Fiabilité Logicielle

Méthodes formelles (généralités)

Arnaud Labourel

2025-26

amU Faculté des sciences Aix Marseille Université

Contenu

- Linting - Analyse de code source
- Méthodes formelles
  - Model-checking
  - Analyse statique
  - Synthèse certifiée de programmes
  - Preuve de programmes - vérification déductive (S. Bardin)

Linting

- analyse du code source
- s'appuie sur une représentation syntaxique du source du programme
- détecte des mauvaises pratiques de programmation (style, constructions, ...)

Pas exactement une méthode formelle, mais la mauvaise pratique ...

- peut provenir d'un vrai défaut
- nuit à la lisibilité du code et à la maintenabilité

Abelson & Sussman, Structure and Interpretation of Computer Programs

Programs must be written for people to read, and only incidentally for machines to execute

Linting : un exemple

```
try  {
      // some code
    } catch (Exception e) \{
      e.printStackTrace();
    }
```

Mauvaise pratique : gestion d'exception inappropriée

- `throw new RuntimeException(e);`
- `Log.log(e)`

L'ancêtre lint

Lint apparait en 1979 dans UNIX

Lint alerte pour des Programmes écrits en C sur des constructions suspectes ou non-portables telles que :

- l'utilisation avant la définition pour les variables
- la définition de variables ou de fonctions non-utilisés
- le renforcement du contrôle des types
- l'existence de code mort
- le fait d'ignorer les valeurs de retour des fonctions

Ces "erreurs" n'en sont pas du point de vue du compilateur.

De nos jours ...

Certains défauts sont détectés ...

- lors de la compilation (warnings, voire erreurs)
- par l'IDE (Intellij inspection)
- lors de l'étape d'analyse de code en CI/CD (Continuous Integration/Continuous Deployment).

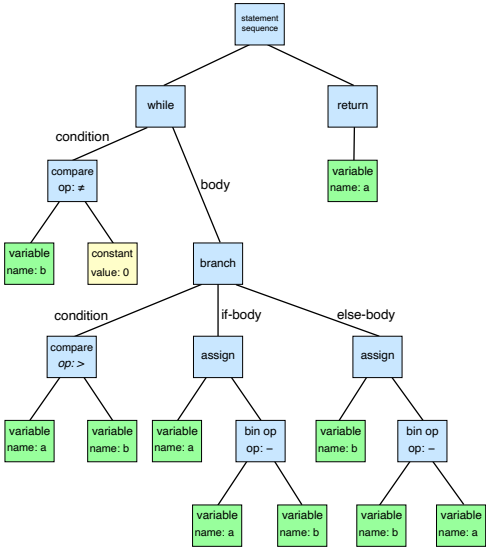
Cruciaux pour les langages typés dynamiquement (python, Javascript)

Les linters s'appuient sur une représentation du programme sous la forme d'un arbre de syntaxe abstrait.

Arbre de syntaxe abstrait (AST)

- pour un même langage, plusieurs types d'arbres et plusieurs syntaxes possibles
- différents de l'arbre syntaxique

```
while b != 0:
    if a > b:
        a = a - b
    else:
        b = b - a
return a
```



Linting et AST

pour le linting, l'AST permet de

- s'abstraire de la syntaxe textuelle.
- de décrire des conditions des défauts comme des conditions complexes liées à la structure syntaxique du programme

```
fn main() {
    1 + (2 * 3)
}
```

```
<Crate>
  <Item>
    <VisItem pub="false">
      <Function>
        <Identifier name="main"/>
        <BlockExpression>
          <Statements>
            <ExpArithmOrLogic>
              <ExpLit Exp="1"/>
              <Op Symbol="+"/>
              <ExpArithmOrLogic>
                <ExpLit Exp="2"/>
                <Op Symbol="*"/>
                <ExpLit Exp="3"/>
              </ExpArithmOrLogic>
            </ExpArithmOrLogic>
          </Statements>
        </BlockExpression>
      </Function>
    </VisItem>
  </Item>
</Crate>
```

Linting et AST (II)

Quand l'AST est descriptible XML, la description des conditions de défaut peut être donnée par une expression XPath.

```
package                          + PackageDeclaration
org.codecop.myapp.db;            | + Name :org.codecop.myapp.db
import                           + ImportDeclaration
java.sql.Connection;             | + Name :java.sql.Connection
public class DBSearch { ...      + TypeDeclaration
}                                 + ClassOrInterfaceDeclaration :DBSearch
                                   + ClassOrInterfaceBody
                                   + ...
```

```
//ImportDeclaration[starts-with(Name/@Image, 'java.sql.')]
```

Possibilité de définir ces propres règles de style de programmation

Linters

Exemple : Lint, FindBugs, PMD, Checkstyle, JLint, SonarLint

Exemples de règles en PMD :

`equals(Object o)` n'est pas redéfini pour toute `Classe`

```
//ClassDeclaration//MethodDeclarator
  [@Image = 'equals']
  [count(FormalParameters/*)=1]
  [not ( FormalParameters//Type/Name[
          @Image='Object' or @Image='java.lang.Object' ] ) ]
```

Exemples d'erreurs classiques (en Java)

- variables ou méthodes non utilisées $\Rightarrow$ code inutile et confusion possible
- gestion des exceptions trop large (attraper `Exception` ou `Throwable`) $\Rightarrow$ masque des erreurs spécifiques
- plusieurs assertions dans un test unitaire $\Rightarrow$ rend difficile l'identification de la cause d'un échec
- définition de champs publics $\Rightarrow$ violation de l'encapsulation
- utilisation d'un objet modifiable non cloné dans un constructeur ou une méthode $\Rightarrow$ risque de modification involontaire (violation encapsulation)

Exemples d'erreurs classiques (en Java)

- oubli de fermeture de ressources (fichiers, connexions réseau, bases de données, ...) $\Rightarrow$ favoriser le *try-with-resources*
- expression conditionnelle toujours vraie ou toujours fausse $\Rightarrow$ code mort
- Définition de `equals(Class o)` au lieu de `equals(Object o)` $\Rightarrow$ confusion possible
- Définition de `Equals` sans redéfinir `hashCode` $\Rightarrow$ incohérence dans les collections basées sur des tables de hachage
- Gestion d'erreur utilisant `printStackTrace()` $\Rightarrow$ perte d'information et difficulté de maintenance (utiliser un logger ou relancer l'exception)
- Utilisation de == pour comparer des chaînes de caractères ou des types d'emballage comme `Integer` $\Rightarrow$ comparer avec `equals()` -non-respect des conventions de nommage (classes, méthodes, variables, ...) $\Rightarrow$ code moins lisible
- classes non placées dans un `package` $\Rightarrow$ risque de conflits de noms et difficulté de gestion du code