Guide Programmeur – TFDefect GitHub Action

Introduction

Ce guide s'adresse aux développeurs impliqués dans l'amélioration, l'extension ou la maintenance de l'application *TFDefect GitHub Action*. Il vise à fournir une documentation complète et approfondie sur la structure, le fonctionnement et les meilleures pratiques associées au développement de cette action GitHub. Ce document offre une vue détaillée de l'organisation du projet, présente le workflow CI/CD utilisé, décrit précisément les interactions entre les composants principaux, et énumère les bonnes pratiques techniques à suivre pour garantir un code clair, maintenable, testable et évolutif sur le long terme.

Configuration de l'environnement

Ce projet nécessite des prérequis spécifiques et une configuration particulière pour fonctionner correctement. Pour plus de détails sur :

- L'installation des dépendances
- La configuration des variables d'environnement
- La préparation de l'environnement de développement
- Les outils recommandés pour le développement

Veuillez consultez le README.md.

Organisation du Projet

Structure du projet

La structure suivante facilite la navigation dans le projet et reflète les principes fondamentaux de la *Clean Architecture*, en assurant une séparation claire des responsabilités logiques :

Dossiers et fichiers principaux

- `.github/`
  - `workflows/` : Contient la définition du pipeline CI/CD via GitHub Actions (notamment `ci.yml`).
  - `ISSUE_TEMPLATE/` : Fournit des modèles standardisés pour la création d'issues (bugs, documentation, fonctionnalités).
  - `PULL_REQUEST_TEMPLATE.md` : Sert de guide lors de la soumission d'une pull request.
- `app/`
  - `action_runner.py` : Point d'entrée principal, orchestre l'ensemble du processus d'analyse.
  - `config.py` : Centralise tous les paramètres et chemins configurables.
- `core/`
  - `use_cases/` : Contient les cas d'usage métiers spécifiques à l'action.
    - `detect_tf_changes.py`, `analyze_tf_code.py`, `feature_vector_builder.py`, `report_generator.py`
  - `parsers/` : Contient la logique d'extraction et de traitement des métriques Terraform.
    - `terraform_parser.py`, `metrics_extractor_factory.py`, `base_metrics_extractor.py`, `delta_metrics_extractor.py`, `process_metrics_extractor.py`, `process_metrics_calculation.py` et `contribution_builder.py`.
- `infrastructure/`
  - `git/` : Gère les interactions avec le dépôt Git via des outils comme PyDriller.
    - Exemples : `git_adapter.py`, `git_changes.py`
  - `ml/` : Contient les modèles de prédiction ML utilisés pour détecter les bogues.
    - `base_model.py`, `dummy_model.py`, `model_factory.py`, `defect_history_manager.py`
- `templates/` : Fichiers HTML pour générer des rapports lisibles et interactifs.
- `libs/` : Librairies externes, par exemple `terraform_metrics-1.0.jar` (outil d'analyse Java).
- `tests/`
  - `unit/` : Tests unitaires.
  - `integration/` : Tests d'intégration complets.
- `data/` : Contient des fichiers Terraform utilisés pour la validation et les tests.
- `doc/` : Diagrammes d'architecture et documentation technique.

- `out/` : Dossier de sortie pour les résultats générés : rapports HTML, fichiers JSON, etc.
- `utils/` : Fonctions utilitaires partagées, réutilisables dans différents modules.

Fichiers critiques

Fichier	Rôle
`action_runner.py`	Exécution principale de l'action GitHub
`detect_tf_changes.py`	Détection de blocs Terraform modifiés dans un commit
`feature_vector_builder.py`	Construction des vecteurs utilisés par le modèle ML
`dummy_model.py`	Modèle simple pour tests et démos
`report_generator.py`	Génère un rapport complet en HTML
`ci.yml`	Configuration du pipeline GitHub Actions

Architecture du projet

Pour une compréhension approfondie de l'architecture, veuillez consulter le document d'architecture.

Ce document détaille:

- La séparation des couches (core, infrastructure, application)
- Les interactions entre les modules
- Les classes et leurs responsabilités
- Les technologies utilisées

Utilisation des fonctionnalités principales

Analyse des métriques

Pour l'analyse des métriques statiques à partir du code Terraform:

```
python app/action_runner.py --extractor codemetrics
```

Pour analyser l'évolution des métriques avant et après des commits pour un même bloc Terraform:

```
python app/action_runner.py --extractor delta
```

Pour analyser les métriques liées au processus de développement:

```
python app/action_runner.py --extractor process
```

Exécution d'une prédiction

Pour exécuter une prédiction utilisant un modèle de Machine Learning:

```
python app/action_runner.py --model dummy
```

Affichage de l'historique des défauts prédits

Pour consulter l'historiques des défauts prédits lors des analyses effectuées:

```
python app/action_runner.py --show-history
```

Workflow CI/CD avec GitHub Actions

Le pipeline CI/CD décrit dans `.github/workflows/ci.yml` automatise les étapes suivantes à chaque push ou pull request sur la branche principale ou de fonctionnalité :

1. **Clonage du dépôt** avec ses sous-modules si présent
2. **Installation de Python**, configuration de l'environnement virtuel et des dépendances
3. **Installation conditionnelle de Java** (nécessaire pour TerraMetrics)
4. **Exécution automatique de TFDefect** avec les paramètres par défaut (modèle `dummy`, chemin des fichiers définis dans `config.py`)
5. **Vérification et sauvegarde des artefacts** (rapports HTML, historique des défauts JSON)
6. **Possibilité d'envoi de notifications** ou publication d'un commentaire sur la pull request

Ce workflow est conçu pour être facilement personnalisable, modulaire et exécutable localement si besoin (en simulant les étapes via un script).

Tests et Assurance Qualité

Le projet TFDefectGA suit une approche rigoureuse en matière de tests pour garantir la fiabilité et la maintenabilité du code. Deux types de tests sont mis en place :

Tests unitaires

Les tests unitaires vérifient le comportement de composants individuels de manière isolée :

- **Test des parsers** (`test_terraform_parser.py`, `test_code_metrics_extractor.py`) : Vérifient la capacité du système à analyser correctement les fichiers Terraform et à extraire les métriques pertinentes.
- **Test des extracteurs** (`test_delta_metrics_extractor.py`, `test_process_metrics.py`) : S'assurent que les différents extracteurs de métriques génèrent les données attendues.
- **Test des cas d'utilisation** (`test_analyze_tf_code.py`, `test_detect_tf_changes.py`) : Valident le fonctionnement des cas d'utilisation principaux.
- **Test des factories** (`test_metrics_extractor_factory.py`) : Garantissent que les patterns Factory fonctionnent correctement.

Tests d'intégration

Les tests d'intégration vérifient le fonctionnement coordonné de plusieurs composants :

- **Test du flow complet de prédiction** (`test_integration_prediction.py`) : Simule une exécution complète du processus de prédiction, depuis l'extraction des métriques jusqu'à la mise à jour de l'historique des défauts.
- **Test de construction des vecteurs** (`test_integration_vector_builder.py`) : S'assure que le `FeatureVectorBuilder` peut fusionner correctement les différentes métriques pour alimenter le modèle prédictif.

Exécution des tests

Pour exécuter les tests, utilisez les commandes suivantes depuis la racine du projet :

```
# Exécuter tous les tests
pytest

# Exécuter uniquement les tests unitaires
pytest tests/unit/

# Exécuter uniquement les tests d'intégration
pytest tests/integration/

# Exécuter un fichier de test spécifique
pytest tests/unit/test_terraform_parser.py
```

Bonnes Pratiques Techniques

Afin d'assurer la stabilité, la qualité et la maintenabilité du projet, les pratiques suivantes sont fortement recommandées :

Architecture et conception

- **Clean Architecture** : Séparation stricte des responsabilités avec des couches distinctes (présentation, cas d'utilisation, domaine, infrastructure). Cette approche facilite les tests unitaires et limite la propagation des changements.
- **Patron de conception Factory** : Utilisé pour la création d'objets sans spécifier leur classe concrète, permettant l'extension du système sans modifier le code existant. Exemples d'implémentation dans le projet : `MetricsExtractorFactory` et `ModelFactory`.
- **Modularité** : Chaque module accomplit une tâche précise avec une interface claire, favorisant la réutilisabilité et la maintenabilité.

Qualité du code

- **Typage statique** : Utilisation du module `typing` pour annoter les paramètres et valeurs de retour des fonctions, facilitant l'autocomplétion et la détection d'erreurs.
- **Documentation** : Chaque classe, méthode et fonction doit être documentée avec des docstrings incluant paramètres, valeurs de retour et exemples si nécessaire.
- **Gestion des erreurs** : Utilisation d'exceptions spécifiques et capture au niveau approprié pour une bonne gestion des erreurs.
- **Logging** : Utilisation du module `logging` configuré via `logger_utils.py` pour faciliter le débogage.

Développement

- **Tests** : Implémentation de tests unitaires pour chaque classe/fonction ainsi que des tests d'intégration pour valider les interactions entre composants.
- **Gestion des dépendances** : Spécification des versions des dépendances dans `requirements.txt`

Versionning et collaboration

- **Commits atomiques** : Chaque commit doit représenter une seule fonctionnalité ou correction
- **Messages de commit explicites** : Format "type: description" (ex: "feat: ajoute support pour modèle XYZ")
- **Pull Requests documentées** : Utilisation du template fourni dans `.github/PULL_REQUEST_TEMPLATE.md`
- **Revue de code** : Obligatoire avant fusion dans les branches principales

Gestion des Configurations

Tous les paramètres critiques sont regroupés dans `app/config.py`. Cela inclut :

- Le chemin vers le fichier `.jar` de TerraMetrics
- Les emplacements de sortie des rapports et fichiers JSON
- Le modèle à utiliser par défaut (ex : `dummy`)
- Le chemin du dépôt local analysé
- Les options activables par développeur pour ajuster le comportement de l'action

Cette centralisation évite la duplication de logique de configuration dans plusieurs fichiers.

Conclusion

Cette documentation fournit une vision d'ensemble rigoureuse et concrète du projet TFDefectGA. Elle permet aux développeurs de comprendre en profondeur le fonctionnement interne de l'application, de participer efficacement à son évolution, et de contribuer à sa qualité logicielle.

En adoptant une architecture modulaire et testable, le projet favorise la contribution ouverte, l'évolutivité des fonctionnalités, et une gestion efficace de la dette technique.