Esercitazioni di Ingegneria del Software

Analisi Statica del Codice

Nico Pellegrinelli

Università degli Studi di Bergamo

1° semestre a.a. 2025/2026

Setup

Introduzione all'analisi statica

Metriche

Prova pratica

Section 1

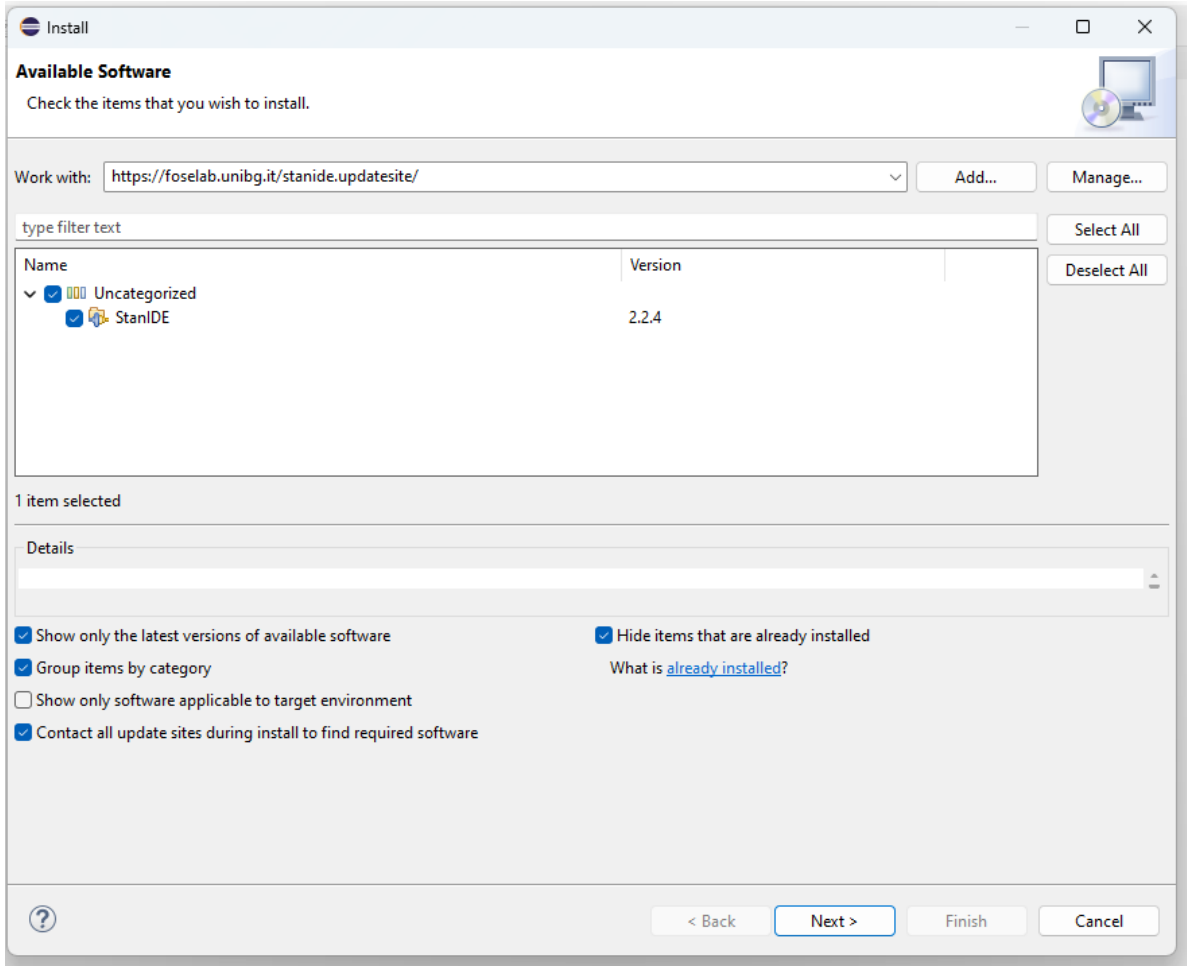
Setup

Eclipse

Useremo Java e l'IDE Eclipse. Potete scaricarlo a:
`https://www.eclipse.org/downloads/packages/release/2025-09/r/eclipse-ide-java-developers`

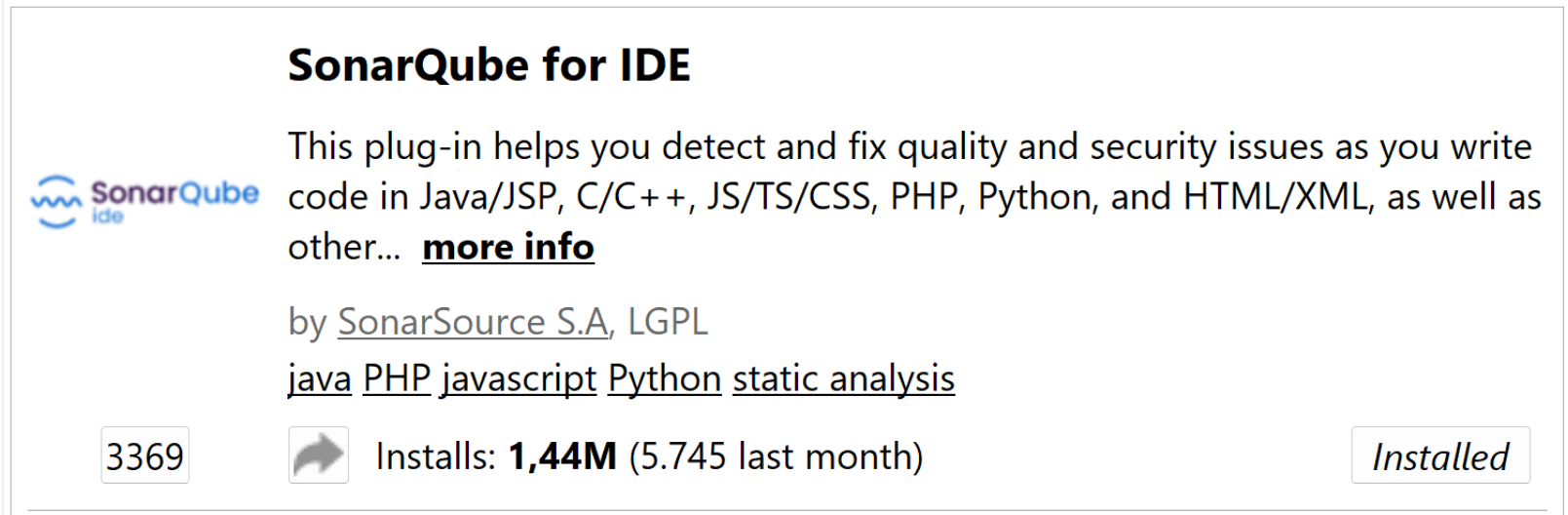
Stan4J

Update site:
`https://foselab.unibg.it/stanide.updatesite/`



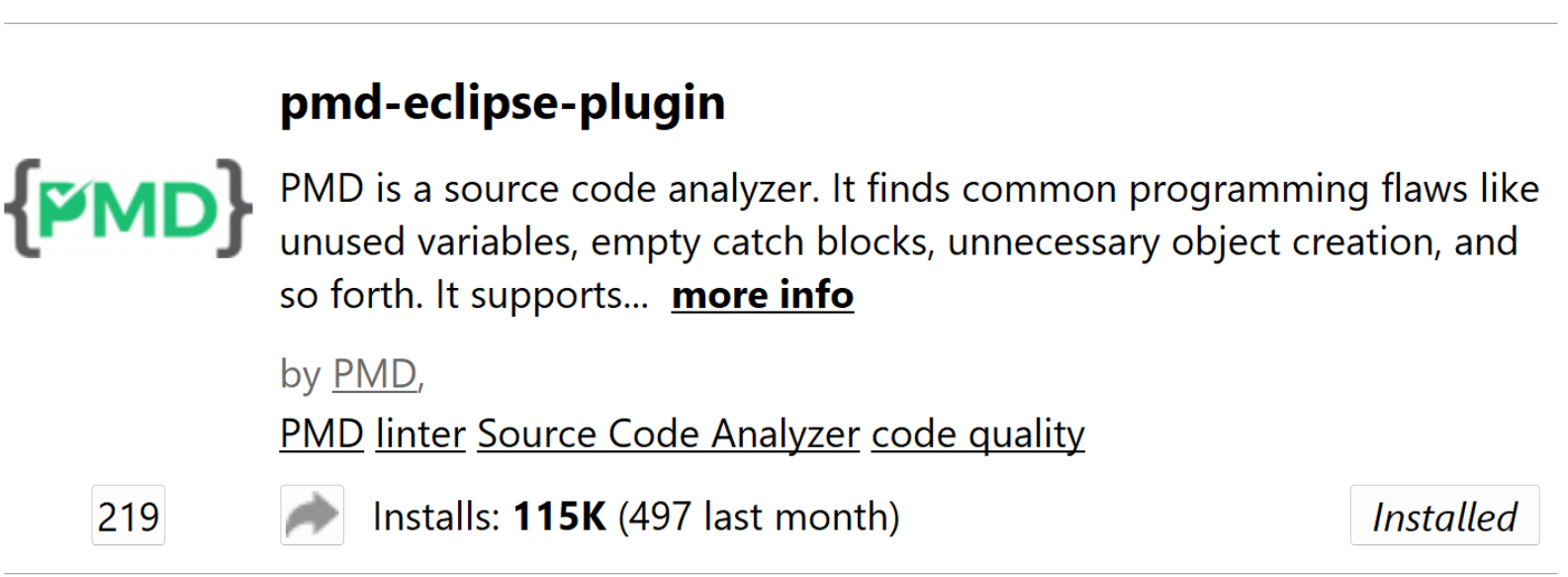
SonarQube

DaI Marketplace:



EclipsePMD

Dal Marketplace:



Attenzione a non installare ‘eclipse-pmd‘ o ‘PMD‘

Progetto di esempio

- Repository GitHub con un semplice progetto Java (BadOnPurpose) su cui useremo i tre tool: `https://github.com/nicopellegrinelli/EsercitazioniIdS-public`. Clonatela sul vostro PC.
- Nel branch solution c'è anche il progetto Refactored ottenuto applicando un refactoring guidato dai tool al progetto BadOnPurpose: `https://github.com/nicopellegrinelli/EsercitazioniIdS-public/tree/solution`.

Section 2

Introduzione all'analisi statica

Introduzione all'analisi statica

- Viene analizzato il codice **senza eseguirlo** (a differenza del testing, che è analisi dinamica del codice).
- Esistono diversi tool di analisi statica del codice che permettono di:
  - Valutare e migliorare la **qualità del codice** attraverso l'uso di metriche;
  - Visualizzare e analizzare la **struttura del codice** (le relazioni di dipendenza) per comprenderlo meglio, agevolando il refactoring;
  - **Identificare i bug** il prima possibile, idealmente quando vengono introdotti.

Aiutano a sviluppare codice di alta qualità, diminuendo i costi di manutenzione.

Metriche del software (1)

- È fondamentale sviluppare del software di **buona qualità**.
- Il codice è astratto e complesso, quindi è difficile valutarne la qualità "a occhio".
- È possibile monitorare la qualità del software che si sta sviluppando attraverso delle **metriche**.
- Esistono dei tool che permettono di valutare automaticamente queste metriche. Noi vedremo Stan4J.

Metriche del software (2)

- Ogni tool implementa un set diverso di metriche.
- **Attenzione**: alcune metriche possono essere interpretate in modi diversi. Quindi, è possibile che diversi tool, eseguiti sullo stesso codice, forniscano valori diversi della stessa metrica[1].

[1] Aversano, L. & Tortorella, Maria. (2018). Assessing the Impact of Measurement Tools on Software Mantainability Evaluation. 392-397. 10.5220/0006793003920397.

Struttura del software (1)

- Alcune metriche, come vedremo, sono legate alle **dipendenze** nel codice (ad esempio, chiamate di metodo e ereditarietà).
- Stan4J, come altri tool, oltre a fornire una valutazione quantitativa di un esteso set di metriche, permette di **visualizzare graficamente** la struttura delle dipendenze del codice.
- Visualizzare le dipendenze permette di comprendere meglio le problematiche relative a tali metriche.

Struttura del software (2)

- In generale, visualizzare graficamente la struttura del codice che si sta sviluppando, permette di comprenderlo meglio e facilita operazioni di **refactoring**.

Code checkers (1)

- Ci sono molti modi per ridurre il numero di bug, come testing e code review. Ma vorremmo scoprire i bug il prima possibile.
- Molti bug sono ricorrenti e possono essere catalogati in categorie note. Un esempio:

```
if(myString == "Hello World!"){...}
```

- L'idea è cercare pattern di errori ben noti. I tool che fanno ciò sono noti come static code checkers[2].
- **Non sostituiscono il testing**!

[2] P. Louridas, "Static code analysis," in IEEE Software, vol. 23, no. 4, pp. 58-61, July-Aug. 2006, doi: 10.1109/MS.2006.114.

Code checkers (2)

- Nessun code checker è **complete** (ossia, nessun code checker è in grado di trovare tutti gli errori nel programma).
- Nessun code checker è **sound** (ossia, i code checker possono riportare falsi positivi).

Code checkers (3)

Ogni tool lavora in modo diverso, ma il funzionamento di base è lo stesso:

1. **Lettura del programma** e costruzione di un suo modello (rappresentazione astratta);
2. **Ricerca dei pattern** di errore nel modello;
3. In aggiunta: **data-flow analysis** per cercare di tener traccia dei valori delle variabili.

Noi vediamo Eclipse PMD e SonarLint.

Section 3

Metriche

Metriche

Vediamole direttamente in Eclipse:
Help $\rightarrow$ Help Contents $\rightarrow$ STAN - Structure Analysis for Java $\rightarrow$ Reference $\rightarrow$ Metric Definitions
Nota: alcune metriche sono **definite su artefatti di tipi diversi** (ad esempio, ELOC è definita a livello di source folder, package, class e method, Cyclomatic Complexuty solo a livello di method)

Complexity Metrics (1)

Più il valore di queste metriche è alto e maggiore sarà la difficoltà nel testare e manutenere il codice.

- McCabe **Cyclomatic Complexity** (CC): misura la complessità di un programma (un metodo in Stan4J) attraverso il numero dei path indipendenti che contiene. Si computa attraverso il control-flow graph:

$$M = E - N + 2P$$

  - E è il numero di archi nel grafo
  - N è il numero di nodi nel grafo
  - P è il numero di componenti connessi, vale 1 nel caso di un metodo ($M = E - N + 2$)

Complexity Metrics (2)

- **Fat**: Numero di **archi del dependency graph** dell'artefatto (non le dipendenze con ciò che sta fuori, ma le dipendenze all'interno dell'artefatto. Ad esempio, per una classe è il numero di dipendenze nel grafo dei suoi metodi, campi e inner class). Intuitivamente, **indica se l'artefatto "fa troppe cose"**.
- **Tangled**: Indica se ci sono **dipendenze cicliche** nel grafo delle dipendenze di un artefatto. In Stan4J è il rapporto in percentuale tra il peso delle dipendenze "rosse" (ossia quelle che vanno contro il flusso delle dipendenze, creando un ciclo) e il peso totale di tutte le dipendenze.

Robert C. Martin Metrics (1)

Vediamo meglio le Robert C. Martin Metrics per i package:

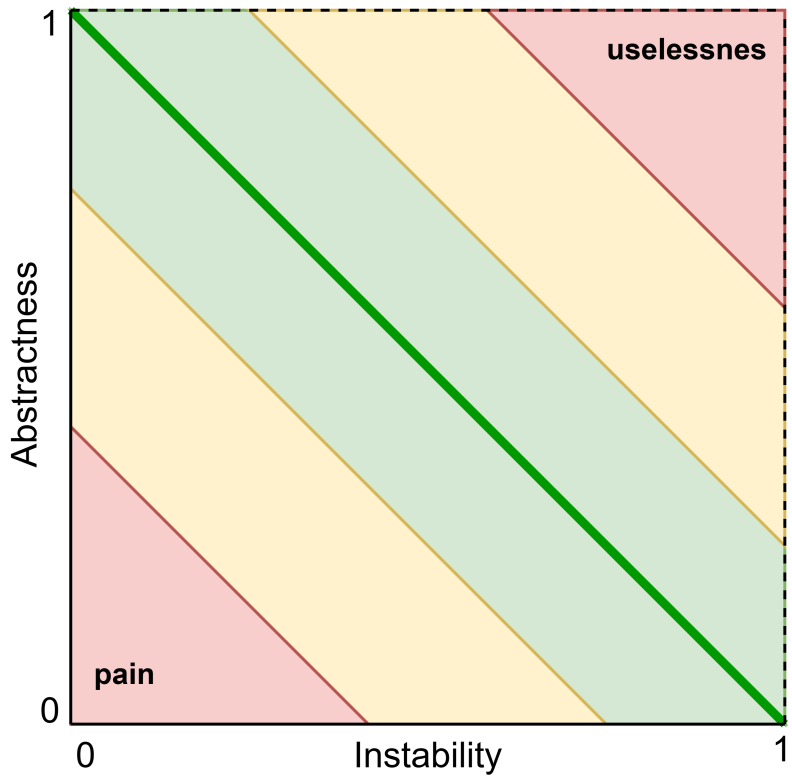
- **Afferent Coupling** (Ca): numero di classi al di fuori del package che dipendono da classi all'interno package.
- **Efferent Coupling** (Ce): numero di classi nel package che dipendono da classi al di fuori del package.

Robert C. Martin Metrics (2)

- **Instability** (I): $Ce/(Ce + Ca)$
  - Se I è vicina a 0: Ca domina su Ce, il package è molto stabile, perché sono altri package a dipendere da esso. Questo lo rende meno incline al cambiamento, perché non è influenzato direttamente da altre parti del sistema.
  - Se I è vicina a 1: Ce domina su Ca, il package è fortemente dipendente da altri package ed quindi è altamente suscettibile ai loro cambiamenti. Se cambia uno dei package da cui dipende, sarà necessario modificare anche questo package.
- **Abstractness** (A): $Na/(Na + Nc)$, con Na numero di classi astratte e interfacce nel package e Nc numero di classi concrete nel package.

Robert C. Martin Metrics (3)

- **Distance** (D): $A + I - 1$



$D = -1$: zone of pain $\qquad$ $D = 1$: zone of uselessness

Section 4

Prova pratica

Stan4J - Utilizzo

Tasto destro sul progetto (o sulla source folder) $\rightarrow$ Run As $\rightarrow$ Structure Analysis

SonarLint - Utilizzo

- Una volta installato, dovrebbe già segnalare le problematiche direttamente sul codice su cui si sta lavorando.
- Per eseguire SonarLint su un progetto:
  Tasto destro sul progetto $\rightarrow$ SonarLint $\rightarrow$ Analyze

Eclipse PMD - Utilizzo

- Lo vediamo solamente per la ricerca di copia/incolla (code duplication), ma può essere utilizzato come SonarLint: Tasto destro sul progetto $\rightarrow$ PMD $\rightarrow$ Controlla il codice con PMD
- Per controllare code duplication:
  - Tasto destro sul progetto $\rightarrow$ PMD $\rightarrow$ Trova sospetto copia ed incolla
  - Selezionare java in Language e spuntare la casella per la generazione del report file (a volte la CPD View non funziona benissimo e potrebbe essere comodo avere il report file).