VALIDATION & VERIFICATION

STATIC TESTING

UNIVERSITY OF RENNES, ISTIC & ESIR, 2024-2025

BENOIT COMBEMALE
FULL PROFESSOR, UNIVERSITY OF RENNES, FRANCE

HTTP://COMBEMALE.FR
BENOIT.COMBEMALE@IRISA.FR
@BCOMBEMALE

Test statique

- Ne requiert pas l'exécution du logiciel sous-test sur des données réelles
- Plusieurs approches
  - inspection de code (lisibilité du code, spécifications complètes...)
  - mesures statiques (couplage, nombre d'imbrications...)
  - etc.

Plan

1. Revue de code
2. Règles de codage
3. Vérifications automatiques

Plan

1. Revue de code
2. Règles de codage
3. Vérifications automatiques

Inspection de code

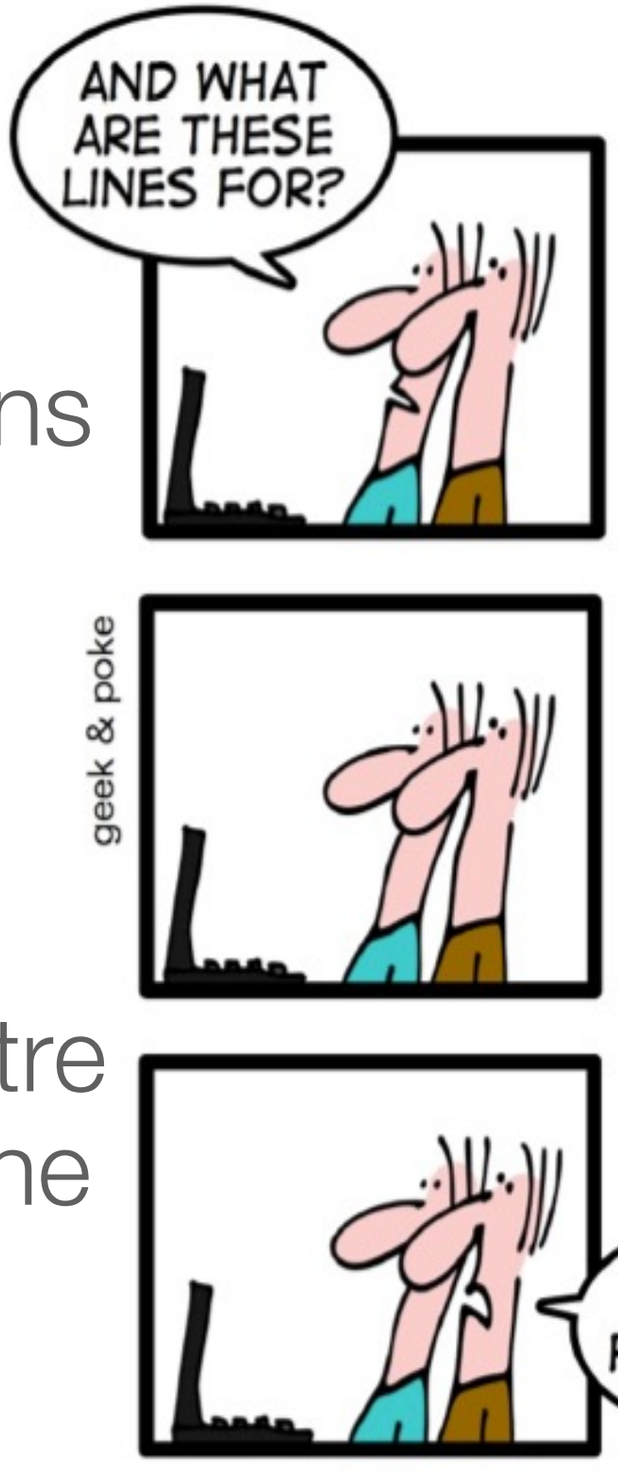
- Workflow impliquant différentes parties prenantes
  - modérateur, programmeur, concepteur, inspecteur…

- Déroulement (ex.)
  - le programmeur fournit et explique son programme
  - le concepteur et l'inspecteur apportent leur expertise
  - les fautes sont listées
  - (pas corrigées => à la charge du programmeur)

Inspection de code

- Efficacité : plus de 50 % de l'ensemble des fautes d'un projet sont détectées lors des inspections si il y en a (en moyenne plus de 75%)

- Défaut : mise en place lourde, nécessité de lien transversaux entre équipes, risques de tension…tâche plutôt fastidieuse



Inspection de code

- Règles
  - être méthodique
  - un critère : le programme peut-il être repris par quelqu'un qui ne l'a pas fait
  - un second critère : les algorithmes/l'architecture de contrôle apparaît-elle clairement ?
  - décortiquer chaque algo et noter toute redondance curieuse (coller) et toute discontinuité lorsqu'il y a symétrie (ce qui peut révéler une modif incomplète du programme)

Test statique : revue de code

Exemple: vérification de la clarté (procédural)

R1 :Détermination des paramètres globaux et de leur impact sur les fonctions propres

```
program recherche_tricho;
uses crt;
const
   max_elt = 50;
   choix1  =  1;
   choix2  =  2;
   fin     =  3;
type
   Tliste  = array[1..max_elt] of integer;
var
   liste              : Tliste;
   taille, choix, val : integer;
   complex            : integer;
```

- ✓But du programme non exprimé
- ✓Manque de commentaires
- ✓Identificateurs non explicites

Test statique : revue de code

R2 : Existence d'un entête clair pour chaque fonction

```
{---------------------------------------------------------------------
                Recherche recursive d'une valeur dans une liste triee
---------------------------------------------------------------------}
```

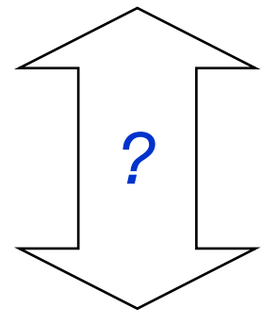
Test statique: : revue de code pour chaque fonction

Commentaires minimum

manque
- le nom de la fonction
- dépendance avec autres variables/fonctions

{ parametres d'entree : liste L
la valeur recherchee
indice gauche
indice droit
resultat : complexite de la recherche }

→ *Interface Spécifiée*



Interface implantée

```
function rech_rec(L : Tliste ; val, g, d : integer) :
                                    integer ;
```

Test statique : revue de code

```
var i, pt, dt : integer;

begin
   affiche(L, g, d);
   if g<d
      then
         begin
            pt :=g+(d-g) div 3;
            if val > L[pt]
               then
                  begin
                     dt := (pt+1+d) div 2;
                     if val > L[pt]
                        then rech_rec:=2+rech_rec(L, val, dt+1, d)
                        else rech_rec:=2+rech_rec(L, val, pt+1, dt)
                  end
               else rech_rec:=1+rech_rec(L, val, g, pt)
         end
      else rech_rec:=0
end; { rech_rec }
```

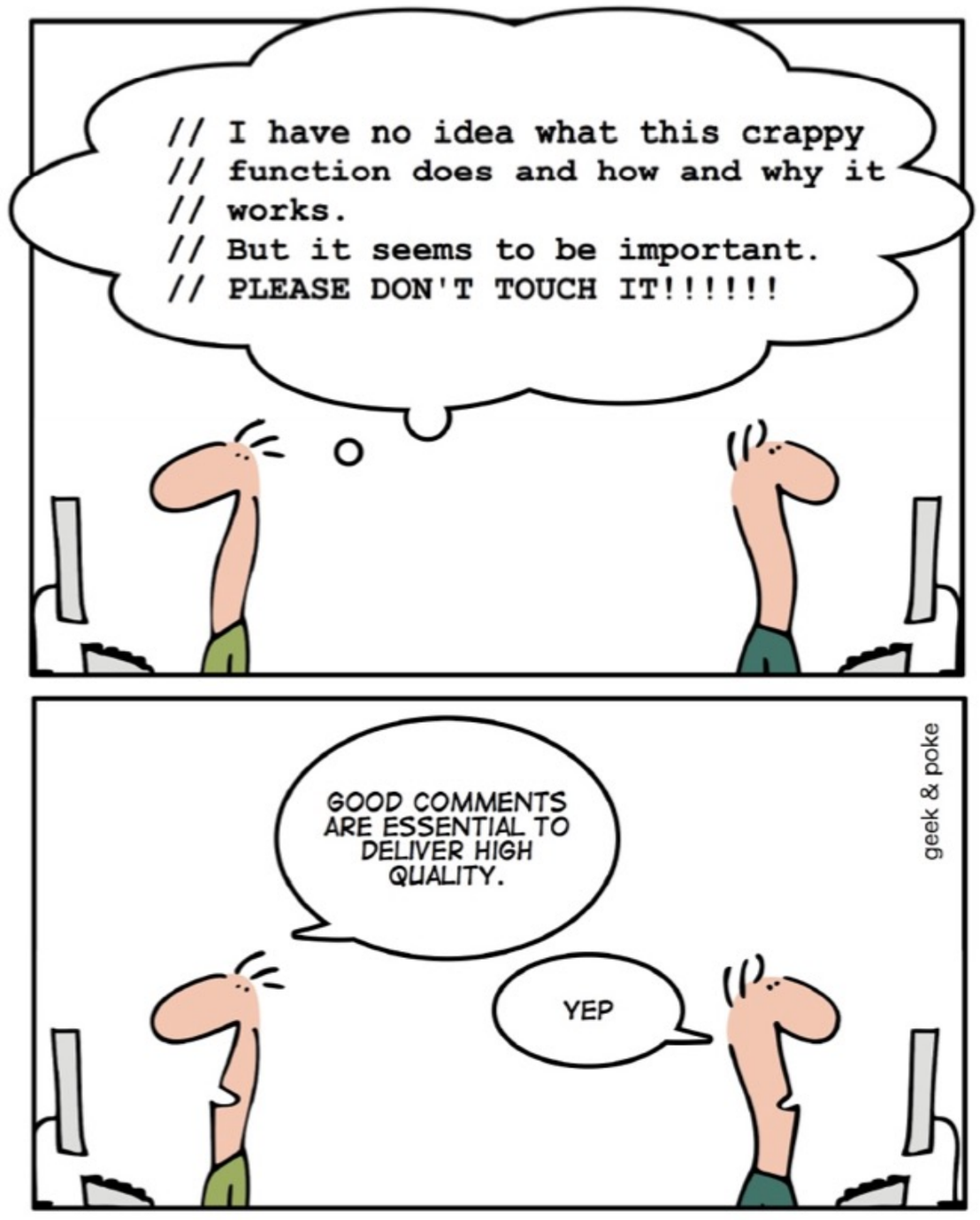
???

Action non spécifiée

Répétition ?

```java
public class C {
	private int[] l;
	private final static int s = 10;
	public C() {
		m();
		x(l.length);
	}
	private void m() {
		l = new int[s];
		Random r = new Random(System.currentTimeMillis());
		for(int i = 0; i < l.length; i++) {
			l[i] = Math.abs(r.nextInt() % s);
		}
	}
	private void n(int i, int k) {
		int t = l[i];
		l[i] = l[k];
	}
	public void x(int n) {
		System.out.println(n);
		boolean a = true;
		while(a) {
			a = false;
			for(int i = 1; i < n; i++) {
				if (l[i] < l[i - 1]) {
					n(i, i - 1);
					a = true;
				}}
			n = n -1;
		}
	}
}
```

Test statique: revue de code - le cas de l'objet



Code Reviews at Google

- "All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language. Most people use Mondrian to do code reviews, and obviously, we spend a good chunk of our time reviewing code."

-- Amanda Camp, Software Engineer, Google

Code reviews at Yelp

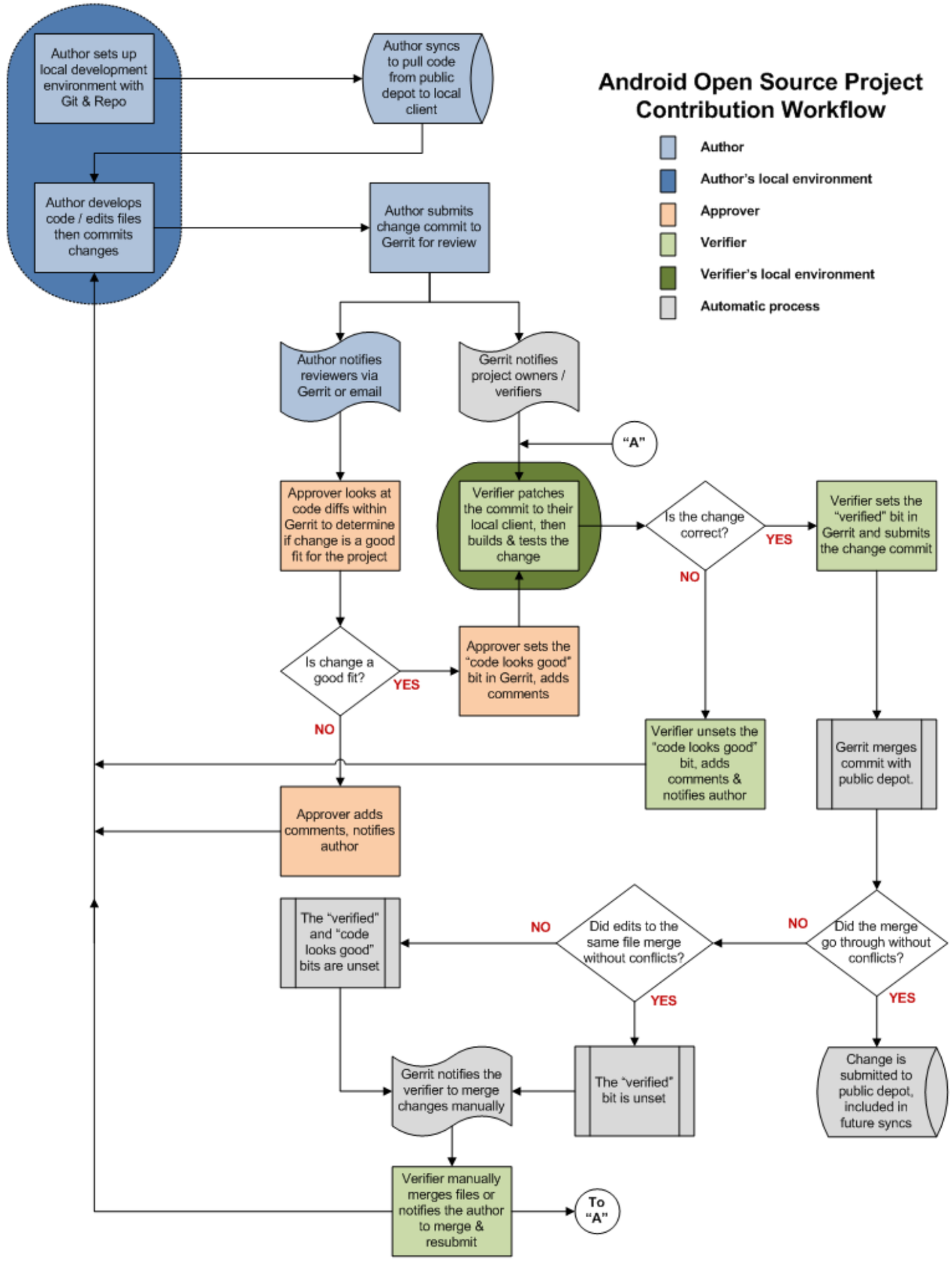
- “At Yelp we use review-board. An engineer works on a branch and commits the code to their own branch. The reviewer then goes through the diff, adds inline comments on review board and sends them back. The reviews are meant to be a dialogue, so typically comment threads result from the feedback. Once the reviewer's questions and concerns are all addressed they'll click "Ship It!" and the author will merge it with the main branch for deployment the same day.”

-- Alan Fineberg, Software Engineer, Yelp

Inspection de code

- Gestionnaires de version
- Processus clair avec Github, Gitlab…
  - Developer makes changes
  - Developer commits code
  - Developer pushes to Github
  - Developer sends pull request to team lead/reviewer
  - Reviewer accepts changes & merges code into the main repository
- Outils dédiés
  - https://www.reviewboard.org/
  - Gerrit (Google) https://code.google.com/p/gerrit/

Android Open Source Project Contribution Workflow



Plan

1. Revue de code
2. Règles de codage
3. Vérifications automatiques

Règles de codage

- Règles de codage
  - Combinaison de règles syntaxiques (règles d'écriture) et de règles sémantiques
- Bonne pratique pour améliorer:
  - Lisibilité du code
  - Correction
  - Gestion mémoire et réduire les effets de bords
  - Testabilité

Règles de codage: exemple

- Règles de structuration (template) d'un
  - Projet (préfixe sur noms de méthode, etc.)
  - D'un programme
    - format des .h et des .c, .cc
    - Présentation des classes
      - Format des fonctions de classe
- Règles d'évolution du code
  - Noter les changements et pourquoi
- Principe : utilisation d'un template et d'annotations/tags prédéfinis
  - Permet des analyses/vérifications automatiques

Règles de codage: exemple

- Une ligne par instruction
- Eviter les instructions qui cassent le flot de contrôle structuré (ex: goto)
- Instructions de contrôle
  - Retour chariot et indentation dans les expressions trop longues

```
if ((foo.next == NULL) &&
    (total_count < needed) &&
    (needed <= MAX_ALLOT) &&
    (server_active(current_input)))
```

  - Dans les prédicats, parenthèser autour de chaque opérateur booléen

```
if (a || b && c)          : NOK

if ((a || b) && c)        : OK
```

  - Ouverture et fermeture d'accolades (C / Java) sur une ligne dans les structures de contrôle
  - Éviter le comparateur « différent » (<> ou =!) dans les conditions de terminaison d'une boucle
  - Switch/case: toujours prévoir le cas par défaut

Example: sun.java coding guidelines

7 - Statements

7.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++;          // Correct
argc--;          // Correct
argv++; argc--; // AVOID!
```

7.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as an `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

7.3 return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

7.4 if, if-else, if else-if else Statements

The `if-else` class of statements should have the following form:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
```

Règles de codage: exemple

- Faire du code simple

```
if (!(My_Bool == true))
{
      bad = true;
}
else
{
      bad = false;
}
```

```
if (My_Bool)
{
    bad = false;
}
else
{
      bad = true;
}
```

... But simply :

```
bad = !My_Bool;
```

```
bad = !My_Bool;
```

⇨ Mesures de complexité du code

Code simple

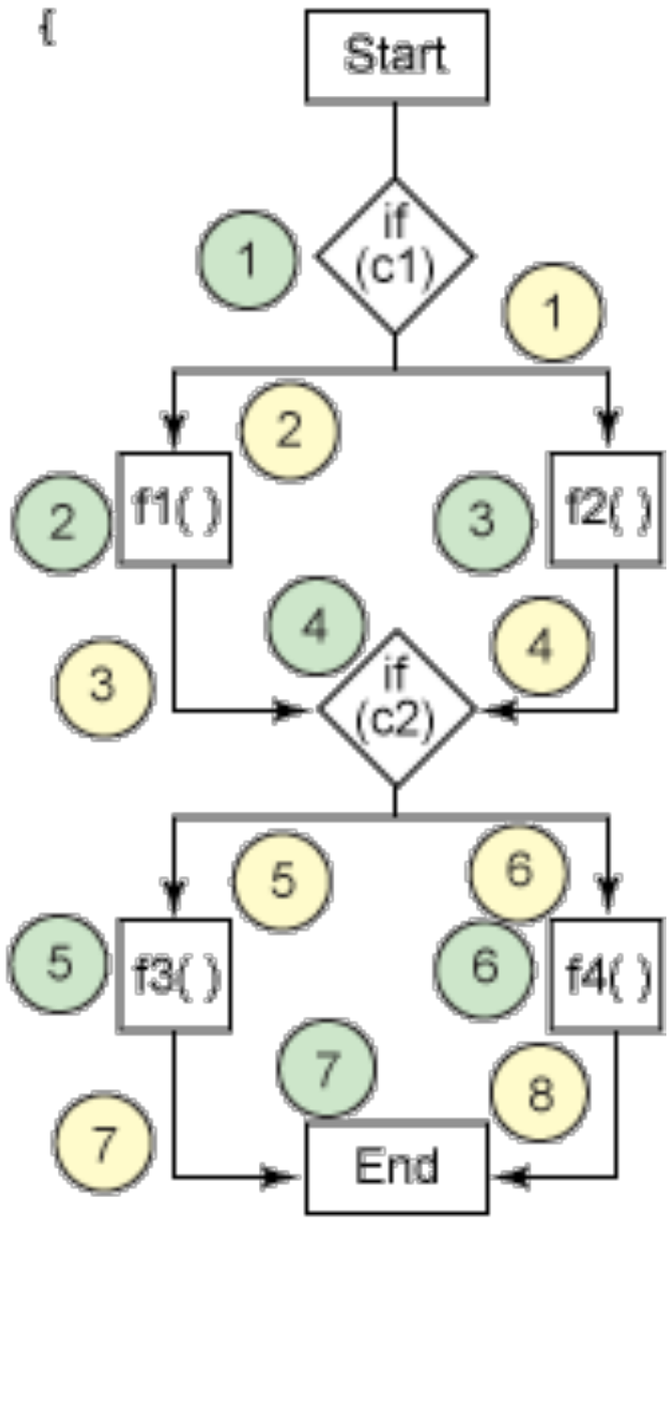
```
if (entityImplVO != null) {
  List actions = entityImplVO.getEntities();
  if (actions == null) {
     actions = new ArrayList();
  }
  Iterator enItr = actions.iterator();
  while (enItr.hasNext()) {
    entityResultValueObject arVO = (entityResultValueObject) actionItr
     .next();
    Float entityResult = arVO.getActionResultID();
    if (assocPersonEventList.contains(actionResult)) {
      assocPersonFlag = true;
    }
    if (arVL.getByName(
      AppConstants.ENTITY_RESULT_DENIAL_OF_SERVICE)
         .getID().equals(entityResult)) {
      if (actionBasisId.equals(actionImplVO.getActionBasisID())) {
        assocFlag = true;
      }
    }
    if (arVL.getByName(
     AppConstants.ENTITY_RESULT_INVOL_SERVICE)
      .getID().equals(entityResult)) {
     if (!reasonId.equals(arVO.getStatusReasonID())) {
       assocFlag = true;
     }
   }
 }
}else{
  entityImplVO = oldEntityImplVO;
}
```

Test statique: métriques

- Taille des méthodes (LOC) – *et variantes*
- Complexité cyclomatique
  - nombre de points de décision + 1 (if, else, for, etc.)
  - nombre minimum de cas de test à écrire
- Couplage / cohésion
  - cohésion d'une classe -> single responsibility
  - couplage du système -> éviter le 'plat de spaghetti'

Complexité cyclomatique

```
public void doIt() {

      if (c1) {
            f1();
      } else {
            f2();
      }
      if (c2) {
            f3();
      } else {
            f4();
      }
}
```

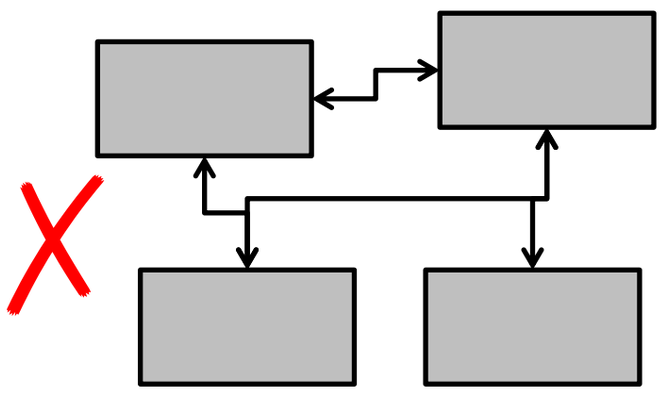


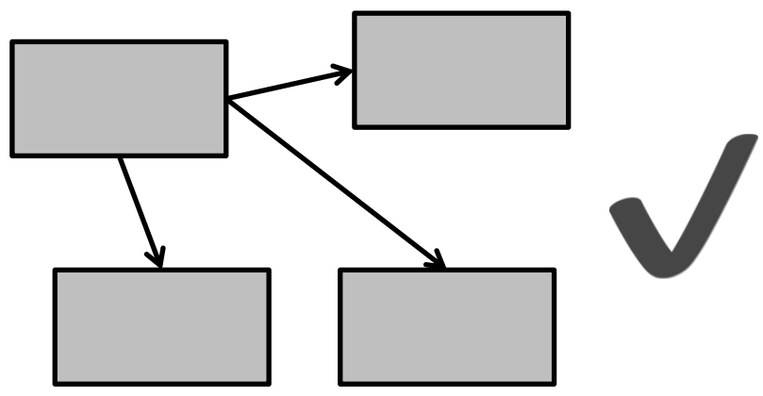
- CC=E-N+2P
  - E: # edges
  - N: # nodes
  - P: # exit nodes

- CC=8-7+2

Couplage

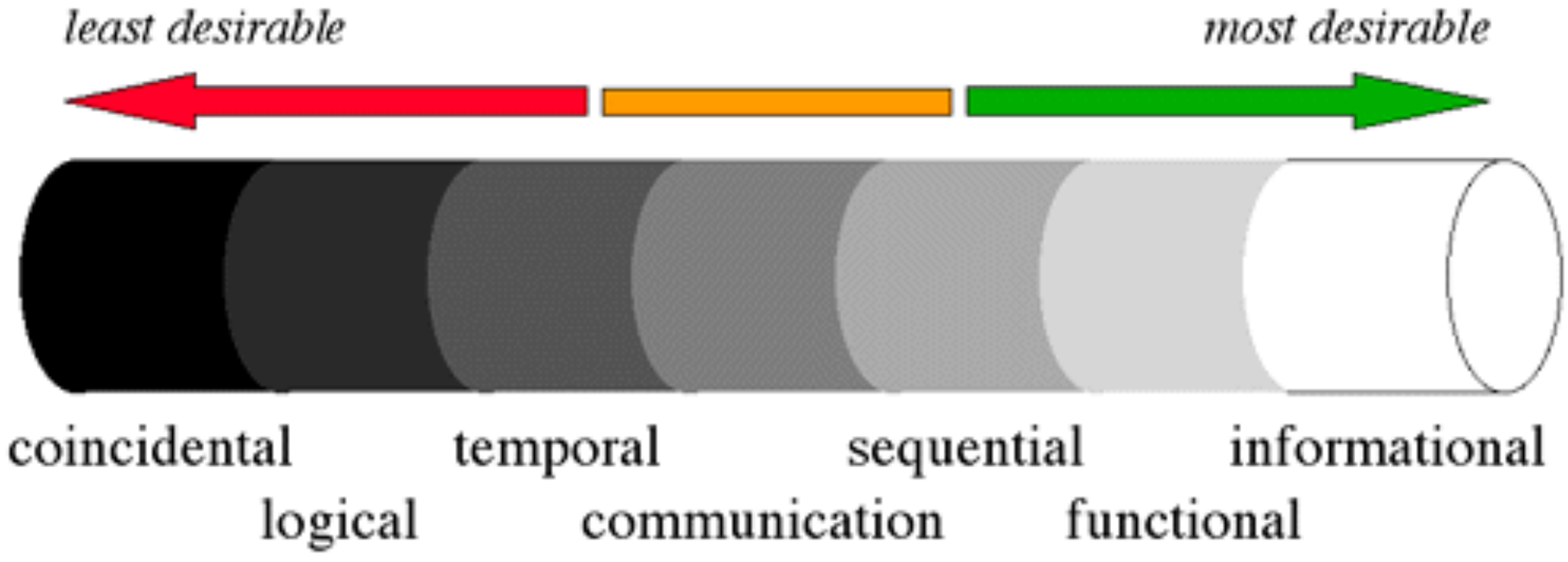
- le couplage d'une classe C est le nombre de classes dont dépend C
- couplage fort= risque de propagation d'erreurs
- patterns de communication limitent le couplage
  - ex: publish / subscribe





Cohésion

- Un module n’a qu’une responsabilité
  - cohérence fonctionnelle de l’API
  - les méthodes d’une classe accèdent toutes à tous les attributs



Loi de Demeter

- Une méthode ne doit interagir qu'avec ses « amis », i.e., appeler des méthodes seulement sur
  - self
  - un paramètre de méthode
  - un objet créé par la méthode
  - un attribut de la classe

Loi de Demeter

```
public class Foo {

    public void example(Bar b) {
        C c = b.getC()
        c.doIt();
        b.getC().doIt();
        D d = new D();
        d.doSomethingElse();
    }
}
```

ok: b is a parameter of example

violation: c is not from Foo
should do b.doItOnC();

violation: variation of the previous

ok: d is created locally

Plan

1. Revue de code
2. Règles de codage
3. Vérifications automatiques

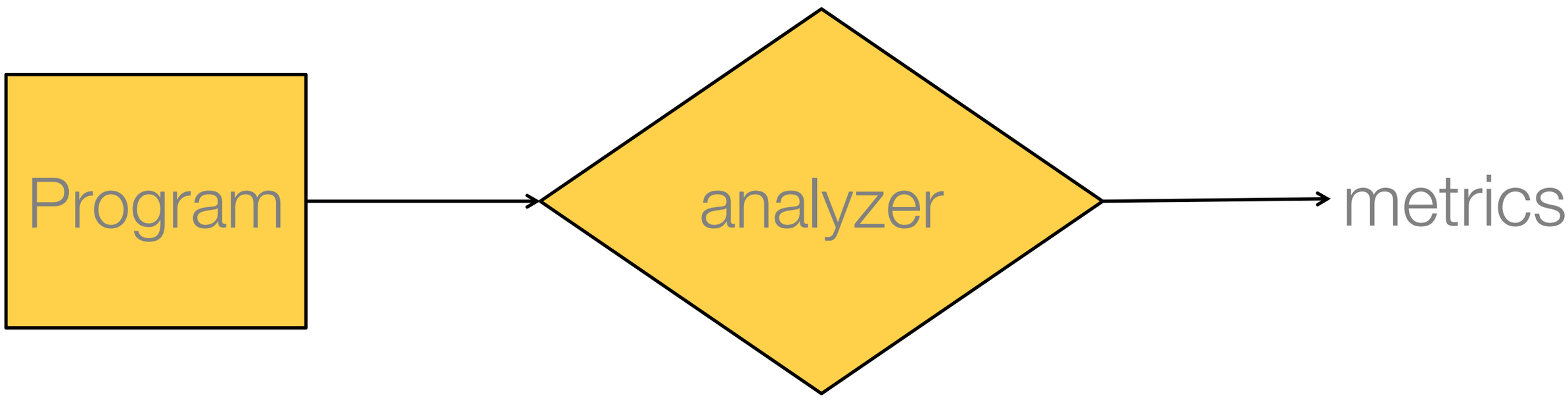
Vérifications automatiques

- Les analyses statiques sont des logiciels qui prennent en entrée le source d'un programme
- Ils analysent le programme dans le but de détecter
  - le non-respect de règles de codage
  - des incohérences dans la programmation (ex: code mort)
- Très efficace pour accompagner les revues de code

Static analysis

- *Control flow analysis.* Cyclic dependencies, dead code.
- *Data flow analysis.* Uninitialized variables, null pointers, buffer overflows, etc.
- *API analysis.* Consistency between interface and implementation
- etc…

Tools for static analysis



- Analyzer is a kind of compiler
- With other types of rules
  - usually assume a syntactically correct source program
- Most analyzers based on the Visitor pattern
  - or AST + pattern matching

PMD

- Automatic analysis of Java source code
- Looks for the occurrence of ‘bugs’ defined as a set of rules over the AST

UnconditionalIfStatement

Since: PMD 1.5

Do not use "if" statements whose conditionals are always true or always false.

```
//IfStatement/Expression
 [count(PrimaryExpression)=1]
 /PrimaryExpression/PrimaryPrefix/Literal/BooleanLiteral
```

Example(s):

```
public class Foo {
        public void close() {
                if (true) {             // fixed conditional, not recommended
                        // ...
                }
        }
}
```

PMD

- WhileLoopsMustUseBraces
  - Since: PMD 5.0
  - Avoid using 'while' statements without using curly braces.

```
while (true) {
    x++;
}
```



```
while (true)
    x++;
```

WhileLoopsMustUseBraces

```
package vv.tp5;

import net.sourceforge.pmd.lang.ast.Node;
import net.sourceforge.pmd.lang.java.ast.*;
import net.sourceforge.pmd.lang.java.rule.AbstractJavaRule;

public class WhileLoopsMustUseBracesRule extends AbstractJavaRule {

    public Object visit(ASTWhileStatement node, Object data) {
        Node firstStmt;
        firstStmt = node.jjtGetChild(1);
        if (!hasBlockAsFirstChild(firstStmt)) {
            //ajout de la violation
            addViolation(data, node);
        }
        return super.visit(node,data);
    }

    private boolean hasBlockAsFirstChild(Node node) {
        return (node.jjtGetNumChildren() != 0 && (node.jjtGetChild(0) instanceof
ASTBlock));
    }
}
```

PMD

- Static analysis does not require running the program
  - no need for test cases
  - provides results that always hold
- Need to approximate
  - potential risk for false positives / negatives

PMD

- CloseResource
  - Since: PMD 1.2.2
  - Ensure that resources (like Connection, Statement, and ResultSet objects) are always closed after use.

CloseResource

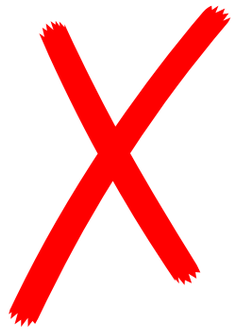
```
public class Bar {

    public void foo() {
        Connection c = pool.getConnection();
        try {
            // do stuff
        } catch (SQLException ex) {
            // handle exception
        } finally {
            // c.close();
        }
    }
}
```

CloseResource

```
public class Bar {

    public void foo() {
        Connection c = pool.getConnection();
        try {
            // do stuff
        } catch (SQLException ex) {
            // handle exception
        } finally {
            c.close();
        }
    }
}
```

CloseResource

```
public class Bar {

    public void foo() {
        Connection c = pool.getConnection();
        try {
            // do stuff
        } catch (SQLException ex) {
            // handle exception
        } finally {
            bar(c);
        }
    }

    public void bar(Connection c) {
        try {
            // do stuff
        } catch (SQLException ex) {
            // handle exception
        } finally {
            c.close();
        }
    }
}
```



false negative

CloseResource

```
public class Stream {
    BufferedReader reader;

    public Stream(String fileNameData) throws FileNotFoundException {
        this.reader = new BufferedReader(new FileReader(fileNameData));
    }
    public void readData()  {
        try {
            String line = reader.readLine();
            while (line != null) {
                System.out.println(parseLine(line));
                line = reader.readLine();
            }
            reader.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    protected double parseLine(String line) {
        String[] split = line.split(";");
        return Double.parseDouble(split[0]) / Double.parseDouble(split[1]);
    }
    public static void main(String[] args) throws FileNotFoundException {
        Stream s = new Stream(args[0]);
        s.readData();
    }
}
```

stream not closed

false positive

PMD

- Approx. 300 rules
  - basic OO rules, language-specific
  - possible to select a subset of rules for each PMD run
- PMD is open
  - possible to define new rules in Java (Visitor pattern) or Xpath (pattern matching on the AST)
- Static analysis approximates runtime information
  - risk of false positives / false negatives

Findbugs

- Automatic analysis of Java bytecode (.jar)
- Based on rules to match common bug patterns
- Syntactic and data flow analysis

```
1  Person person = aMap.get("bob");
2  if (person != null) {
3      person.updateAccessTime();
4  }
5 String name = person.getName();
```

Findbugs

- Findbugs is open
  - possible to define custom detectors
- Uses BCEL (Byte Code Engineering Library)
  - an API on the tree through a Visitor pattern

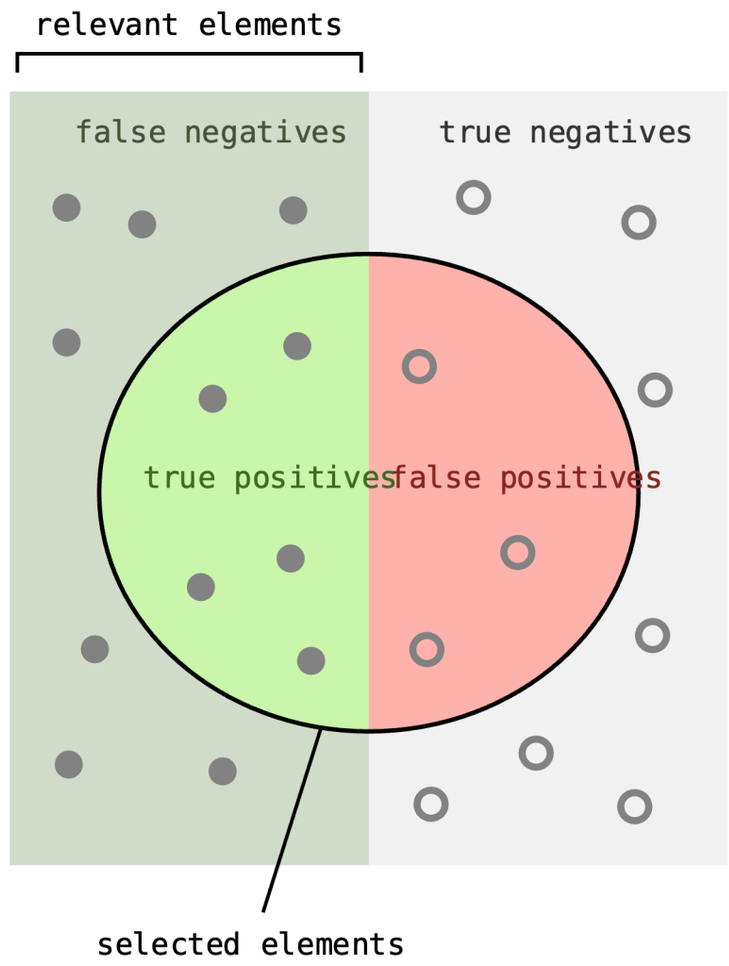
Static analysis tools

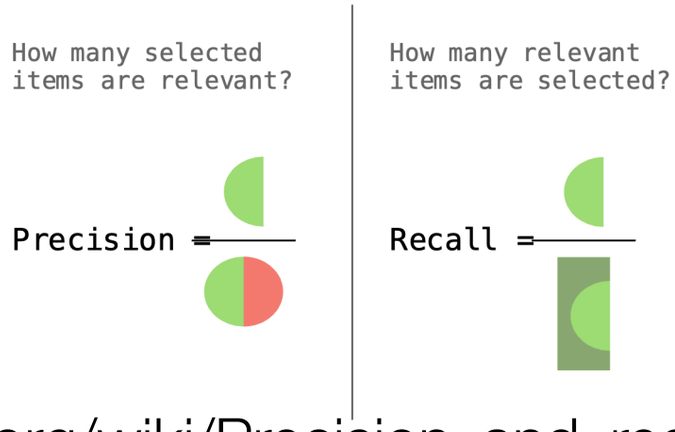
- PMD and Findbugs
  - both rule based, with different set of rules
  - integrated in most of the IDEs
  - can be integrated in build and continuous integration processes
- Other tools
  - CheckStyle for coding rules
  - JDT (non initialized and non used variables, etc.)

Precision & Recall Scoring

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$





https://en.wikipedia.org/wiki/Precision_and_recall

Summary

- Static testing can be done to find defect and deviation using:
  - Structured group examinations
    - Reviews: Inspection, walkthrough, technical review, informal review
  - Static analysis using static analyzers
    - Compiler
    - Data flow analysis
    - Control flow analysis

Summary

- Static testing
  - looks for generic defects
  - finds mostly « suspicious » zones
  - is exhaustive
- Static testing does not find all kinds of bugs
  - false negatives/positives in static analysis
  - no behavioral verification