UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Facoltà di Ingegneria

Corso di Studi in Ingegneria Informatica

Elaborato finale in Ingegneria del Software e dei Dati

Strumenti open source per la misura di caratteristiche di qualità del codice sorgente

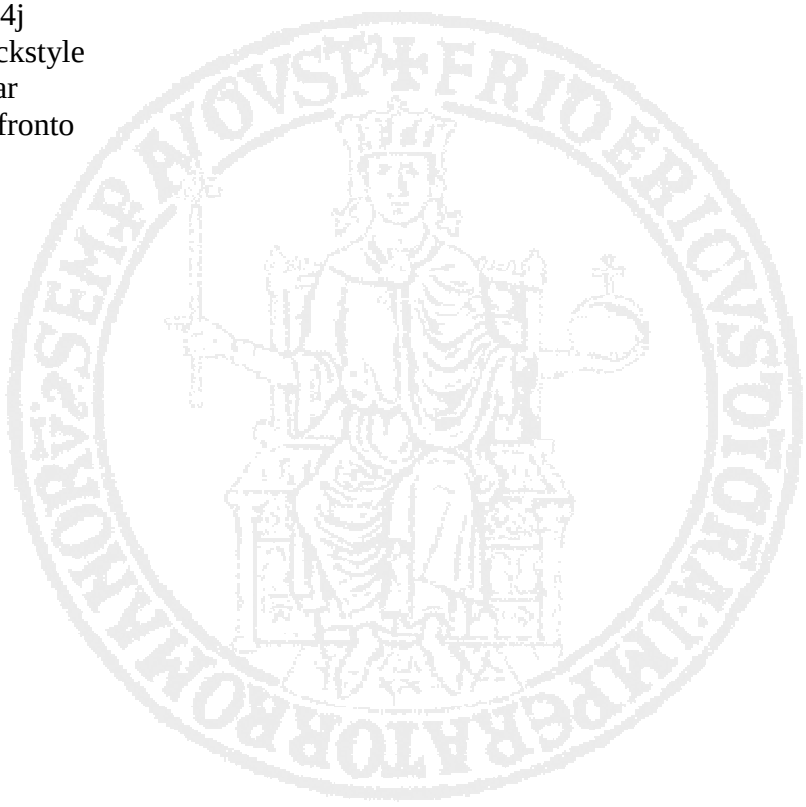
Anno Accademico 2011/2012

Candidato:

Gabriella Laurenza

matr. N46/000857

Indice



Introduzione

Se ci ritrovassimo coinvolti nella realizzazione di un progetto software, la qualità del codice sorgente non sarebbe certamente il primo problema al quale rivolgeremmo i nostri pensieri. Cercheremmo di rendere il progetto funzionante, di rispettare tutte le richieste espresse dal cliente, ma nel momento in cui andremmo a testare il nostro prodotto, ci accorgeremmo di quanti errori sono stati compiuti a discapito della qualità del nostro codice sorgente.
L'analisi del codice dovrebbe essere, invece, un processo da utilizzare durante la stesura del software, durante la fase di testing e anche dopo il completamento del prodotto. Qualsiasi momento è buono per andare a controllare se ciò che abbiamo scritto rispetta le leggi della "buona progettazione". La riuscita di un progetto software dipende dall'organizzazione e dalla definizione di standard e procedure che si adattano al particolare processo. La definizione della qualità di un prodotto software è un concetto complesso, ma nell'accezione comune, un prodotto è di qualità se rispetta le specifiche imposte.
Uno degli scopi fondamentali dell'Ingegneria del Software è diventato in poco tempo quello dell'analisi della qualità del codice sorgente, identificata anche come l'insieme di metriche e standard che il software d'interesse deve soddisfare per il suo corretto funzionamento. Nel corso degli anni, quando ci si è accorti che l'analisi statica di un software era uno dei passi fondamentali per la realizzazione di un buon progetto, sono state inventate tantissime metriche riguardanti gli aspetti più svariati.
Ci si è accorti, in seguito, che poiché i software più importanti sono nella maggior parte dei casi di grosse dimensioni, non si riesce sempre a rispettare tutte le regole imposte dagli standard. Spesso infatti, è obbligatorio un compromesso che ci permetta, anche in base alle caratteristiche richieste dal cliente, di realizzare un prodotto che possa rispettare la maggior parte delle buone norme imposte. Ci si è accorti quindi, che produrre del software senza aver prima stabilito dei requisiti completi e non ambigui, porta irrimediabilmente al fallimento e ad un prodotto di bassa qualità e dai costi molto elevati.
Per aiutare gli analisti nel raggiungimento di questi obiettivi, sono stati creati dei software appositi che consentono l'analisi automatica del codice sorgente. Questo è stato un grandissimo passo in avanti per l'Ingegneria del Software, che ha garantito uno sviluppo più immediato dei progetti senza però sorvolare sulle metriche qualitative del prodotto in esame.
Nel seguito, abbiamo studiato le metriche principali di alcuni strumenti open source per l'analisi del codice sorgente e confrontato i risultati per vedere come studi di caratteristiche differenti portano a risultati differenti.

Capitolo 1

Metriche di qualità

La domanda principale da farsi è: Che cos'è l'architettura software?
Per rispondere in modo esaustivo a questa domanda dovremmo spendere molte righe, ma per definire il concetto in maniera generale potremmo dire che essa è l'insieme dei componenti che realizzano il prodotto software. Il modo più immediato per analizzare correttamente l'architettura di un software è quello di dividerla in livelli. Il livello di cui noi ci interesseremo sarà quello più basso, nel quale si trovano i moduli e le loro interconnessioni: il dominio del "design pattern".

1.1 Standard ISO/IEC 9126 Engineering-Product Quality

Lo standard più utilizzato per la descrizione di un modello di qualità del software è lo standard ISO/IEC 9126. Esso è composto di 4 parti:

1- Quality Model: un insieme di caratteristiche di qualità che descrivono i fattori di qualità di un prodotto. Sono i fattori visti nel modello di Mc Call.
2- External Metrics: un insieme di metriche indirette attraverso le quali si può valutare la conformità di un prodotto in relazione all'ambiente operativo in cui si trova.
3- Internal Metrics: un insieme di metriche applicabili al software non eseguibile(come il codice sorgente), legate alle specifiche richieste dall'utente, che permettono di individuare eventuali problemi nelle metriche esterne, prima che il software sia eseguibile.
4- Quality In Use Metrics: un insieme di metriche, dipendenti dalle metriche esterne ed interne, legate alle caratteristiche positive che un utente riscontra nell'utilizzo del software.

Per determinare la qualità di un prodotto software si possono quindi osservare i tre punti di vista: qualità esterna, qualità interna e qualità in uso.
I tre punti di vista si influenzano a vicenda, ma è ovvio che un prodotto software percepito positivamente dall'utente è sintomo di una buona qualità di base del codice sorgente.
Una rappresentazione di questo modello, ci può essere data dal seguente grafico:

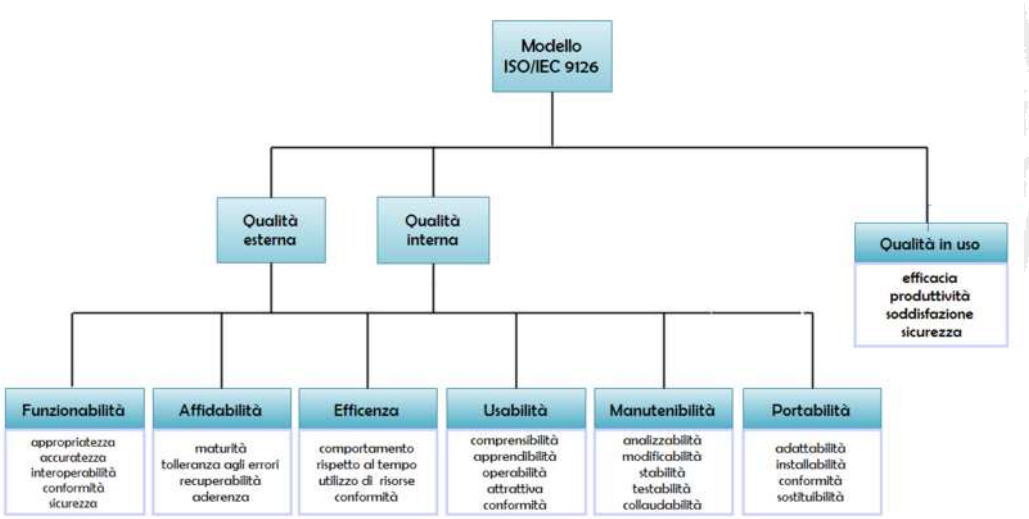


Fig. 1. Modello ISO/IEC 9126

1.2 "Bad smells" and "Best Practices"

L'analisi che noi faremo riguarderà codici sorgenti basati sulla programmazione "object oriented", che utilizzano il linguaggio Java e C++. Quando un progetto è all'inizio della stesura, gli sviluppatori hanno tutto sotto controllo, conoscono alla perfezione ogni parte del codice e riescono con tranquillità a gestire le connessioni fra i vari moduli. I problemi iniziano ad insorgere quando il progetto si evolve, diventa sempre più grande e di conseguenza più articolato. E' molto facile a questo punto che il software diventi difficile da testare, manutenere ed estendere.
Robert C. Martin descrive questo processo come:
"The software starts to rot like a piece of bad meat"
(" Il software inizia a marcire come un pezzo di carne andata a male").
Il problema principale si presenta quando più sviluppatori iniziano a contribuire alla stesura del codice, quando il lavoro su di esso è già avanzato. Sono stati definiti cinque sintomi primari che sono indice di questo degrado: quelli che comunemente definiamo "bad smells".

BAD SMELLS	DEFINIZIONI
Rigidità	Diventa complicato modificare il software, poiché piccole modifiche portano irrimediabilmente ad ulteriori modifiche
Fragilità	Cambiamenti a parti del software potrebbero provocare danni irreparabile in punti sconosciuti
Immobilità	Diventa difficile riutilizzare parti del software
Viscosità	E' molto più facile incappare in degli errori
Opacità	Il software è scritto in maniera tale da non essere facilmente compreso

Al contrario le "best practices" sono identificate da Flessibilità, Solidità, Mobilità, Fluidità e Trasparenza.

1.3 Dipendenze

Come abbiamo capito, la causa principale che genera il degrado di un software è la sua modifica. Il prodotto inizia a cambiare in modo completamente differente da come era stato stabilito inizialmente. Spesso questi cambiamenti avvengono molto velocemente e nella maggior parte dei casi dipendono da sviluppatori che non sono a conoscenza dei requisiti stabiliti per la stesura del progetto originale. I cambiamenti che causano il degrado del software, sono quelli che introducono "dipendenze". Le "bad smells" definite precedentemente sono sempre direttamente o indirettamente la causa che genera dipendenza fra i moduli. Le problematiche che derivano dalle dipendenze sono molteplici.
Prima fra tutte il fatto che un piccolo cambiamento in una parte del codice, provoca cambiamenti in tutte le parti che dipendono da esso. In questi casi molti leader preferiscono bloccare lo sviluppo del software per paura che ulteriori cambiamenti lo possano degradare oltremodo.
Un altro problema è che il software sarà difficile da riutilizzare quando le parti interessate al riuso, hanno dipendenze con altre parti che al contrario non interessano lo sviluppatore.
Ciò non vuol dire che all'interno di un codice sorgente non possano essere create dipendenze. Anzi, nella programmazione object oriented il loro uso è indispensabile. Per questo motivo, sono state introdotte delle tecniche di programmazione che consentono la creazione di "good dependency".
Una "buona dipendenza" si crea tra classi definite "stabili". Più la classe è stabile, più è buona la dipendenza creata. Una classe è stabile se possiede pochi legami con altre classi. Capiamo quindi che le classi più stabili sono quelle che non possiedono dipendenze: le cosiddette classi indipendenti. La stabilità di una classe aumenta anche quando da essa dipendono molte altre classi ma essa non dipende da nessuna. Questa caratteristica è chiamata responsabilità.

Le classi più stabili di tutte, sono quindi quelle Indipendenti e Responsabili.
Un altro problema riguarda il riuso del software. A causa delle dipendenze, è raro che una classe possa essere riutilizzata da sola. Questo perché nella maggior parte dei casi essa è legata ad altre classi di cui si serve per il suo utilizzo. Al fine di riutilizzare questa classe, bisogna riutilizzare tutto il gruppo di classi a cui essa è legata. Questo gruppo di classi è chiamato "Class Category".
Una "class category" è un insieme di classi molto coese tra loro che rispettano tre regole:

1- Le classi che appartengono ad una categoria sono "complici" di fronte a qualunque cambiamento. Ciò vuol dire che se una classe di una categoria è costretta a cambiare, lo faranno anche tutte le altre.
2- Le classi che appartengono ad una categoria vengono "riutilizzate" sempre insieme.
3- Le classi che appartengono ad una categoria raggiungono obiettivi comuni.

A questo punto ciò che vogliamo gestire al meglio sono le dipendenze tra categorie, e non le dipendenze all'interno di una stessa categoria. Questo perché in una categoria, le classi sono già molto coese tra di loro, vengono riutilizzate tutte allo stesso momento, e quindi la loro interdipendenza non danneggia più di tanto il prodotto software. Dobbiamo quindi applicare i concetti di "Indipendenza", "Responsabilità" e "Stabilità" alle categorie, e non più alle classi. Le dipendenze tra categorie stabili sono le "good dependency".

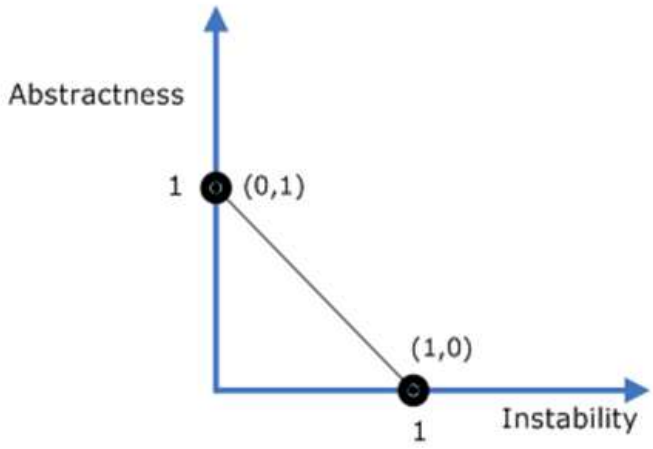
1.4 Metriche di codice

L'Indipendenza, la Responsabilità e la Stabilità possono essere misurate, contando le dipendenze che una categoria possiede. Per fare ciò individuiamo una serie di "metriche", che rappresentano delle caratteristiche concrete e misurabili matematicamente di un prodotto software.
Identifichiamo tre metriche:

- Ca: Afferent Coupling. Numero di classi al di fuori di una categoria che dipendono dalle classi che si trovano all'interno.
- Ce: Efferent Coupling. Numero di classi all'interno di una categoria che dipendono da classi esterne.
- I: Instability: (Ce/(Ca+Ce)). Questa metrica ha valori compresi nell'intervallo [0.1].
  I=0 indica la massima stabilità e I=1 indica la massima instabilità.

La prima cosa da fare, è stabilire una relazione tra la Stabilità(I) e l'Astrazione(A). Definiamo l'astrazione come A=Na/Nc dove Na è il numero di classi astratte e Nc è il numero di classi concrete. Creiamo quindi un grafico con A sull'asse delle ordinate e I sull'asse delle ascisse. I due punti che rappresentano le condizioni migliori per una categoria, sono il punto (0,1) che rappresenta massima astrazione e massima stabilità, e il punto (1,0) che rappresenta massima concretezza e massima instabilità. Ovviamente, non tutte le categorie ricadono in questi due punti.
Consideriamo una categoria con A=0 e I=0. Questo comporta massima concretezza e massima stabilità, ma una categoria con queste caratteristiche non è ottimale perché risulta rigida. Non può essere estesa perché non è astratta e non può essere modificata facilmente a causa della sua stabilità.
Consideriamo una categoria con A=1 e I=1. Questo comporta massima astrazione e massima instabilità. Anche questo tipo di categoria non è considerata positivamente perché diventa rigida. A causa dell'instabilità infatti, l'astrazione non può essere utilizzata per estendere la categoria.
Se invece consideriamo una categoria con A=0.5 e I=0.5, questa categoria è detta "equilibrata". La stabilità è in equilibrio con l'astrazione, infatti una categoria con queste caratteristiche è parzialmente estendibile, essendo parzialmente astratta, e le sue estensioni non sono soggette ad instabilità, essendo parzialmente stabili.

Considerando il grafico A-I, possiamo tracciare una linea che unisce i punti (0,1) e (1,0). Questa linea, chiamata “main sequence”, indica le categorie in cui c’è equilibrio tra astrazione e stabilità.



Le categorie che si trovano sulla “main sequence” non sono né troppo astratte, né troppo instabili. Esse hanno il giusto numero di classi concrete e astratte in proporzione alle loro dipendenze afferenti ed efferenti. Le categorie con caratteristiche ideali dovrebbero trovarsi su i punti della “main sequence” che incontrano gli assi, ma poiché questo accade molto raramente, possiamo accontentarci di averle su un punto qualsiasi della linea.

Fig. 2. Main sequence

Le metriche software possono essere utilizzate per:

- stimare il budget per il progetto e la codifica
- stimare la produttività individuale e la qualità
- stimare la produttività del progetto e la qualità
- stimare la qualità del software

Esistono tantissime metriche generali che ci permettono di capire quali sono i punti su cui focalizzare la nostra attenzione per confrontare la qualità del codice sorgente. Possiamo dividere le principali metriche in cinque categorie secondo le caratteristiche su cui esse agiscono: Size, Tests, Duplication, Design e Rules.

METRICHE “Size”	DEFINIZIONE
Physical lines	Numero di ritorni a capo. Deve essere un valore basso.
Comment lines	Numero di linee di commento. Deve essere un valore basso.
Lines of code	Numero di linee di codice. Il progetto che possiede meno righe di codice ha un design superiore e richiede una manutenzione minore.
Packages	Numero di pacchetti.
Classes	Numero di classi. Confrontando progetti con le stesse funzionalità, il progetto che possiede più classi è quello che realizza l’astrazione migliore.
Files	Numero di files analizzati.
Directories	Numero di directories analizzate.
Methods	Numero di metodi.

METRICHE “Tests”	DEFINIZIONE
Unit tests	Numero di test.
Unit tests duration	Tempo impiegato per effettuare un test.
Unit test error	Numero di test falliti.
Line coverage	Linee coperte dal test.
Branch coverage	Rami di un’espressione booleana coperti(true e false).

METRICHE “Duplication”	DEFINIZIONE
Duplicated lines	Numero di linee duplicate.

METRICHE “Rules”	DEFINIZIONE
Violations	Numero di regole violate
Weighted violations	Somma delle violazioni calcolate in base ad un coefficiente di priorità
Rules compliance index	E’ calcolato come [100 - (weighted_violations / nloc * 100)]

METRICHE “Design”	DEFINIZIONE
Depth of inheritance tree	E’ la posizione della classe all’interno dell’albero di eredità. Si consiglia di mantenere il numero dei livelli da attraversare per giungere la radice, al di sotto del 5.
Response for class	E’ il numero di tutti i metodi implementati da una classe più il numero di metodi accessibili tramite un oggetto della classe. Si consiglia di mantenere questo numero basso. Maggiore è la RFC, maggiore è lo sforzo dato per apportare modifiche.
Afferent couplings	E’ il numero delle classi che utilizzano una classe.
Efferent couplings	E’ il numero delle classi che una classe utilizza.
Lack of cohesion of methods	Essa misura il numero di componenti collegati ad una classe. Un valore basso indica che il codice è semplice da comprendere e riutilizzare. Un valore alto suggerisce di dividere la classe in classi più piccole. Si consiglia di effettuare un refactoring, se questo valore diventa maggiore di 2.
Weighted methods per class	E’ il numero medio di metodi contenuti in una classe. Si consiglia di mantenere tale numero al di sotto del valore 14.
Cyclomatic Complexity (CC)	Essa misura il numero di cammini linearmente indipendenti all’interno di una parte di codice. Più il numero è alto, più lo sforzo per effettuare un testing del codice sarà alto. Si consiglia di mantenere questo numero al di sotto del valore 10.
Distance	Rappresenta la distanza dalla linea ideale A+I=1. Identifica quanto una categoria è lontana dal caso ideale. Minore è la distanza del software dalla linea, maggiore è la sua qualità.
Tangle	

Capitolo 2

Analisi del codice sorgente

Abbiamo visto come qualsiasi cambiamento apportato ad un progetto potrebbe facilmente compromettere la qualità del suo codice sorgente. Poiché è difficile tenere sotto controllo lo sviluppo del codice attraverso semplici analisi visive, avere a disposizione dei software che effettuano automaticamente dei test e delle analisi approfondite è un grande aiuto per il team di sviluppo.

2.1 Strumenti per l'analisi

Gli strumenti di cui ci occuperemo sono tutti open source, cioè software il cui codice sorgente è messo a disposizione di tutti permettendone lo studio e la modifica, se ritenuta necessaria. Tali software effettuano una scansione del codice e ne individuano gli errori in maniera automatica con un elevato grado di fiducia. Essi consentono, attraverso numerosi strumenti, di analizzare il codice in base ad un vasto numero di metriche e di conseguenza garantiscono una copertura quasi totale degli aspetti principali riguardanti il grado di qualità del codice. Nella maggior parte dei casi essi vengono utilizzati come supporto all'analista per individuare delle falle all'interno del codice ed indirizzarlo verso modifiche appropriate. Alcuni di questi strumenti agiscono all'interno degli IDE(ambienti di sviluppo integrato), il che permette di effettuare l'analisi durante la fase di sviluppo. Questo è effettivamente il momento migliore per consentire al team di tornare indietro(azione di feedback) e aggiustare parti di codice che potrebbero, nello sviluppo futuro, provocare danni irreparabili. Tra tutti gli strumenti messi a disposizione, quelli che utilizzeremo per l'analisi sono:

- Stan4j
- Sonar
- Checkstyle

Utilizziamo Stan4j e Checkstyle come plug-in dell'ambiente di sviluppo Eclipse, mentre l'analisi con Sonar prevede l'utilizzo di 3 elementi: il database MySQL che contiene i risultati dell'analisi, il web server Apache Tomcat che serve per esplorare le varie metriche, e il client Maven che consente di caricare il progetto desiderato.

2.2 Esempio 1: analisi di un piccolo codice

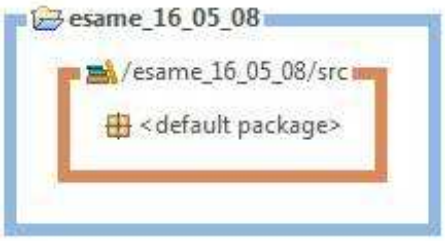
Partiremo da un codice di piccole dimensioni per esplorare nel dettaglio le caratteristiche dei software utilizzati e anche per confrontarlo, in seguito, con progetti più consistenti e notare come la grandezza di un prodotto possa influire pesantemente sulla sua qualità. Analizziamo un semplice programma che realizza un'applicazione Client/Server attraverso l'utilizzo delle socket.

2.2.1 Stan4j

- Analisi delle dipendenze

Stan4j permette di effettuare un'analisi visiva del progetto nella sua interezza, e di esplorarne i vari moduli e pacchetti presenti per vedere di cosa essi sono composti e quali e quante dipendenze possiedono.

- Composition View



La Composition View di questo progetto ci mostra che esso è composto semplicemente da un pacchetto, contenuto all'interno dell'unica libreria presente.

Fig. 3. Composition View in Stan4j del <default package>
del progetto "esame_16_05_08"

Ogni modulo può essere espanso per analizzare la sua struttura interna e le dipendenze fra i vari componenti.

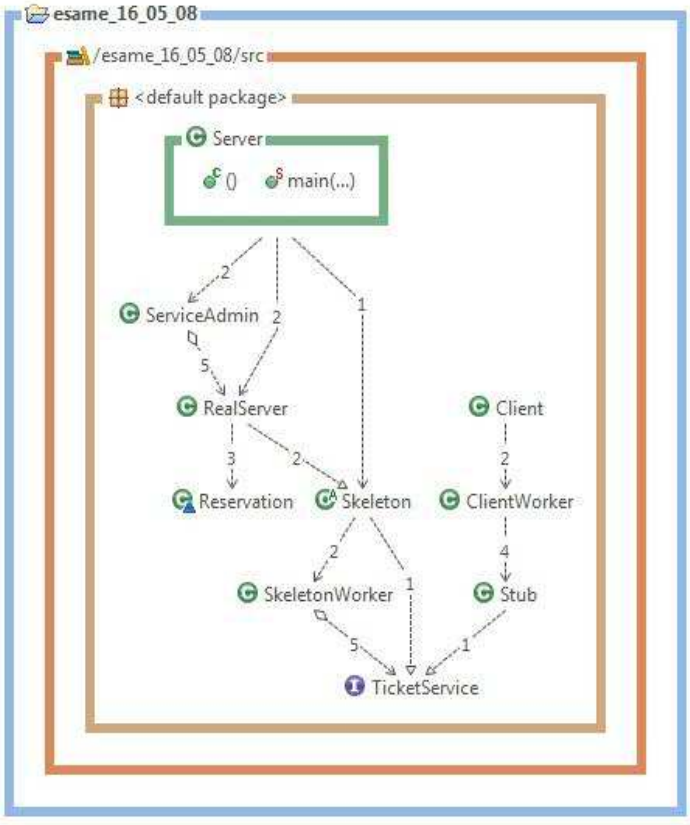


Fig. 4. Composition View espansa in Stan4j del <default package>
del progetto "esame_16_05_08"

- Coupling View

La Couplings View permette di analizzare un singolo elemento e di leggerne le dipendenze in entrata e in uscita. I numeri sulle linee esprimono il numero di volte che una classe ne richiama un'altra, indicata con la freccetta.

Fig. 5. Coupling View in Stan4j della classe 'Skeleton' del progetto "esame_16_05_08"

Cliccando sulla dipendenza RealServer ---> Skeleton, Stan4j ci mostra le 2 volte in cui RealServer utilizza Skeleton.

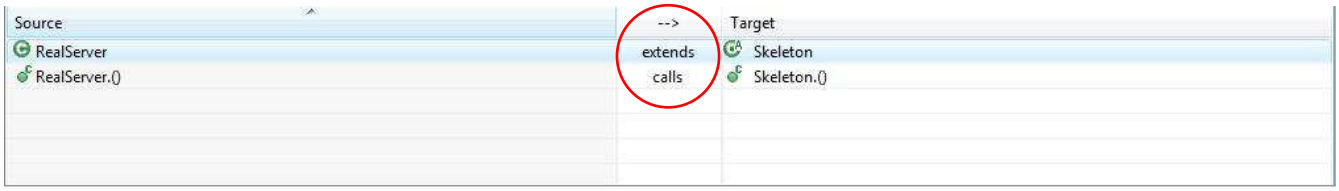


Fig. 6. Dipendenze in Stan4j della classe 'RealServer' del progetto "esame_16_05_08"

- Map

La mappa ci consente di individuare le dipendenze attraverso un approccio visivo legato ai colori. Il codice rappresentato da uno dei blocchi, dipende dal codice rappresentato dal blocco sottostante e ha una dipendenza bidirezionale dai blocchi collocati sul suo stesso livello.

Selezionando il pacchetto in esame, la divisione sarà:

- Il pacchetto selezionato (color ambra)
- Gli elementi richiesti dal pacchetto selezionato (color verde)
- Gli elementi che dipendono dal pacchetto selezionato (color blu)

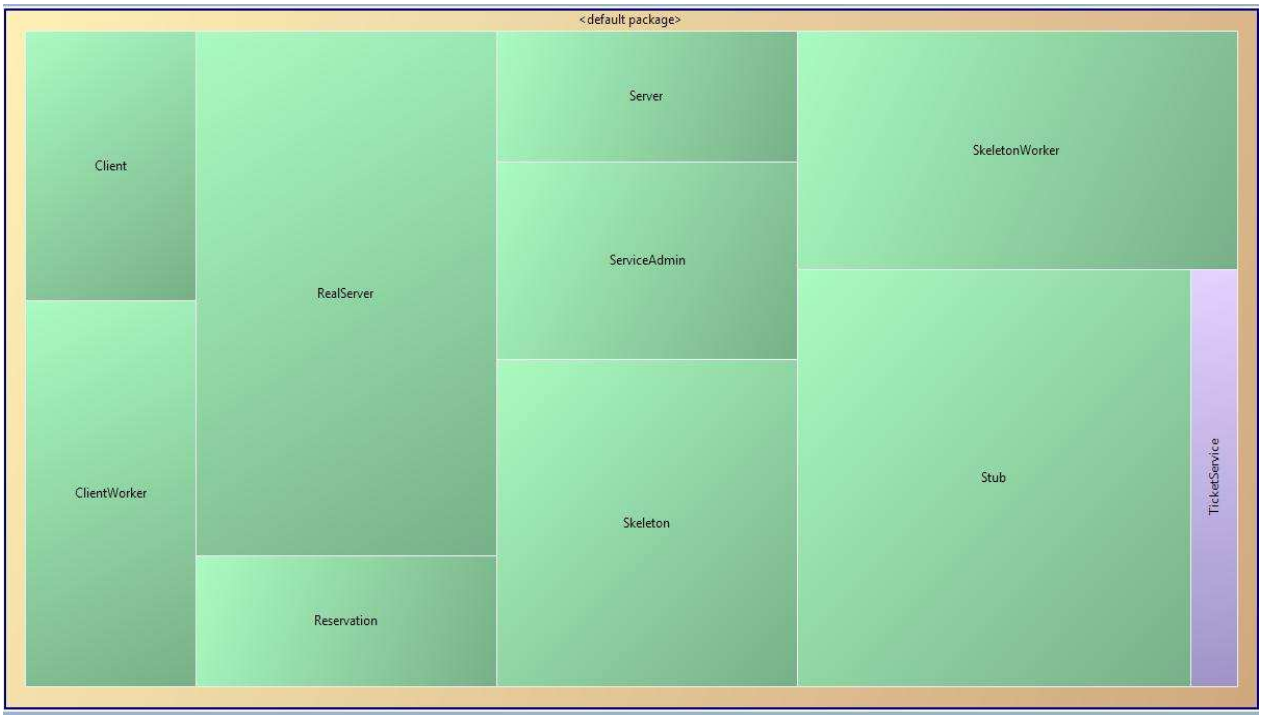


Fig. 7. Mappa in Stan4j del <default package> del progetto “esame_16_05_08”

- Metriche

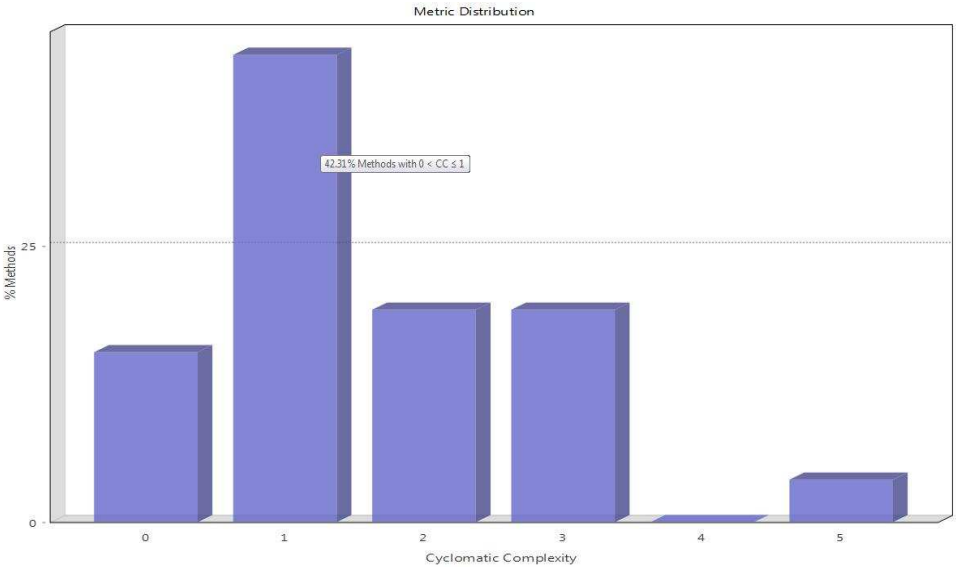
Stan4j copre un elevato numero di metriche. Le più interessanti per la nostra analisi sono quelle che ci consentono di individuare eventuali violazioni. Stan4j attraverso il suo report ci mostra, in questo progetto, l'assenza di violazioni. Ci sono due modi per rappresentare graficamente una metrica: il rating e la distribution.

- Rating

Il rating individua il numero di righe di codice entro le quali la metrica in esame non altera la qualità del codice sorgente. Per esempio, riguardo la complessità ciclomatica, le linee di codice dal 10 al 15 hanno un valore perfetto (colore verde), mentre la linea 30 rappresenta il caso peggiore (colore rosso). Con questo intendiamo dire che la complessità ciclomatica, come qualsiasi altra metrica, ha bisogno di essere espressa attraverso dei valori partizionati che ci dicono entro quali soglie essa può essere considerata accettabile e oltre quali soglie è consigliato effettuare un refactoring.

Fig. 8. Rating delle metriche in Stan4j del progetto “esame_16_05_08”

- Distribution



Per ogni metrica Stan4j ci mostra anche la distribuzione in base ai vari metodi. Riguardo questo progetto notiamo, per esempio, che la colonna più alta indica che il 42.31% dei metodi ha complessità ciclomatica compresa tra 0 e 1.

Fig. 9. Distribuzione della complessità ciclomatica in Stan4j del progetto "esame_16_05_08"

- Distance

Un ultimo elemento interessante è rappresentato dalla metrica distance di Robert C.Martin che indica la distanza che il progetto, o i suoi elementi, posseggono dalla retta ideale Instabilità-Astrazione.
In questo caso il punto di distanza dalla retta ideale è dato dalle coordinate [1,0.2]. Il valore di distanza è sufficientemente basso.

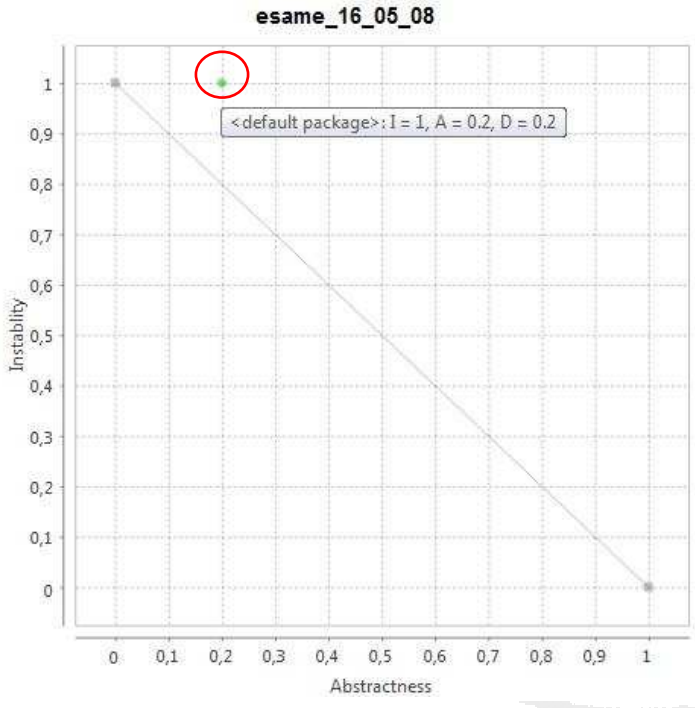


Fig. 10. Distanza dalla main sequence in Stan4j del progetto "esame_16_05_08"

2.2.2 Checkstyle

Analizzando il codice precedente con un altro strumento, alcune caratteristiche cambiano. Notiamo infatti che Checkstyle, al contrario di Stan4j, individua una serie di violazioni che non erano state riscontrate prima. Questo perchè Checkstyle aderisce a delle regole di codifica diverse e molto più rigide.

- Checkstyle Violation Chart

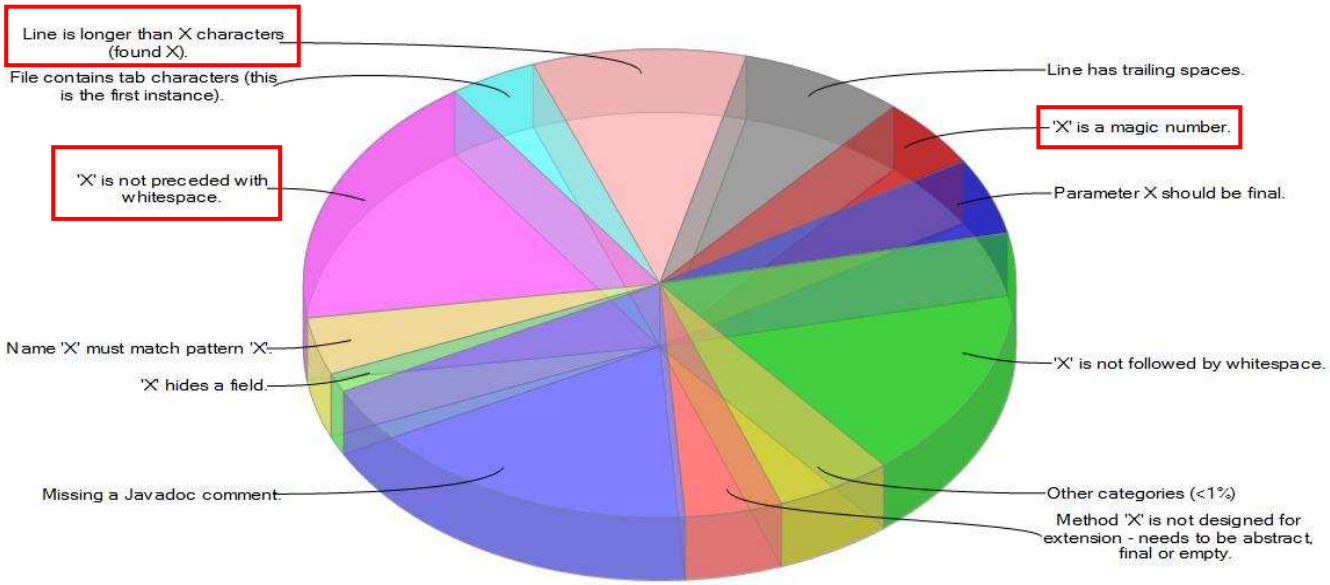
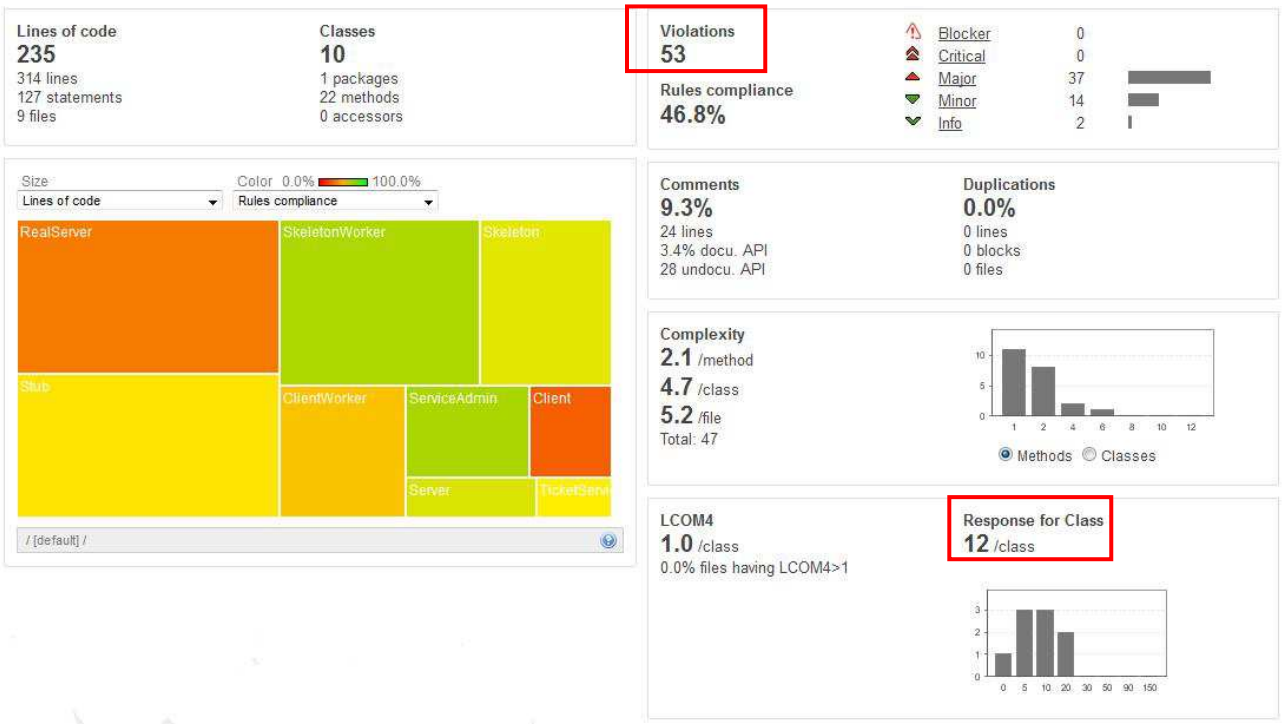


Fig. 11. Grafico delle violazioni in Checkstyle del progetto “esame_16_05_08”

Ad esempio, Checkstyle ci fa notare che nel progetto in esame i token presenti non sono preceduti dallo spazio bianco, oppure che sono presenti linee di codice con più di 80 caratteri, o ancora che sono presenti dei numeri, che Checkstyle definisce “magic number”, che non sono stati definiti come costanti. Capiamo quindi che la maggior parte di queste violazioni guarda più alla forma del progetto che alla qualità. Checkstyle cerca di rendere il codice sorgente il più chiaro possibile, ma nella maggior parte dei casi le violazioni individuate vengono sorvolate. D'altronde in un progetto di vasta entità sarebbe inutile concentrarsi sul numero di caratteri che ogni linea possiede, piuttosto che su metriche che inficiano in modo maggiormente negativo sulla qualità del codice sorgente.

2.2.3 Sonar

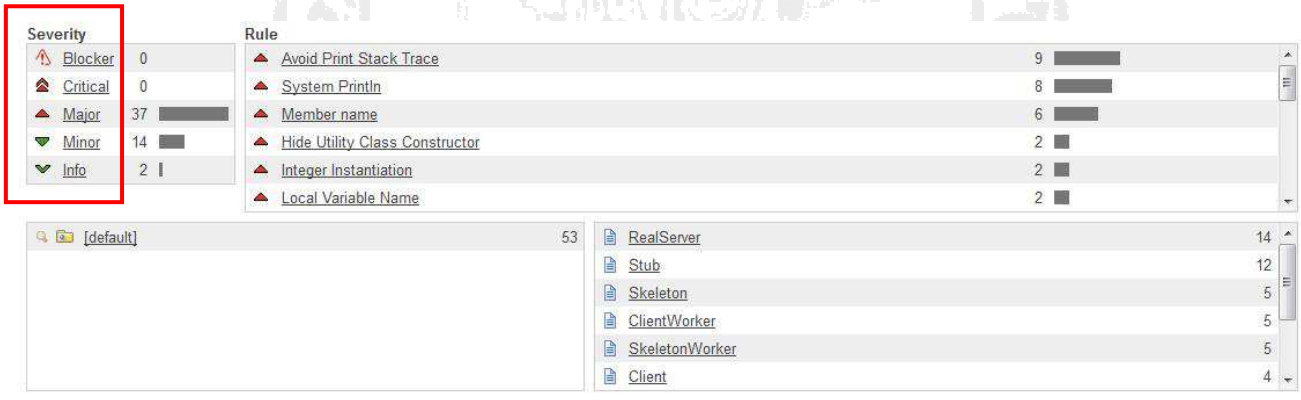
In Sonar la dashboard consente di visualizzare le metriche che più ci interessano per avere una visione completa dall'analisi del progetto. Possiamo configurare la dashboard a nostro piacimento attraverso l'opzione configure widgets. Ogni metrica, poi, può essere espansa per analizzarla più nel dettaglio.



- Treemap

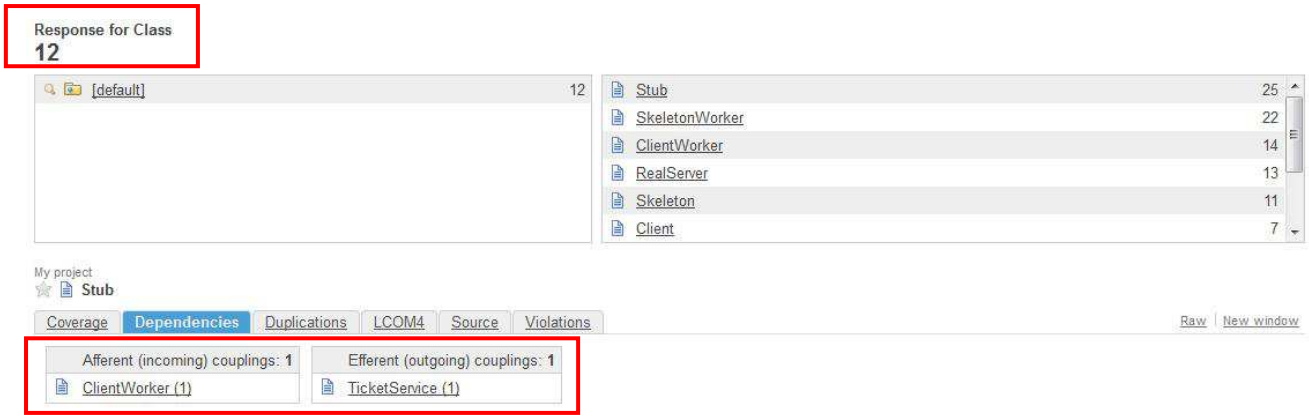
La treemap rappresenta attraverso una serie di colori la percentuale di metrica selezionata per ogni pacchetto. In questo progetto abbiamo chiesto di visualizzare il numero di linee di codice e la percentuale di regole di conformità per ogni classe del pacchetto di default.

- Violazioni



Sonar mostra tutte le violazioni in base alla loro criticità all'interno del progetto, e consente di navigare attraverso il codice per individuare il punto esatto in cui la metrica è stata violata. Le violazioni individuate in questo progetto sono 53 ma nessuna di esse appartiene né alla categoria “blocker” né a quella “critical”.

- RFC e Dependencies



Sonar ci mostra per tutte le classi le dipendenze, le duplicazioni, le violazioni e le metriche LCOM4 e RFC. A differenza di Stan4j, Sonar ci consente di analizzare se il tipo di accoppiamento è Afferente o Efferente. In questo caso notiamo la dipendenza tra le classi 'ClientWorker' e 'TicketService', mentre il Response for Class è pari a 12, un valore sufficientemente basso.

2.3 Esempio 2: Jakarta DBCP

Analizziamo adesso progetti più vasti e confrontiamo i risultati ottenuti dai diversi software utilizzati per la loro analisi. Jakarta DBCP è uno dei componenti del progetto Commons del gruppo Apache Jakarta. Questo gestore si occupa di aprire una serie di connessioni verso un database. Ogni volta che altri oggetti hanno bisogno di accedere al database, essi richiedono la connessione al gestore che gli fornisce il libero accesso.

2.3.1 Stan4j

- Grafico delle dipendenze tra le librerie

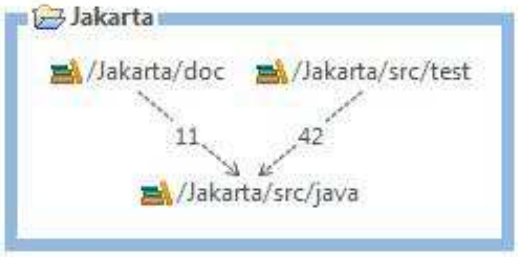


Fig. 12. Composition View in Stan4j del progetto "Jakarta"

- Violazioni

Notiamo subito che, essendo il progetto molto più esteso rispetto al precedente, sono presenti un elevato numero di violazioni. Il report generato ci mostra le più importanti.

Top Violations (20 of 58)

Artifact	Metric	Value
dbcp2	D	-0.96
dbcp2	Units	51
dbcp2	Fat	79
dbcp2.DelegatingResultSet	WMC	403
dbcp2.DelegatingResultSet	ELOC	1161
dbcp2.DelegatingResultSet	Fat	390
dbcp2.DelegatingResultSet	Methods	198
dbcp2.DelegatingDatabaseMetaData	WMC	361
dbcp2.DelegatingDatabaseMetaData	ELOC	1082
dbcp2.DelegatingDatabaseMetaData	Fat	376
dbcp2.DelegatingDatabaseMetaData	Methods	180
dbcp2.DelegatingCallableStatement	ELOC	747
dbcp2.DelegatingCallableStatement	WMC	227
dbcp2.DelegatingCallableStatement	Methods	114
dbcp2.TesterResultSet	ELOC	662
dbcp2.TesterResultSet	Methods	193
dbcp2.DelegatingConnection	ELOC	497
dbcp2.TesterResultSet	Fat	166
dbcp2.DelegatingConnection	Fat	181
dbcp2.DelegatingResultSet	RFC	202

Fig. 13. Violazioni principali in Stan4j del progetto "Jakarta"

La prima violazione che andiamo a studiare è la distanza del pacchetto 'dbcp2' dalla main sequence.

- Distanza

In figura vediamo la distanza che ogni pacchetto, rappresentato da un pallino colorato, ha dalla main sequence. Il pacchetto ‘dbcp2’ è rappresentato dal pallino giallo, e dista dalla retta ‘-0.96’. Questo indica che l'elemento in questione ha un'elevata stabilità e una bassa astrazione. Il pacchetto è troppo rigido, quindi difficile da modificare, e non può essere esteso a causa della sua concretezza. Questo valore è infatti inserito nella lista delle violazioni riscontrate.

Per ridurre la distanza di questo pacchetto dalla main sequence, si dovrebbe aumentare l’astrazione del progetto, sintetizzando il più possibile i modelli concettuali ed escludendo i dettagli inutili.

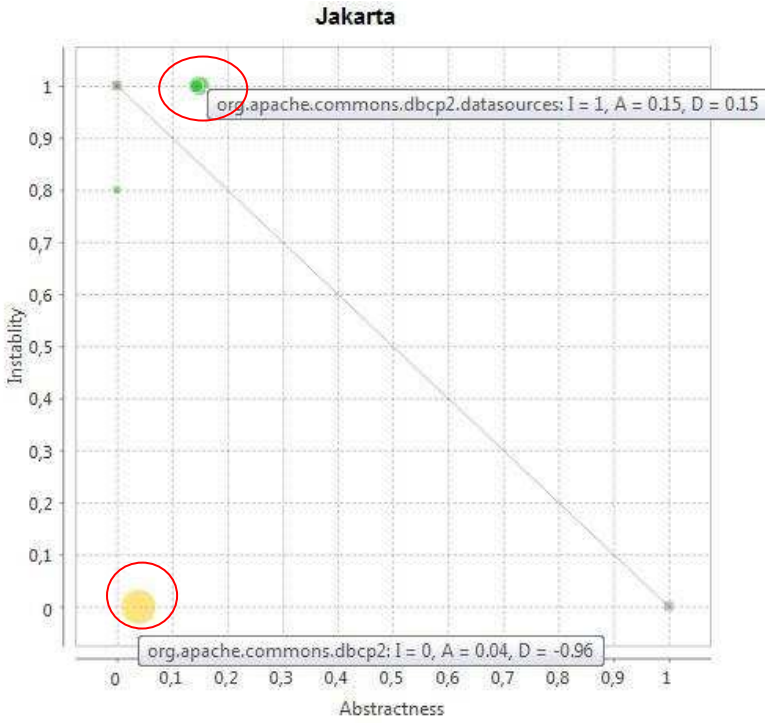
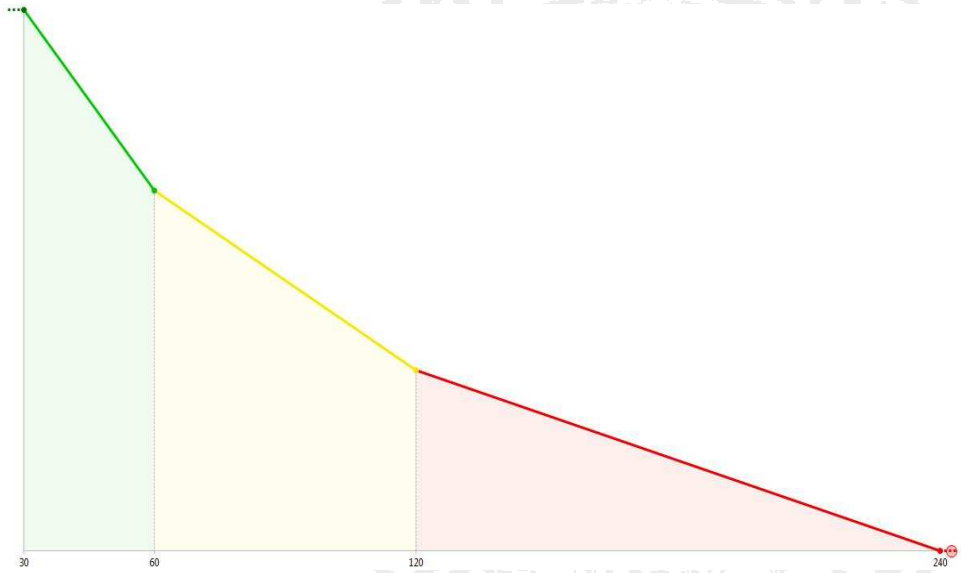


Fig. 14. Distanza dalla main sequence in Stan4j del progetto “Jakarta”

Anche il pacchetto 'datasources' mantiene una certa distanza dalla main sequence, ma il valore ‘0.15’ non è da considerarsi una violazione.

- Fat



L'altra metrica osservata è Fat, anch’essa legata al grado di complessità che una libreria, un pacchetto o una classe possiede. In questo caso la figura, ci mostra la complessità della classe 'DelegatingResultSet', il cui fat è pari a ‘390’.

Fig. 15. Fat in Stan4j della classe ‘DelegatinfResultSet’ del progetto “Jakarta”

Questa metrica è calcolata come il numero di archi presenti nel grafo delle dipendenze tra tutti i metodi della classe. Il numero è in questo caso molto elevato, e potrebbe comportare dei problemi alla funzionalità del software.

L'obiettivo di questa metrica è quello di limitare il più possibile la dimensione del codice riducendo il numero dei metodi presenti. Non esiste un valore standard per questa metrica, ma tutto dev'essere commensurato all'entità del progetto.

- Pollution Chart

La presenza di tutte queste violazioni, ci consente di ottenere la Pollution Chart, che invece non avevamo riscontrato nel primo esempio.

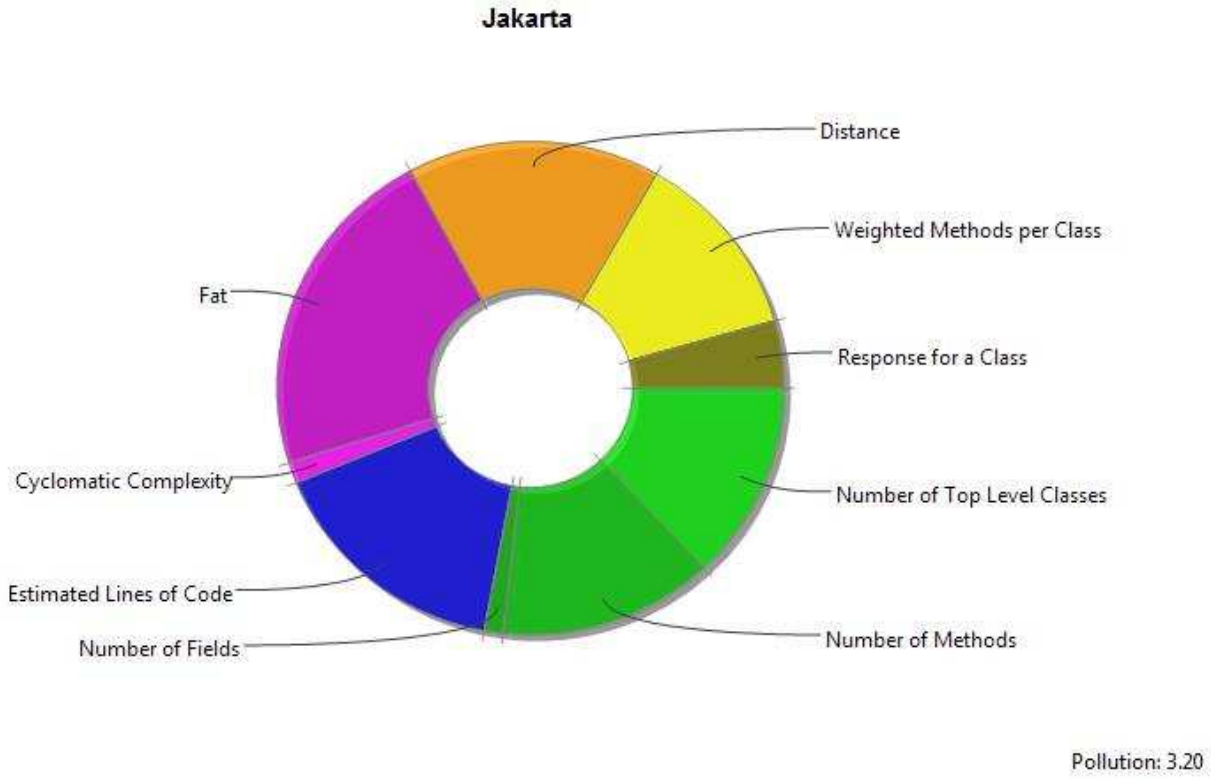


Fig. 16. Pollution Chart in Stan4j del progetto "Jakarta"

La Pollution Chart ci indica il grado di "inquinamento" del codice e la percentuale in base a questo grado, di tutte le metriche violate.

2.3.2 Sonar

- Dashboard

Sonar attraverso la sua dashboard, ci permette di configurare la pagina principale del server in modo tale da avere una visuale completa sulle metriche di nostro interesse.

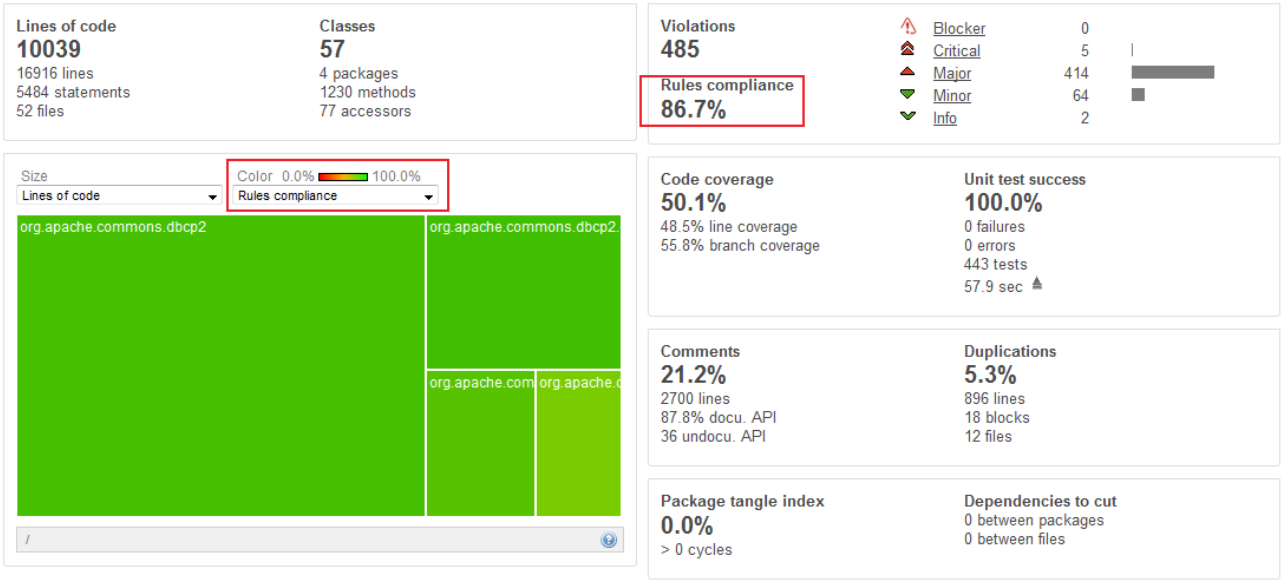


Fig. 17. Dashboard in Sonar del progetto “Jakarta”

- Violazioni

Sonar presenta 485 violazioni, sempre legate al pacchetto ‘dbcp2’, e le divide in 5 gruppi: Blocker, Critical, Major, Minor e Info. Come possiamo intuire facilmente dal nome, ogni gruppo corrisponde ad un valore minore o maggiore di gravità della metrica violata. In rosso abbiamo evidenziato le regole di conformità (RCI), mostrate sia in base alla mappa dei colori per ogni pacchetto, sia in percentuale.

- Complessità Ciclomatica

Esplorando le violazioni più incisive nel codice (Major) possiamo leggere direttamente le linee di codice alle quali la violazione si riferisce. La Complessità Ciclomatica della classe 'BasicDataSource' è di 14, quando sarebbe richiesto un minimo di 10. Tale valore dipende dal numero di cammini linearmente indipendenti attraverso il grafo della classe in questione.

Considerato 10 il limite massimo, per abbassare questo valore si dovrebbe aumentare la modularità del codice.

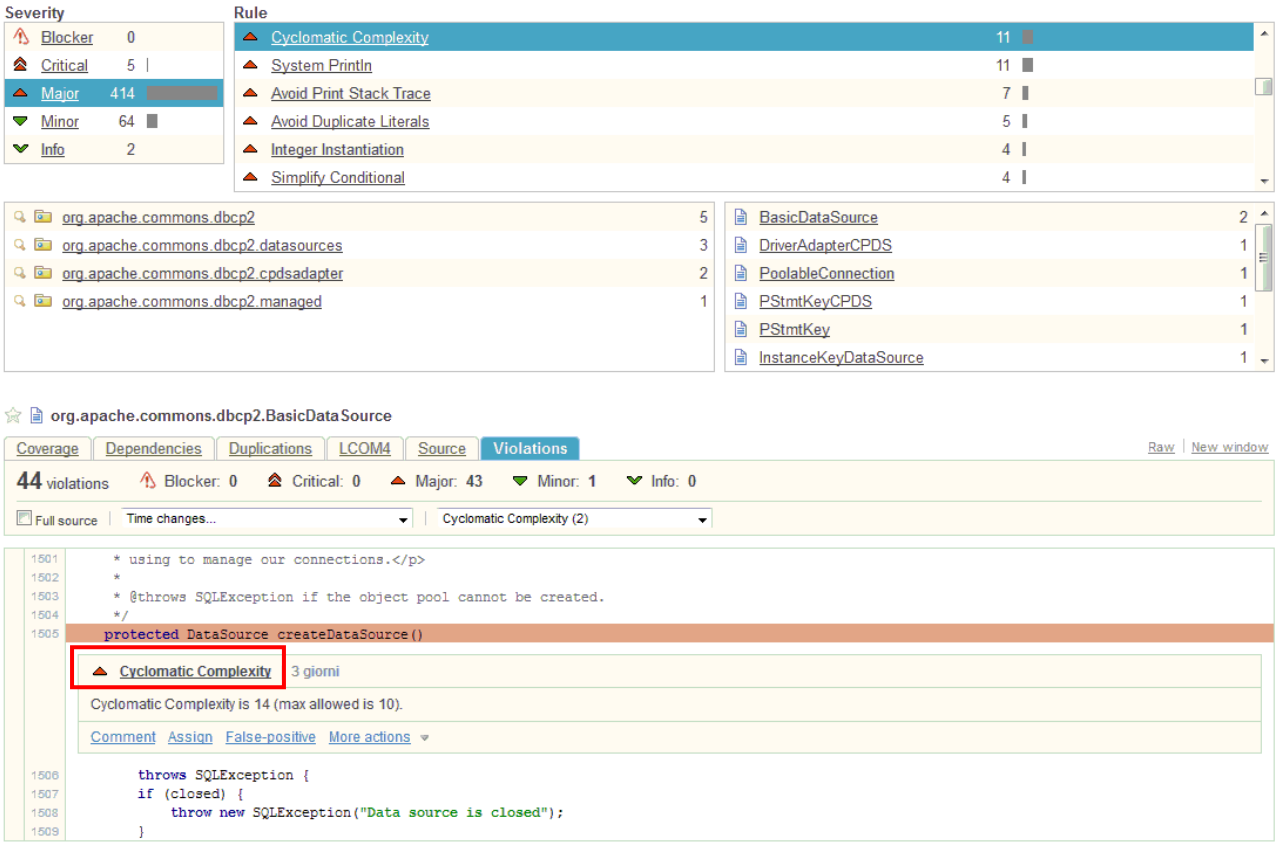


Fig. 18. Complessità Ciclomatica in Sonar della classe 'BasicDataSource' del progetto "Jakarta"

- Violazioni critiche

Tra le violazioni critiche di Sonar ne riscontriamo una fondamentale: 'Security-Array is stored directly' che riguarda il metodo 'PstmtKeyCPDS'. Questa violazioni evidenzia che se la funzione chiamante modifica l'array in esame, in questo caso 'columnNames[ ]', la modifica verrà apportata anche all'array memorizzato nell'oggetto (e quindi all'oggetto stesso).

La soluzione, chiamata "copia difensiva" sarebbe quella di creare una copia dell'oggetto quando esso viene passato al chiamante. In questo caso un'eventuale modifica dell'array, non cambierà anche l'oggetto riferito.

Fig. 19 Violazione critica 'Array is stored directly' in Sonar del metodo 'PsmtKeyCPDS' del progetto "Jakarta"

- Duplicated Lines

Osserviamo, attraverso la Treemap, la percentuale di linee duplicate per ogni pacchetto esaminato. I pacchetti colorati di grigio non presentano duplicazioni.

Dall'analisi del grafico, il numero di linee duplicate non è tale da incidere sulla qualità del codice.

Fig. 20. Treemap delle linee duplicate in Sonar del progetto “Jakarta”

- LCOM e RFC

Anche queste metriche si riferiscono al grado di complessità del progetto, e individuano il valore di coesione dei metodi di ogni classe. Nel progetto in esame il valore LCOM4 è sufficientemente basso. Infatti, il valore migliore per questa metrica è di 1, e nel nostro progetto solo il 25,0% dei file superano tale soglia. Per la metrica RFC, 48 è il numero medio per ogni classe, dei metodi presenti e di quelli richiamati. La metrica RFC non dovrebbe superare il valore 50, anche se è accettabile un limite massimo di 100.

In questo caso quindi entrambi i valori sono da considerarsi sotto soglia.

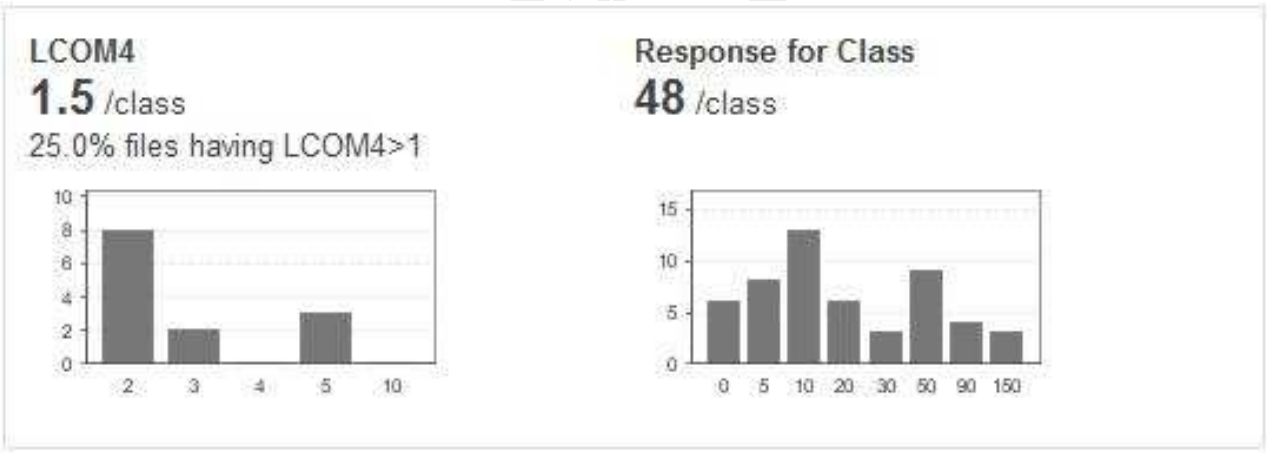


Fig. 21. Metriche LCOM4 e RFC in Sonar del progetto “Jakarta”

2.3.3 Checkstyle

- Checkstyle violation chart

Checkstyle, attraverso il suo grafico riscontra un numero elevatissimo di violazioni, addirittura pari a 10502. Questo evidenzia il fatto, messo in risalto precedentemente, che le metriche di Checkstyle sono diverse da quelle di Stan4j e Sonar, e nella maggior parte dei casi effettuano analisi troppo dettagliate.

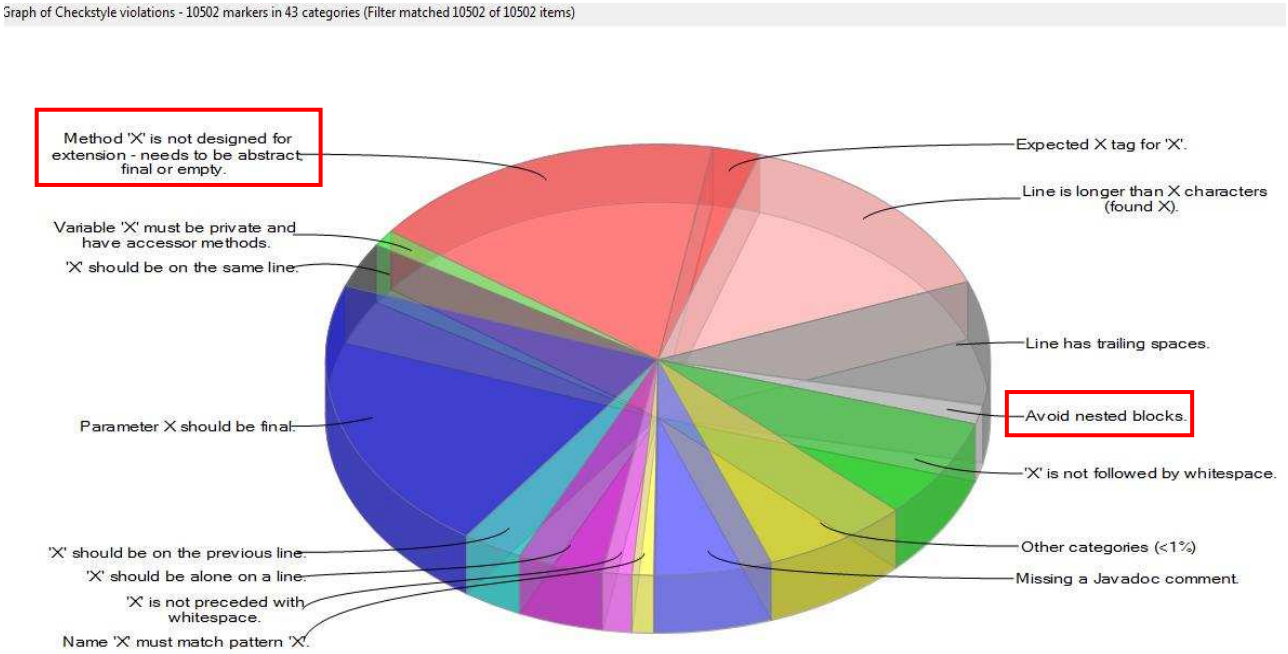


Fig. 22. Grafico delle violazioni in Checkstyle del progetto “Jakarta”

Le violazioni evidenziate in rosso sono le uniche da tenere più sotto controllo.

- Method ‘X’ is not designed for extension.

Questa metrica controlla che le classi siano progettate correttamente per essere estese. La regola da seguire è che le superclassi dovrebbero presentare un metodo vuoto, la cui implementazione sarà compito delle sue sottoclassi. I metodi di tali superclassi devono essere astratti, finali o possedere un’implementazione nulla. Anche se questa struttura potrebbe limitare la flessibilità di una superclasse, ciò prevede che le sottoclassi non potranno mai “corrompere” una superclasse dimenticandosi di richiamarla.

- Avoid nested blocks

La presenza di blocchi innestati è dovuta spesso al processo di debugging. Essi provocano confusione ai progettisti durante la lettura del codice, per questo motivo, se non ci si riferisce ad un blocco ‘switch’, è sempre consigliato “pulire” il codice dalla presenza di tali blocchi.

2.3.4 Confronto

Effettuiamo un confronto delle metriche principali, calcolate dai 3 strumenti utilizzati.

- Violazioni

Nel pacchetto 'dbcp2' viene individuata, da Checkstyle e da Sonar, la violazione critica “Equals Hash Code”. Essa riconosce la mancanza di un metodo 'hashCode' nonostante la presenza del metodo 'equals'. Quest'ultimo serve per controllare l'uguaglianza tra due oggetti ed è legato al metodo 'hashCode' in modo che, se uno viene sovrascritto(override), deve essere sovrascritto di conseguenza anche l'altro. La sovrascrizione infatti si basa proprio sul codice hash, richiamato dal metodo 'hashCode'.
Checkstyle, e Sonar individuano 3 punti in cui questa caratteristica non viene rispettata.

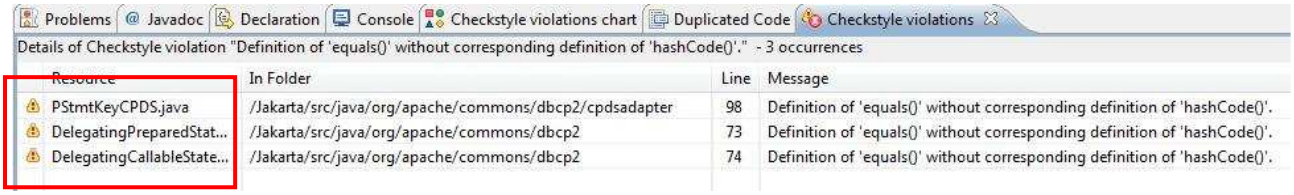


Fig. 23. Violazioni in Checkstyle del progetto “Jakarta”

Di seguito mostriamo, con Sonar, la violazione precedente nella classe 'DelegatingPreparedStatement'.

Fig. 24. Violazione ‘Equals Hash Code’ in Sonar della classe ‘DelegatingPreparedStatement’ del progetto “Jakarta”

Stan4j, al contrario, non prevede questo tipo di violazione, ma dall'analisi della classe 'DelegatingPreparedStatement' notiamo la presenza di problemi, legati alle metriche ELOC, WMC e Methods.

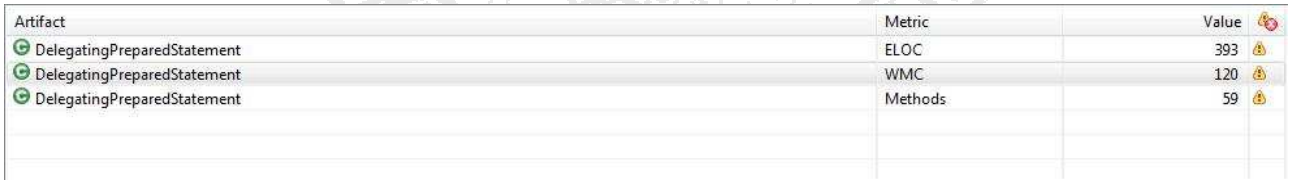
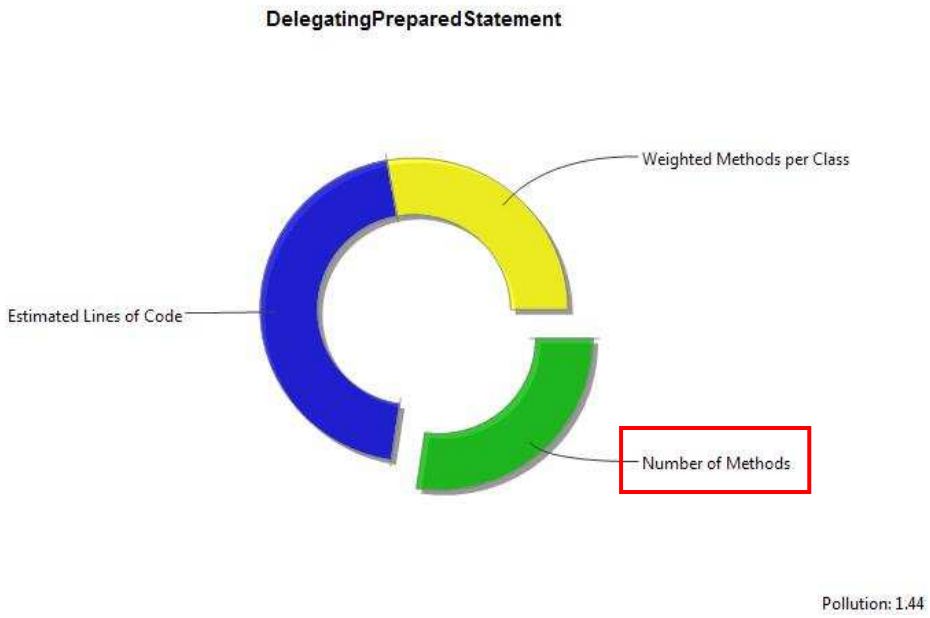


Fig. 25. Violazioni in Stan4j della classe ‘DelegatingPreparedStatement’ del progetto “Jakarta”



Dalla Pollution Chart di Stan4j, notiamo la complessità della classe in esame a causa di un numero elevato di metodi che aumentano la metrica WMC.

Fig. 26. Pollution Chart in Stan4j della classe 'DelegatingPreparedStatement' del progetto "Jakarta"

Le stesse considerazioni vengono fatte da Sonar, anche se esso non le inserisce come violazioni del codice ma come semplice analisi della sua struttura.

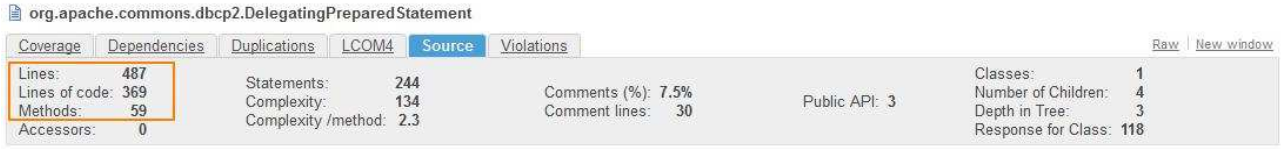


Fig. 27. Caratteristiche del codice in Sonar della classe 'DelegatingPreparedStatement' del progetto "Jakarta"

- Complessità

Sonar evidenzia la complessità media associata ai metodi, alle classi e ai singoli file, della cartella 'dbcp2'.

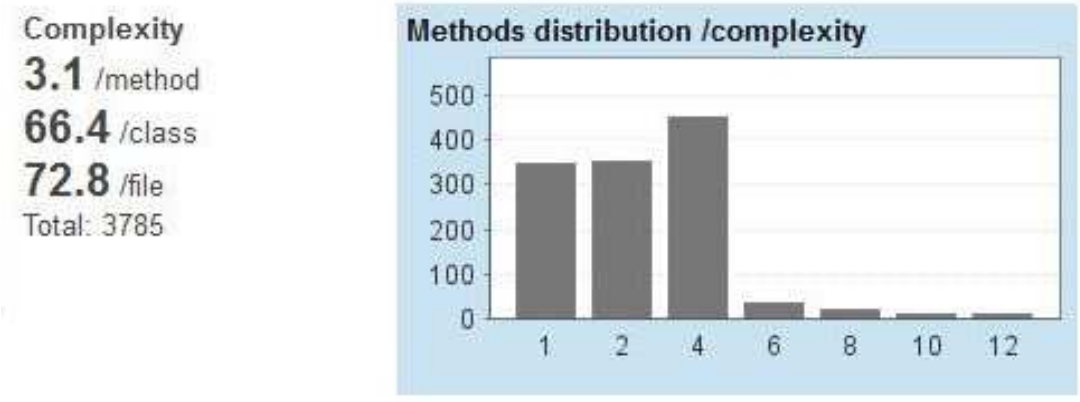


Fig. 28. Distribuzione della complessità media per metodo in Sonar del progetto "Jakarta"

Per i metodi in esame otteniamo anche un grafico che identifica la loro distribuzione in base al grado di complessità. La complessità per metodo, pari in questo caso ad un valore di '3.1', si

riferisce alla complessità ciclomatica media dei metodi presenti all'interno della cartella; il valore è sufficientemente basso.

Stan4j individua il valore medio totale della complessità ciclomatica, non riferito ai soli metodi, sia con la Pollution Chart che attraverso i valori matematici. Anche in questo caso il valore di '1.8' è sufficientemente basso.

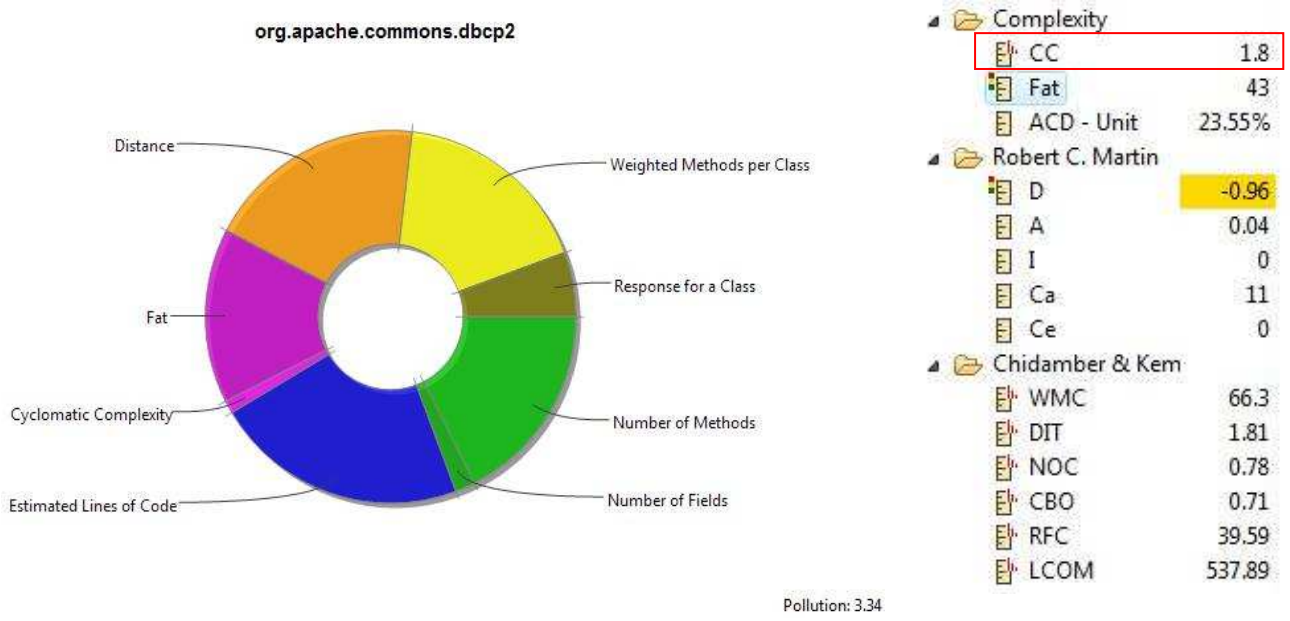


Fig. 29. Pollution Chart ed elenco di metriche in Stan4j del pacchetto 'dbcp2' del progetto "Jakarta"

- LCOM

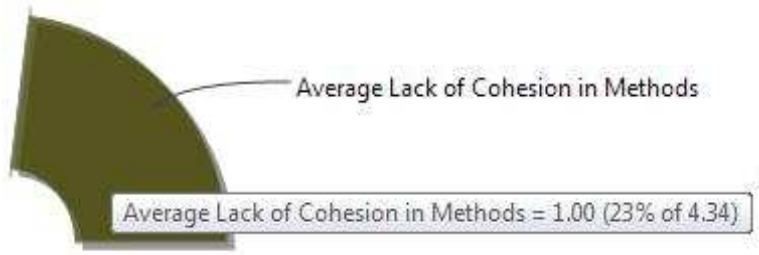


Fig. 30. Spicchio di Pollution Chart della metrica LCOM in Stan4j del progetto "Jakarta"

Dall'analisi di questa metrica, notiamo che Stan4j e Sonar utilizzano modelli di calcolo differenti. In generale un elevato valore di questa metrica indica una bassa coesione.

Stan4j utilizza la metrica "Lack of Cohesion in Methods" di Chidamber & Kemerer che esprime tale valore come il numero di coppie di metodi presenti in una classe, che non posseggono coesioni.

Esso è calcolato come la differenza tra il numero di coppie di metodi che non hanno variabili d'istanza in comune, e il numero di coppie di metodi che le posseggono. L'analisi con Stan4j del pacchetto 'dbcp2' individua un valore di LCOM pari a 537.89, riportato nel report come una violazione.

Sonar invece utilizza la metrica "Lack of Cohesion of Methods" che prevede il calcolo del grado di coesione con il quale i metodi di una classe sono collegati tra loro. La metrica utilizzata da Sonar, calcola tale valore secondo i criteri definiti da Hitz & Montazeri, la quale identifica con il valore 1 una classe coesa nel migliore dei modi, con il valore 0 una classe che non possiede metodi, quindi un cattivo progetto, e con un valore maggiore o uguale di 2, una classe che è troppo coesa e

dev'essere divisa in più sottoclassi. In questo caso, il valore associato al pacchetto 'dbcp2' è pari a 1.5, un valore sufficientemente buono.

Effettivamente, questa è una delle metriche più dibattute di tutta l'analisi statica del software.

Fig. 30. LCOM4 in Sonar del pacchetto 'dbcp2' del progetto "Jakarta"

A causa della sua versatilità, sono stati creati diversi metodi di calcolo per ottenere un valore che potesse rappresentare nel migliore dei modi la caratteristica di LCOM. Per questo motivo è una metrica poco utilizzata dagli analisti, poiché potrebbe facilmente creare disguidi.

2.4 Esempio 3: Service Mix

ServiceMix è un Enterprise Service Bus, rilasciato sotto licenza Apache, che consente di integrare varie tecnologie con la finalità di creare servizi disponibili per il riuso.

2.4.1 Stan4j

- Composition View

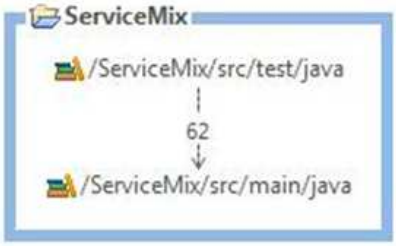


Fig. 31. Composition View in Stan4j del progetto "ServiceMix"

- Treemap

Da un'analisi superficiale del progetto, notiamo subito la sua elevata complessità in confronto a Jakarta.

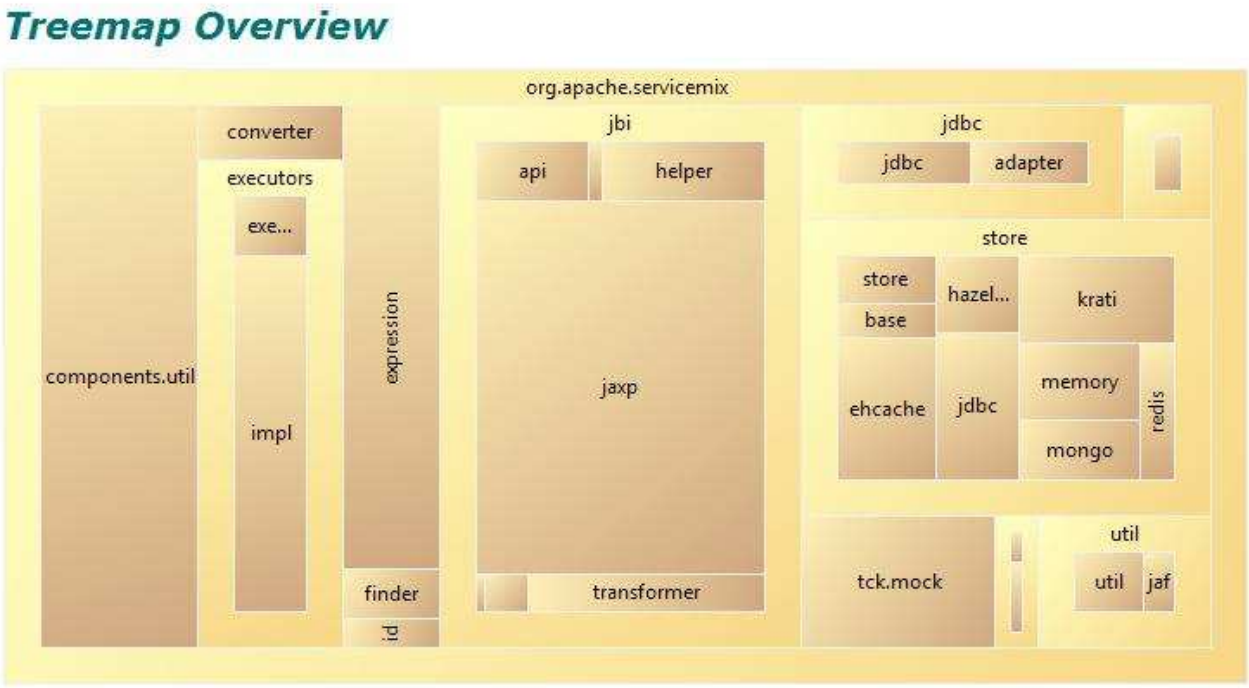


Fig. 32. Treemap in Stan4j del progetto “ServiceMix”

- Violazioni

Le violazioni più significative riscontrate da Stan4j in questo progetto riguardano le metriche LCOM, Complessità Ciclomatica, Distance e Tangled.

- Complessità Ciclomatica

Stan4j individua nel metodo 'parse' un valore di complessità ciclomatica pari a 28. Il valore è effettivamente troppo elevato, ricordando che il limite massimo è di 10.

In questo caso sarebbe consigliato, ridurre la complessità diminuendo il numero degli statement presenti.

Fig. 33. Complessità Ciclomatica in Stan4j del metodo ‘parse’ del progetto “ServiceMix”

- LCOM

Abbiamo evidenziato in rosso, i valori più alti, e quindi i peggiori, che esprimono il valore medio di LCOM.

Average Lack of Cohesion in Methods

Artifact	Value
servicemix.jbi.jaxp	81.13
servicemix.components.util	68.59
servicemix.executors.impl	32.57
servicemix.expression	25.33
servicemix.util	3
servicemix.tck.mock	68.22
servicemix.store.jdbc	29.20
servicemix.jdbc	108.67
servicemix.store.ehcache	106.50
servicemix.store.krati	90.25
servicemix.jdbc.adapter	12.33
servicemix.store.memory	5.14
servicemix.jbi.helper	15.80
servicemix.jbi.transformer	14.60
servicemix.executors	6.20
servicemix.locks.impl	4
servicemix.converter	1.67
servicemix.finder	6
servicemix.store.mongo	82
servicemix.store.hazelcast	24
servicemix.util.jaf	13.50
servicemix.store.redis	32.50
servicemix.timers.impl	8
servicemix.id	6
servicemix.jbi.marshaler	3

Fig. 34. LCOM in Stan4j dei pacchetti del progetto “ServiceMix”

Di fianco vediamo la distribuzione metrica di LCOM per quanto riguarda il pacchetto 'servicemix.jdbc'.

Il 66.67% delle classi presenti all'interno di questo pacchetto hanno LCOM<=0 e il 33.33% hanno 25<LCOM<=inf. Anche se la percentuale di tali classi è minore, il valore di LCOM è troppo elevato per esse.

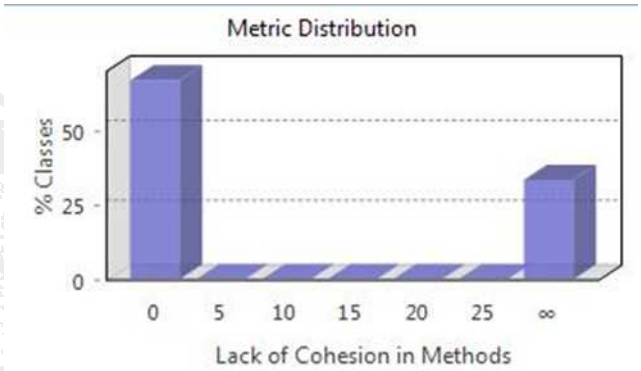
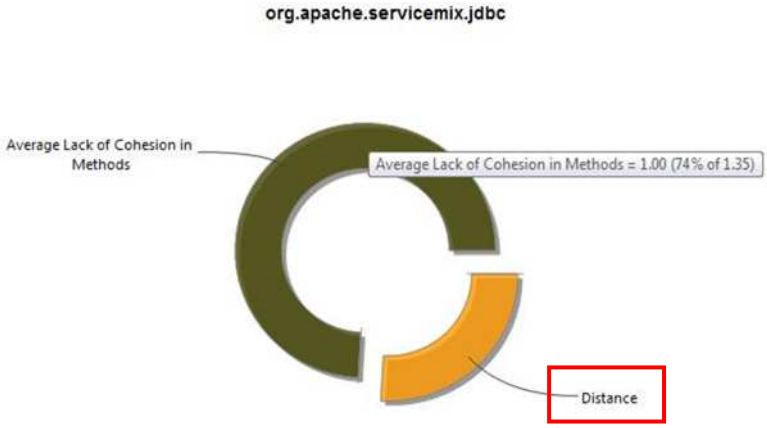


Fig. 35. Distribuzione della metrica LCOM in Stan4j del pacchetto ‘jdbc’ del proegtto “ServiceMix”



La Pollution Chart applicata al pacchetto analizzato, ci mostra la percentuale della metrica LCOM violata(74% del valore totale della pollution).

Fig. 36. Pollution Chart in Stan4j del pacchetto ‘jdbc’ del progetto “ServiceMix”

- Distance

Stan4j evidenzia ben nove punti nel codice in cui tale metrica viene violata, ma due di essi sono quelli che incidono di più in modo negativo. In ogni caso nessuna di loro è da considerarsi una violazione critica. La prima riguarda il pacchetto 'jaxp', con un valore di –‘0.81’ e delle coordinate di [0,0.19]. In questo caso essendo entrambi i valori molto vicini allo zero, la violazione intende esprimere l'elevata concretezza e rigidità dell'elemento in esame.
Il secondo valore riguarda il pacchetto 'expression' con coordinate pari a [0.33,0.05]. Il valore della coordinata riguardante l'astrazione è più piccolo di quello del pacchetto precedente, questo indica un aumento della concretezza e dell’instabilità, che avvicina il comportamento del pacchetto in esame al valore ideale.

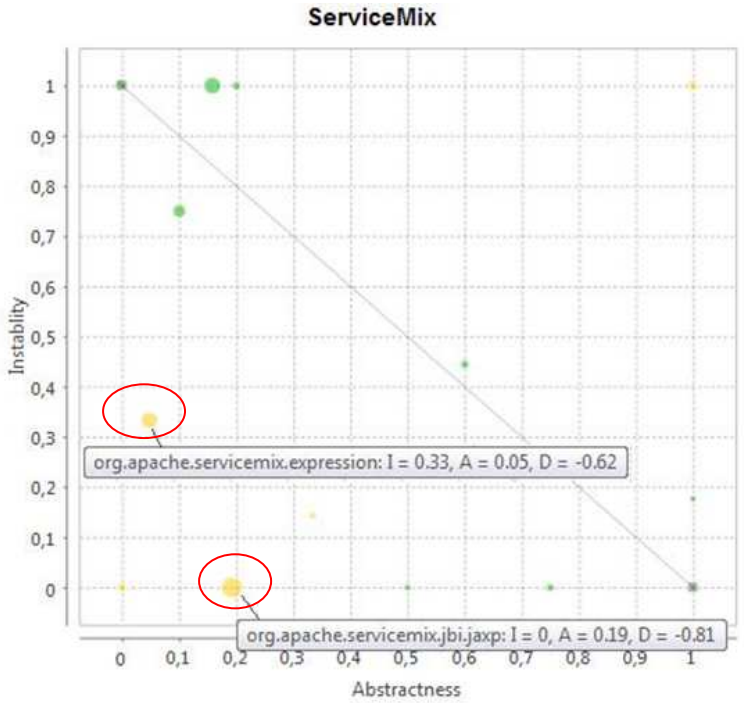


Fig. 37. Distanza dalla main sequence in Stan4j del progetto “ServiceMix”

- Tangle

La novità che riscontriamo rispetto ai progetti precedenti è questa metrica, che rientra nell'insieme delle metriche riguardanti la complessità. Un tangle è una porzione di un grafico delle dipendenze in cui gli elementi coinvolti sono interdipendenti, presentano cioè una dipendenza ciclica. La presenza dei tangle comporta un aumento della complessità del progetto.

Fig. 38. Tangle in Stan4j della cartella ‘executors’ del progetto “ServiceMix”

I tangles, in Stan4j, sono calcolati in percentuale, e in questo caso il valore violato, associato al pacchetto 'executors', è di ‘30.77%’.
Essendo un valore alto, bisognerebbe ridurlo trasformando il grafo in esame in un grafo aciclico. Ovviamente il numero di archi è legato al numero di linee di codice dipendenti, ed è molto più facile rimuovere un arco riferito ad un legame leggero, piuttosto che ad uno pesante. Per eliminare le dipendenze cicliche bisogna adottare la teoria del Minimum Feedback Set: ridurre il più possibile il numero di archi, o, se questo è già ridotto al minimo, eliminare gli archi che puntano nella direzione sbagliata.

Fig. 39. Grafico del tangle in Stan4j della cartella ‘executors’ del progetto “ServiceMix”

Il report ci mostra l'interdipendenza tra i pacchetti 'impl' ed 'executors' della cartella ‘executors’.

2.4.2 Checkstyle

- Checkstyle violation chart

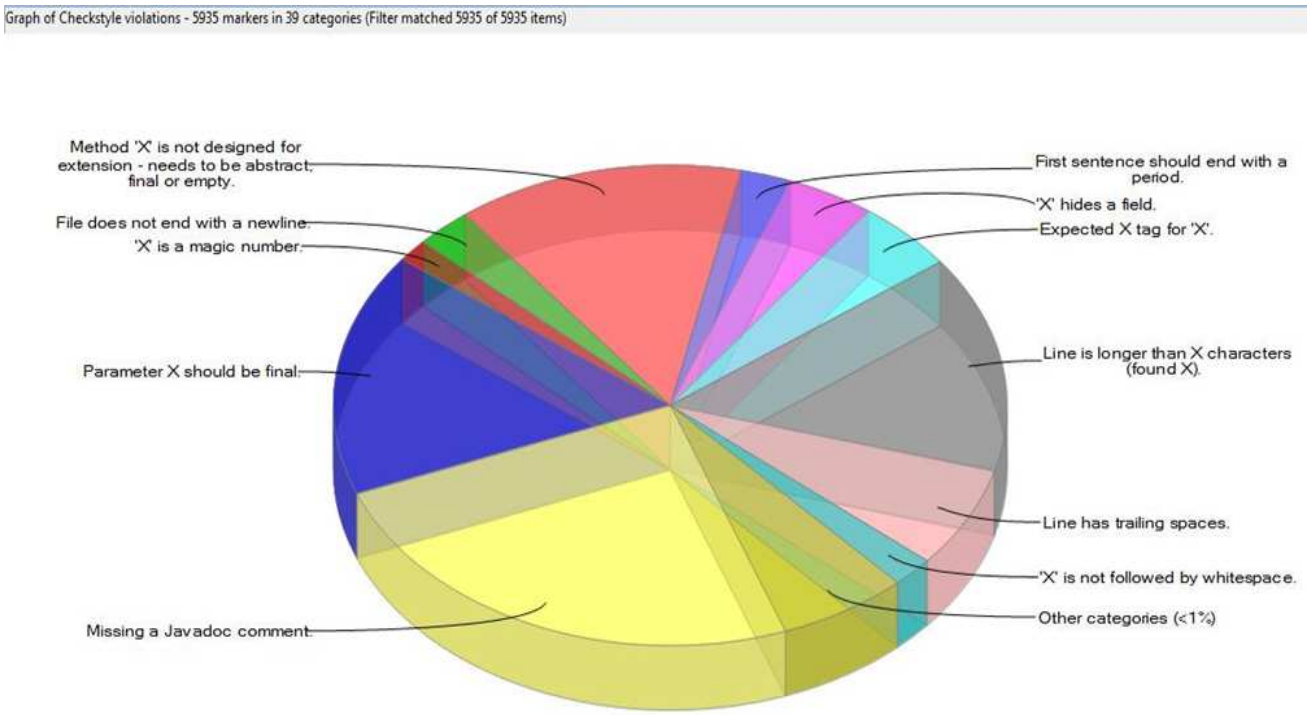


Fig. 40. Grafico delle violazioni in Checkstyle del progetto “ServiceMix”

In questo caso l’analisi del progetto tramite Checkstyle non ha evidenziato violazioni critiche.

2.4.3 Sonar

- Dashboard

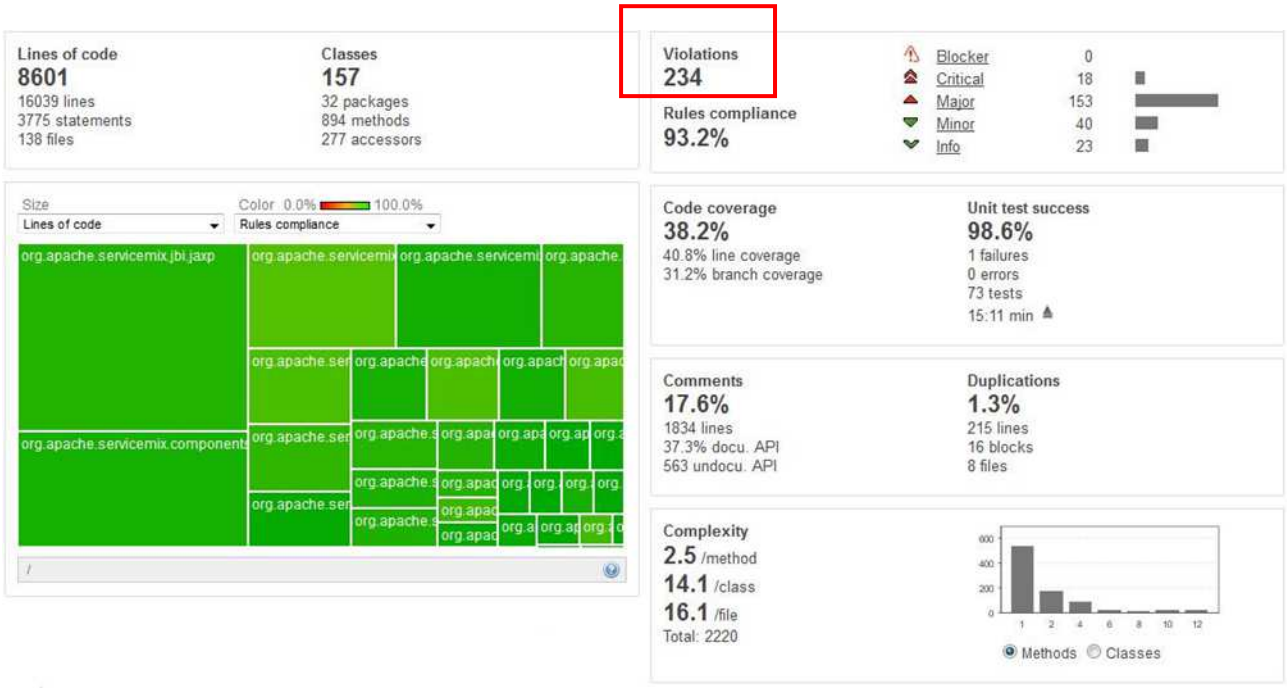


Fig. 41. Dashboard in Sonar del progetto “ServiceMix”

- Violazioni Critiche

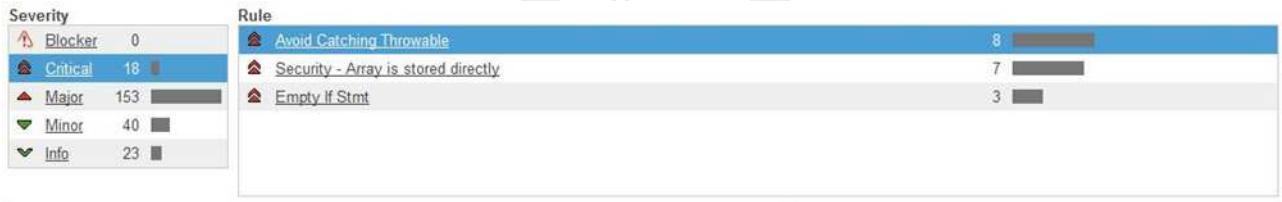


Fig. 42. Violazioni critiche in Sonar del progetto “ServiceMix”

In questo progetto, Sonar individua 18 punti all'interno del codice in cui sono state violate delle metriche in maniera critica. Tali violazioni sono divise in 3 gruppi: 'Avoid catching throwable', 'Array is stored directly'(già analizzato precedentemente), e 'Empty if statement'.

- Avoid Catching Throwable

La prima violazioni riguarda proprio la classe 'Throwable'. Non è quasi mai una buona pratica cercare di catturare eventuali eccezioni attraverso questa classe; sarebbe molto meglio utilizzare la classe 'Exception' (sottoclasse di Throwable). Poichè la classe 'Throwable' copre una rete molto ampia di eccezioni potrebbe capitare che a causa del suo utilizzo non si riescono a catturare eccezioni come 'OutOfMemoryError'. Spiegare meglio.

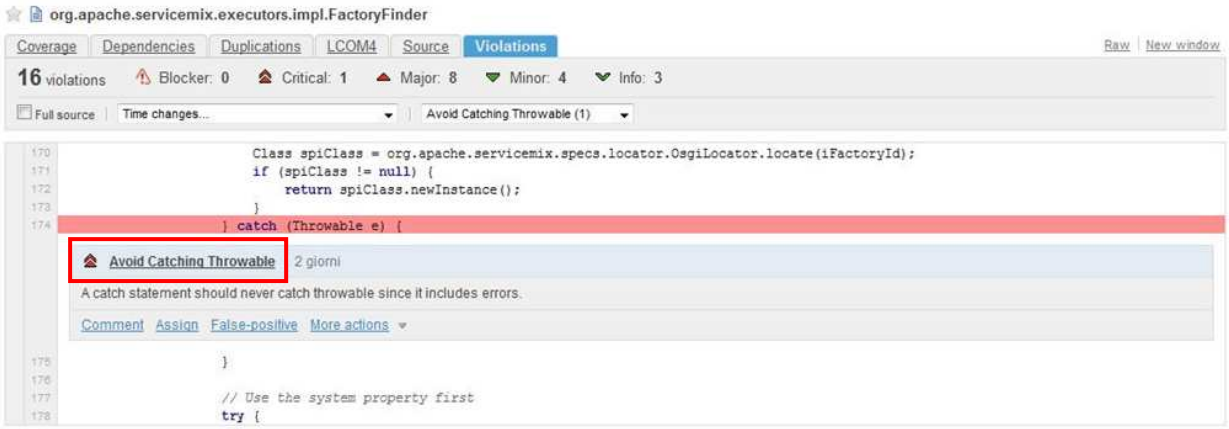


Fig. 43. Violazione 'Avoid Catching Throwable' in Sonar della classe 'FactoryFinder' del progetto "ServiceMix"

- Empty If Statement

Questa violazione evidenzia il fatto che esistono in 3 parti del codice, degli statement con il corpo vuoto. Anche se può sembrare strano che una situazione del genere possa apportare un danno critico, l'utilizzo di statement vuoti indica che il programmatore non è riuscito a gestire al meglio un'eventuale errore e ha sorvolato su di esso senza verificare quali danni potrebbe creare. Sarebbe meglio inserire un qualsiasi controllo, anche semplice, all'interno di ogni blocco.

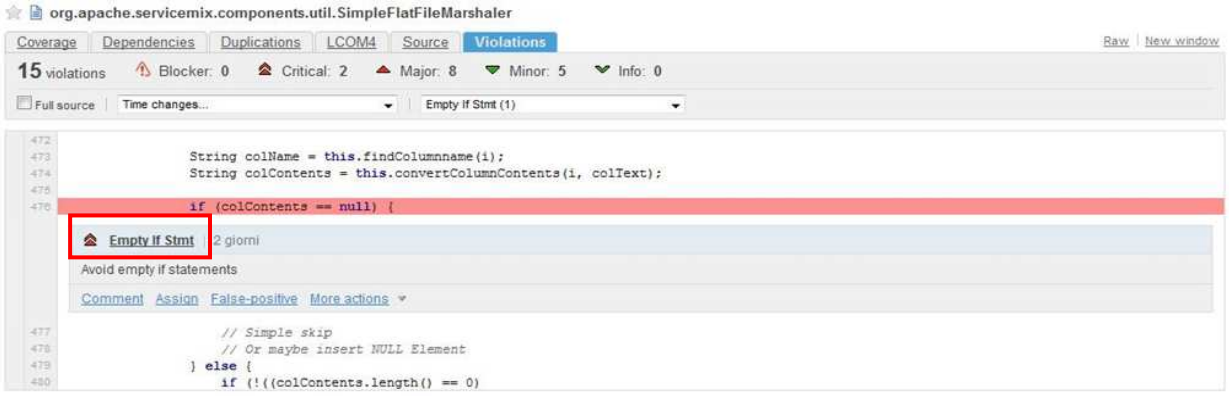
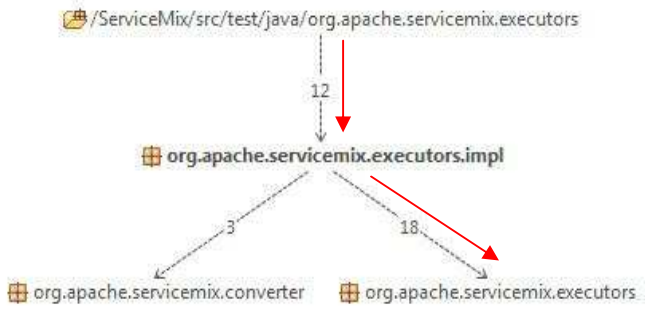


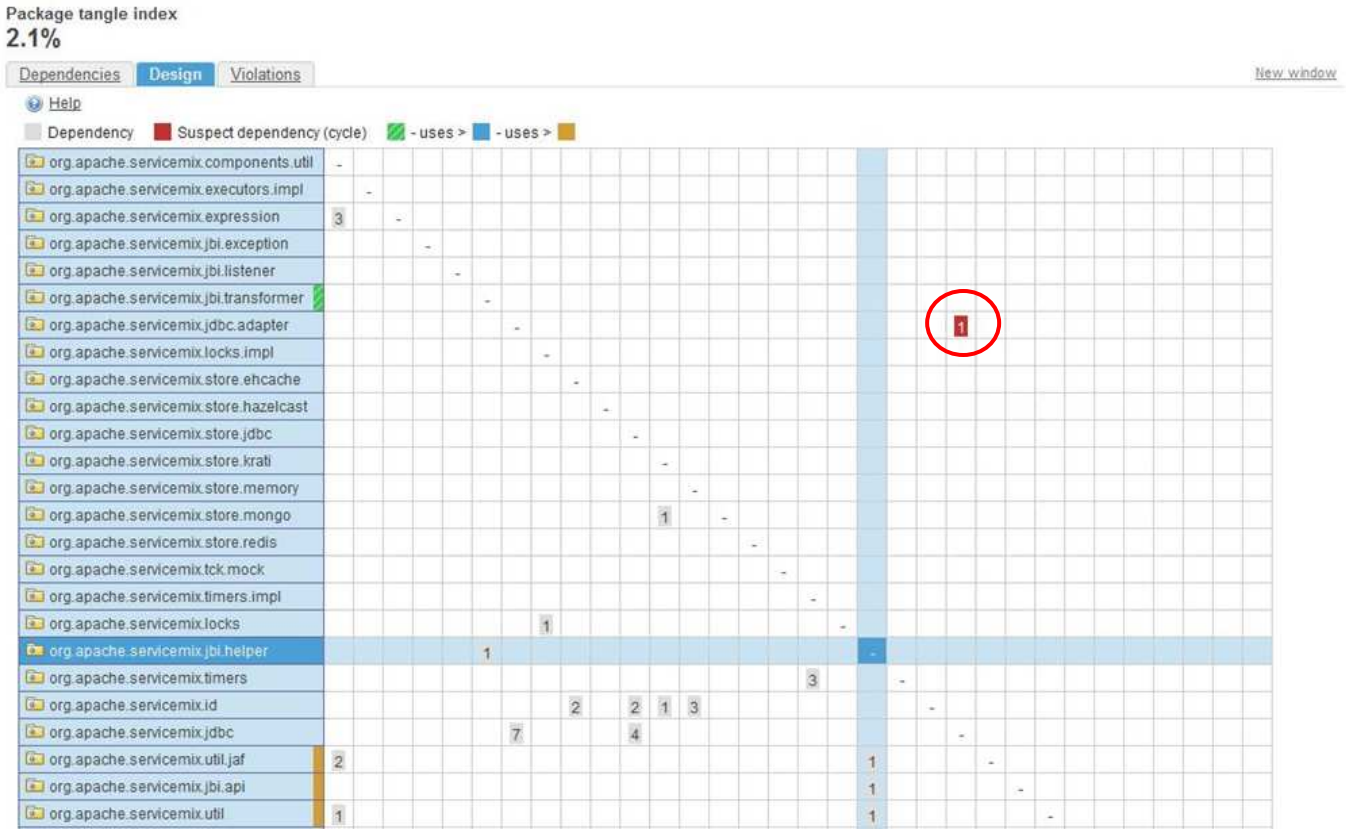
Fig. 44. Violazione 'Empty If Statement' in Sonar della classe 'SimpleFlatFileMarshaler' del progetto "ServiceMix"

2.4.4. Confronto

- Tangle



Stan4j individua la presenza di un'interdipendenza, legata alla cartella 'executors' tra i pacchetti 'executors' e 'imp'. In particolare nella libreria '/main/java', il pacchetto 'imp' dipende da 'executors', mentre nella libreria '/test/java' il pacchetto 'executors' dipende dal pacchetto 'imp'.



Sonar, consente di effettuare un'analisi di tutte le dipendenze individuate tra i pacchetti appartenenti al progetto. Notiamo, dalla legenda, come i pacchetti evidenziati in verde utilizzano i pacchetti evidenziati in blu, i quali a loro volta utilizzano i pacchetti evidenziati in color ambra. Il package tangle index individua il livello di interdipendenza tra i pacchetti; il miglior valore è lo 0% e il peggiore è il 100%. Nel grafico il sospetto di una interdipendenza è evidenziato in rosso, ed è associato alla cartella 'jdbc' tra i pacchetti 'jdbc' e 'adapter'. Il valore di 2.1% è molto basso e quindi non è visto come una violazione.

Conclusioni

La qualità del software è migliorata moltissimo negli ultimi quindici anni, grazie all'introduzione di nuove tecniche e tecnologie, e nell'industria del software è aumentata la consapevolezza dell'importanza della gestione della qualità del codice sorgente. L'utilizzo di questi strumenti che analizzano automaticamente il codice sorgente, ha permesso ai progettisti di creare in maniera sempre più rapida dei prodotti pronti all'uso, evitando continue fasi di refactoring come in passato, causate dalla scarsa qualità del progetto.
Di seguito, i motivi che hanno portato i progettisti a servirsi di tali strumenti:

- Rendere maggiormente industrializzato un processo che è ancora molto artigianale
- Evitare il fallimento di progetti di grandi dimensioni
- Minimizzare i costi di produzione e manutenzione per consentire un facile intervento da parte di persone diverse
- Riciclare il software per permettere il riutilizzo di un lavoro già testato

Purtroppo ci sono ancora molte società che non fanno uso di misurazioni sistematiche del software per valutarne la qualità, essendo i loro prodotti mal definiti e controllati, ma il continuo uso di tali prodotti open source promette uno sviluppo positivo in questo senso.
Nonostante ciò il concetto di qualità del software è ancora molto complesso e non basta seguire standard e metriche per ottenere un prodotto ottimale. I manager dello sviluppo puntano a infondere una "cultura della qualità" incoraggiando il team a lavorare nel migliore dei modi, cercando di non affidare del tutto il miglioramento del progetto a questi software.
Inoltre, gli standard di processo possono causare difficoltà se al team di sviluppo vengono imposte delle metriche non adeguate al progetto in esame. Abbiamo infatti visto che, a seconda del prodotto utilizzato le modifiche suggerite cambiano anche in modo radicale. Tipi diversi di software hanno bisogno di processi di sviluppo diversi. I progettisti devono quindi evitare il problema degli standard inadatti pianificando la qualità del codice all'inizio della stesura del progetto, e settando in modo adeguato le metriche che verranno utilizzate per analizzare il codice.
Un altro punto da mettere in risalto è quello dell'open source, caratteristica comune a tutti gli strumenti che abbiamo utilizzato. I programmatori che lavorano per l'open source incoraggiano lo studio, l'implementazione e la modifica dei programmi da loro generati. Questo perché è importante condividere qualsiasi tipo di conoscenza in una società dove il profitto è diventato la nuova filosofia di vita. Il semplice ed immediato utilizzo di tali strumenti ha prodotto tre benefici:

- Hanno incoraggiato gli sviluppatori nell'adozione di tecniche di analisi sempre migliori
- Hanno consentito l'aumento delle conoscenze del codice sorgente da parte del team di sviluppo
- Hanno ridotto il costo di manutenzione

Quindi sia che si tratti di soddisfare le richieste di un cliente, o gli standard imposti da modelli generali, avere un codice sorgente di più elevata qualità può aiutare le imprese a soddisfare i requisiti di conformità, e aiuta i progettisti a lavorare in maniera più efficiente e motivata.

Bibliografia

[1] *Design Principles and Design Patterns*, Robert C.Martin, 2000

[2] *OO Design Quality Metrics -An Analysis of Dependencies*, Robert C. Martin, 1994

[3] *Standard di riferimento e modelli di qualità in ambito software*, Salvatore Rubino, 2007

[4] *Tools for static code analysis: A survey*, Patrik Hellström, 2009

[5] *Commons sense and code quality*, "http://techforenterprise.blogspot.in/2012/03/common-sense-and-code-quality-part-1.html"

[6] *In pursuit of code quality: Code quality for software architects*, Andrew Glover, 2006v

[7] *Ingegneria del software*, Ian Sommerville, 8 ed., 2010

[8] *Stan4j*, "http://stan4j.com/"

[9] *Sonar*, "http://www.sonarsource.org/"

[10] *Checkstyle*, "http://checkstyle.sourceforge.net/"