La formule de calcul du risque est une combinaison pondérée de ces deux métriques :

$$RiskScore = f(ChurnCount, ComplexityScore) \tag{1}$$

où f est une fonction qui normalise et combine les deux métriques pour produire un score de risque unifié.

4.3 Stratégie d'Analyse

Le processus d'analyse suit les étapes suivantes :

1. Réception du chemin du projet à analyser
2. Extraction des métriques de churn via GitPython
3. Parcours récursif du répertoire du projet
4. Pour chaque fichier :
   — Récupération du churn count
   — Calcul de la complexité cyclomatique
   — Calcul du score de risque
5. Tri des fichiers par score de risque décroissant
6. Présentation des résultats dans l'interface web

4.4 Filtrage Intelligent

Le système exclut automatiquement certains répertoires de l'analyse :
— `.git` : Dossier de contrôle de version
— `__pycache__` : Cache Python
— `venv` / `.venv` : Environnements virtuels
— `node_modules` : Dépendances Node.js
— `.idea` / `.vscode` : Fichiers IDE
Cette approche garantit que seuls les fichiers sources pertinents sont analysés.

5 Mise en Œuvre

5.1 Structure du Projet

L'organisation des fichiers du projet est la suivante :

```
BugPredictor/
 app/
    __init__.py
    main.py              # Point d'entrée FastAPI
    models/
       analysis_model.py
    services/
       git_analysis.py
       complexity.py
       risk.py
    templates/
        index.html
 docs/
    design/
       architecture.md
       class_diagram.md
    scrum/
        product_backlog.md
        sprint_backlog.md
 tests/
 requirements.txt
 README.md
```

5.2 Implémentation des Services

5.2.1 Service d'Analyse Git

Le `GitAnalysisService` extrait les métriques de churn en analysant l'historique des commits Git. Chaque fichier modifié est comptabilisé pour déterminer sa fréquence de changement.

5.2.2 Service de Complexité

Le `ComplexityService` utilise la bibliothèque Radon pour calculer la complexité cyclomatique de McCabe. Cette métrique mesure le nombre de chemins indépendants dans le code.

5.2.3 Calculateur de Risque

Le `RiskScorer` combine les deux métriques précédentes pour produire un score de risque. Les fichiers avec un churn élevé et une complexité importante reçoivent les scores les plus élevés.

5.3 Interface Web

5.3.1 Application FastAPI

L'application principale (`main.py`) expose deux endpoints :
— `GET /` : Affiche la page d'accueil avec le formulaire d'analyse
— `POST /analyze` : Lance l'analyse d'un projet et affiche les résultats

5.3.2 Template HTML

Le template utilise Jinja2 pour générer dynamiquement la page HTML avec :
— Formulaire de saisie du chemin du projet
— Tableau des résultats triés par score de risque
— Affichage des métriques pour chaque fichier

5.4 Gestion des Dépendances

Le fichier `requirements.txt` spécifie les dépendances du projet :

```
fastapi
uvicorn
gitpython
radon
jinja2
python-multipart
```

5.5 Installation et Exécution

Pour installer et exécuter le projet :

1. Installation des dépendances : `pip install -r requirements.txt`
2. Lancement du serveur : `uvicorn app.main:app -reload`
3. Accès à l'interface : `http://localhost:8000`

5.6 Tests

Un plan de tests a été élaboré pour valider le fonctionnement du système :
— Tests unitaires des services (GitAnalysisService, ComplexityService, RiskScorer)
— Tests d'intégration de l'API FastAPI
— Tests de validation de l'interface utilisateur

6 Conclusion Générale

6.1 Récapitulatif

Ce projet a permis de développer **Bug Predictor**, un outil d'analyse prédictive des bugs pour les projets logiciels. L'outil combine avec succès l'analyse de l'historique Git et l'analyse de la complexité du code pour identifier les fichiers à risque.

6.2 Objectifs Atteints

Les objectifs principaux du projet ont été réalisés :
Implémentation d'un moteur d'analyse de l'historique Git
Calcul de la complexité cyclomatique avec Radon
Création d'un algorithme de scoring de risque
Développement d'une interface web interactive avec FastAPI
Documentation complète du projet (architecture, backlog, tests)

6.3 Apport Pédagogique

Ce mini-projet a permis de mettre en pratique plusieurs concepts du génie logiciel :
— **Méthodologie Agile** : Utilisation de Scrum avec backlog et sprints
— **Architecture Logicielle** : Conception d'une architecture en couches
— **Développement Web** : Utilisation de FastAPI et des templates Jinja2
— **Analyse de Code** : Application de métriques logicielles (complexité, churn)
— **Intégration Git** : Exploitation de l'historique de version

6.4 Limites et Perspectives

6.4.1 Limites Actuelles

— L'algorithme de scoring pourrait être affiné avec des pondérations ajustables
— Support limité aux projets Git uniquement
— Absence de fonctionnalité d'export des résultats (US-05 non implémentée)
— Pas de persistance des analyses effectuées

6.4.2 Perspectives d'Évolution

Plusieurs améliorations pourraient être apportées au projet :

1. **Machine Learning** : Entraîner un modèle de prédiction basé sur des données historiques réelles de bugs
2. **Métriques Supplémentaires** :
   — Taux de couverture de tests
   — Nombre de dépendances
   — Taille des fichiers (lignes de code)
3. **Visualisation Avancée** :
   — Graphiques de tendance
   — Cartes thermiques du code
   — Évolution du risque dans le temps

4. **Intégration CI/CD** : Intégration dans les pipelines de développement
5. **Support Multi-VCS** : Extension à SVN, Mercurial, etc.
6. **Export et Reporting** : Génération de rapports PDF/Excel
7. **API REST Complète** : Endpoints pour une utilisation programmatique

6.5 Conclusion Finale

Le projet **Bug Predictor** démontre qu'il est possible de créer des outils d'analyse prédictive efficaces en combinant des métriques logicielles simples mais pertinentes. L'outil développé peut aider les équipes de développement à identifier proactivement les zones à risque dans leur code et à prioriser leurs efforts de refactoring et de tests.

Ce projet illustre également l'importance d'une approche méthodique dans le développement logiciel, depuis l'analyse des besoins jusqu'à la mise en œuvre, en passant par une conception rigoureuse.

L'approche modulaire adoptée facilite l'évolution future du système et l'ajout de nouvelles fonctionnalités. Bug Predictor constitue ainsi une base solide pour un outil plus avancé de prédiction et de prévention des bugs dans les projets logiciels.