Impact Analysis

MARK 2.0 Plus

Data	23/01/2026
Presentato da	Cerchia Giovanni (NF22500202) Medica Vincenzo (NF22500203)
Repository	https://github.com/vmedica/MARK-2.0-Plus

Sommario

1 CR1 — Metriche di qualità del codice (Radon)

1.1 Metodo di Impact Analysis adottato

1.1.1 Contesto e obiettivo

MARK 2.0 Plus è un tool object-oriented che esegue analisi statica di repository Python per classificare progetti "applied ML" (Producer/Consumer) e produrre output in CSV; l'architettura è modulare e include, tra i moduli "core", l'orchestrazione dell'analisi tramite modules/analyzer e la costruzione degli analyzer tramite un insieme di **Facade + Factory-Registry + Builder**.
La Change Request CR1 introduce un'estensione funzionale: integrare il calcolo di metriche di qualità del codice (Radon), in particolare **Cyclomatic Complexity (CC)** e **Maintainability Index (MI)**, calcolate sui file sorgente analizzati e aggregate a livello progetto (media su blocchi per CC, media pesata su SLOC per MI), con salvataggio dei risultati in output allineato alla run corrente.
La CR è classificabile come **Enhancement**, con manutenzione di tipo **Additive/Perfective** (nuove informazioni che arricchiscono l'analisi).

1.1.2 Assenza di documentazione e scelta degli SLO

In assenza di artefatti di requisiti e design (es. SRS, diagrammi UML aggiornati), l'Impact Analysis viene condotta sugli artefatti disponibili (codice sorgente), usando traceability interna (vertical traceability) e relazioni di dipendenza tra componenti.
Il riferimento teorico utilizzato (*SOFTWARE EVOLUTION AND MAINTENANCE - TRIPATHY e NAIK*) introduce gli **SLO (Software Life-cycle Objects)** come oggetti che possono variare di granularità. Nel nostro contesto, per mantenere una granularità uniforme e compatibile con un'analisi call-graph-based in ambiente OO, si assumono inizialmente come SLO i **metodi** del sistema software.

1.2 Starting Impact Set (SIS)

1.2.1 Presupposto

Sulla base dell'analisi del sistema (Initial Report) e del contenuto della CR1, vengono identificati come *core methods* da modificare i tre metodi "template" della classe `MLAnalyzer`, responsabili del workflow principale di analisi:

- `MLAnalyzer.analyze_single_file`

- `MLAnalyzer.analyze_project`
- `MLAnalyzer.analyze_projects_set`

1.2.2 SIS finale (granularità uniforme a livello di metodo)

SIS = {
`MLAnalyzer.analyze_single_file, MLAnalyzer.analyze_project, MLAnalyzer.analyze_projects_set`
}

1.3 Candidate Impact Set (CIS)

1.3.1 Approccio: Costruzione del Call Graph (fan-in + fan-out)

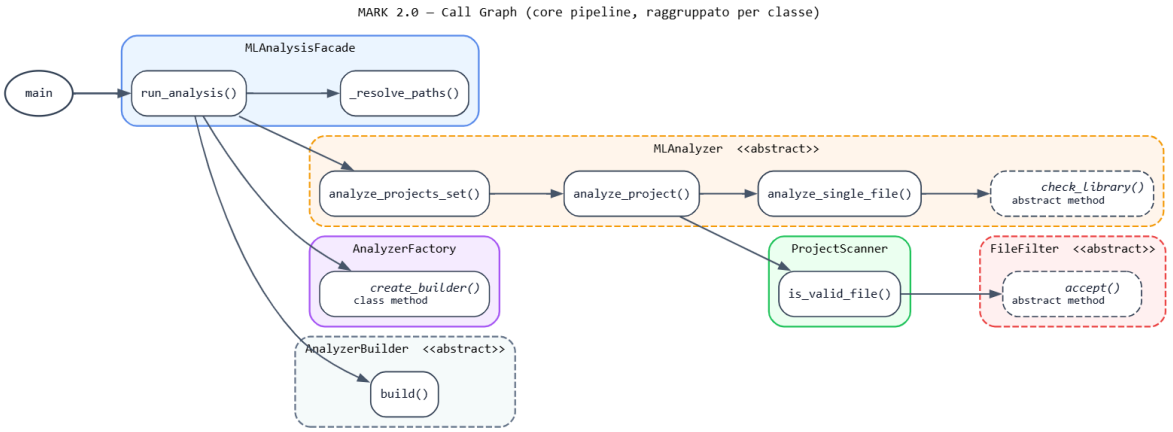
La modifica dei metodi core di MLAnalyzer può propagarsi:

- verso i **callees** (fan-out): nuove dipendenze invocate per calcolare CC/MI e salvarle;
- verso i **callers** (fan-in): eventuali adattamenti dovuti a variazioni di ritorni/dati prodotti, o a step aggiunti alla pipeline.

Per questo motivo si costruisce un sottografo includendo sia fan-out sia fan-in, così da ottenere una stima prudenziale dei candidati al cambiamento. La procedura adottata è:

1. **Seeding**: inizializzazione dei nodi del grafo con gli SLO del SIS.
2. **Espansione fan-out**: aggiunta iterativa dei metodi chiamati (callees) dai nodi correnti.
3. **Espansione fan-in**: aggiunta iterativa dei metodi chiamanti (callers) che invocano i nodi correnti.
4. **Arresto**: terminazione quando non emergono nuovi nodi interni al sistema.
5. **Filtraggio**: esclusione di nodi esterni (librerie di terze parti / built-in) per mantenere il grafo sugli SLO del sistema.

Il risultato è un sottografo statico che rappresenta una **sovra-approssimazione** dei possibili impatti.



A partire dai tre metodi del SIS, il sottografo ricostruito include:

- l'entry-point della pipeline (`main`), che invoca il Facade;
- `MLAnalysisFacade.run_analysis`, che orchestrata l'analisi e attiva la factory/builder;
- i metodi di costruzione (`AnalyzerFactory.create_builder`, `AnalyzerBuilder.build`);
- i metodi di workflow dell'analyzer (`analyze_projects_set` → `analyze_project` → `analyze_single_file`);
- i metodi ausiliari chiamati nel workflow (validazione file e filtraggio; metodo astratto `check_library`).

1.3.2 Approccio: Valutazione incrementale dei nodi

Il call graph costruito produce un insieme di nodi raggiungibili dai metodi del SIS e/o che li raggiungono: tale insieme è un **superset** di possibili impatti. Tuttavia, la CR1 è specifica (aggiunta CC/MI e relativo salvataggio/aggregazione), quindi non tutti i nodi raggiungibili sono necessariamente da modificare.
Per rendere l'Impact Analysis aderente alla definizione di CIS ("insieme potenzialmente affetto" ma non arbitrariamente esteso), si introduce un passo incrementale:

1. per ciascun nodo del sottografo, si valuta se la CR può richiedere modifica **diretta** del comportamento, dei dati prodotti/consumati, o dell'orchestrazione in cui il nodo è coinvolto;
2. se la pertinenza è nulla o trascurabile, il nodo viene escluso dal CIS.

Di seguito la valutazione dei componenti emersi:

- `main` (entry-point) — **IN** (CIS): potrebbe essere necessario per introdurre un nuovo flag/step nella pipeline.

- `MLAnalysisFacade.run_analysis` (orchestrazione end-to-end) — **IN** (CIS): l'introduzione di un nuovo ruolo di analisi può richiedere gestione specifica di output/directory e corretta attivazione della build.
- `MLAnalysisFacade._resolve_paths` — **OUT** (CIS): gestisce la risoluzione dei path dei dizionari ML (producer/consumer) e la creazione della cartella di output principale; non è necessaria per CR1.
- `AnalyzerFactory.create_builder` — **IN** (CIS): candidato prudenziale; può risultare necessario per introdurre/risolvere il nuovo ruolo `METRICS` (potrà diventare FPIS se l'estensione avviene senza modifiche alla factory).
- `AnalyzerBuilder.build (astratto)` — **IN** (CIS): candidato prudenziale nel caso siano necessarie dipendenze/configurazioni diverse per la costruzione del nuovo analyzer.
- `ProjectScanner.is_valid_file / FileFilter.accept` — **OUT** (CIS): CR1 non modifica i criteri di selezione dei file; si riusano invariati i file già analizzati.
- `MLAnalyzer.check_library` (astratto) — **OUT** (CIS), con precisazione: la logica librerie/keyword non viene modificata; tuttavia CR1 richiede una nuova classe concreta che implementi/override `check_library` per il ruolo metrics (ad es. restituendo liste vuote). L'impatto è quindi sulla nuova implementazione concreta, non sul metodo astratto.
- `AnalyzerRole` (Enum) — **IN** (CIS): necessario per introdurre il nuovo ruolo `METRICS`.

Componenti da aggiungere

Si prevede l'introduzione dei seguenti elementi:

- `MLMetricsAnalyzer`: classe concreta di `MLAnalyzer` che esegue l'override di `check_library` restituendo liste vuote (in quanto non interessa l'estrazione di librerie/keyword).
- `MLAnalyzer._save_metrics_csv`: metodo per salvare i CSV relativi alle metriche.
- `MetricsAnalyzerBuilder`: classe concreta di `AnalyzerBuilder`, costruttore di `MLMetricsAnalyzer` richiamato dal Facade.

File e cartelle di output impattate

- Cartella `logs`: i file `.log` contenuti in `logs` potrebbero essere arricchiti con nuove informazioni di log generate dal nuovo step introdotto.

- Cartella `output`: la cartella degli output potrà essere modificata dalla creazione di una nuova cartella risultati `metrics`, con una sottocartella per ogni run (`metrics_<num_run>`) e file `.csv` contenenti le metriche calcolate.

1.3.3 CIS finale

Sulla base del sottografo e della valutazione incrementale, il Candidate Impact Set viene definito come:

CIS.methods = {
`main`, `MLAnalyzer.analyze_single_file`, `MLAnalyzer.analyze_project`, `MLAnalyzer.analyze_projects_set`, `MLAnalysisFacade.run_analysis`, `AnalyzerFactory.create_builder`, `AnalyzerBuilder.build`, `MLAnalyzer._save_metrics_csv` (*TO_ADD*)
}

CIS.classes = { `AnalyzerRole`, `MLMetricsAnalyzer` (*TO_ADD*), `MetricsAnalyzerBuilder` (*TO_ADD*) }

CIS.folders = { `logs`, `output` }

1.4 Implementazione delle Modifiche

A seguito dell'implementazione di CR1, è stato possibile verificare l'impatto effettivo delle modifiche introdotte sui componenti del sistema.

1.4.1 Actual Impact Set (AIS)

Risultano effettivamente modificati/aggiunti:

AIS.methods = {
`main`, `MLAnalyzer.analyze_single_file`, `MLAnalyzer.analyze_project`, `MLAnalyzer.analyze_projects_set`, `MLAnalyzer._save_metrics_csv` (*ADDED*)
}

AIS.classes = { `AnalyzerRole`, `MLMetricsAnalyzer` (*ADDED*), `MetricsAnalyzerBuilder` (*ADDED*)}

AIS.folders = { `logs`, `output` }

1.4.2 False positive Impact Set (FPIS)

Sono risultati falsi positivi i seguenti elementi previsti nel CIS ma non modificati nell'implementazione:

FPIS = CIS \ AIS = {

`MLAnalysisFacade.run_analysis, AnalyzerFactory.create_builder, AnalyzerBuilder.build`

}

1.4.3 Discovered Impact Set (DIS)

Non sono stati individuati componenti impattati non inclusi nel CIS:

DIS = ∅

1.5 Metriche

Per valutare la qualità dell'Impact Analysis si utilizzano le metriche tipiche di information retrieval applicate ai set CIS/AIS:

$$Precision = \frac{|CIS \cap AIS|}{|CIS|} = \frac{10}{13} = 0,77$$

$$Recall = \frac{|CIS \cap AIS|}{|AIS|} = \frac{10}{10} = 1$$

In sintesi, l'analisi è stata conservativa: ha massimizzato la copertura dei componenti realmente impattati (ottimo recall), al costo di una moderata sovra-approssimazione (precision < 1).

2 CR2 (GUI) e CR3 (Dashboard)

2.1 Metodo di Impact Analysis adottato

Per le CR2 (GUI) e CR3 (Dashboard) si adotta un approccio differente rispetto alla CR1.
Entrambe le modifiche sono state progettate **come moduli separati**, con l'obiettivo esplicito di **non modificare il codice esistente** del core (analisi/cloning). In particolare, GUI e Dashboard operano come *strati aggiuntivi* che **invocano** funzionalità già presenti (interfacciandosi al core come "client"), senza richiedere refactoring o variazioni a classi/metodi preesistenti.
Di conseguenza, l'Impact Analysis sui componenti già presenti nel sistema risulta **banalmente nulla**: non è prevista propagazione dell'impatto sul core e non è necessario costruire un call graph basato sui fan-in/fan-out come in CR1.

2.2 Starting Impact Set (SIS)

Dato che la CR è implementata **solo tramite componenti aggiunti** (nuovo sottosistema GUI/Dashboard) e con vincolo progettuale di **non modifica del core** del sistema esistente, il SIS relativo ai componenti esistenti è:

SIS = ∅

2.3 Candidate Impact Set (CIS)

Per la stessa motivazione, anche il CIS **relativo ai componenti esistenti** risulta:

CIS = ∅

Poiché non è prevista alcuna modifica al codice preesistente, non viene svolto un calcolo approfondito del CIS tramite dipendenze interne al core (come fatto per CR1).

2.4 Implementazione delle modifice (AIS / FPIS/ DIS)

Coerentemente con il vincolo progettuale ("no changes to core"), i componenti effettivamente impattati (AIS) sono attesi come segue:

- **AIS =** ∅

2.5 Metriche

Dato che, per definizione progettuale, **SIS =** ∅ e **CIS =** ∅, le metriche (Precision/Recall) applicate ai soli componenti preesistenti non risultano informative.
L'analisi è quindi volutamente **semplificata** e si limita a esplicitare che l'impatto delle nuove componenti non comporta propagazione sulle componenti esistenti.