Análise de Grandes Grafos
2024/25 – 2º Semestre

Relatório Trabalho Final
Dependências do JUnit

Trabalho realizado por:
Eduardo Vieira - 65361
Gustavo Feio - 65714
Ivo Lopes - 65511

Com apoio dos docentes:
Pedro Mota
Maria Isabel Gomes

DEPARTAMENTO DE INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Departamento Matemática
NOVA FCT
Portugal
Junho de 2025

1 Introdução

Este trabalho aborda a construção de uma rede, sob a forma de grafo, e análise algumas das suas características relevantes. Para esse fim, escolhemos como rede as dependências do JUnit [2], uma framework bastante útil para testes unitários para aplicações Java. Esta análise é importante por diversas razões. Primeiramente permite-nos visualizar mais claramente a arquitetura das classes desta framework e a forma como se relacionam. Desta forma, podemos identificar, por exemplo, ciclos que podem dificultar a manutenção e/ou evolução da framework, e complicam testes.

Além disso, o grafo de dependências possibilita a identificação de módulos altamente acoplados, que podem representar pontos críticos do sistema. Módulos com muitas dependências de entrada tornam-se mais difíceis de modificar, pois qualquer alteração pode ter impactos significativos em outras partes da framework. Por outro lado, módulos com muitas dependências de saída revelam um elevado grau de dependência de outros componentes, o que reduz a sua autonomia e reutilização.

A análise do grafo também permite observar a modularidade da framework, ou seja, quão bem definidos e isolados estão os diferentes componentes. Uma boa modularidade contribui para a escalabilidade do projeto e facilita a manutenção, testes e extensão da framework. Através de análises como o grau de entrada e saída, a densidade do grafo e a identificação de componentes centrais, é possível obter uma visão quantitativa do estado estrutural da arquitetura.

Para a construção do grafo, foram considerados os módulos do JUnit e as suas relações de dependência diretas. O grafo foi modelado como uma estrutura direcionada, onde cada nó representa uma classe e cada aresta uma dependência explícita entre as mesmas. Esta representação visual e matemática permite uma compreensão mais profunda da organização interna da framework e serve como base para eventuais melhorias arquiteturais.

Estas informações foram obtidas através de uma matriz de adjacências disponível em [1].

2 Grafo de Dependências

O grafo obtido encontra-se na Figura 1.

Como já foi referido, este é um grafo direcionado, visto que se trata de uma rede de dependências. Cada nó representa uma classe e cada aresta uma dependência entre estas, nomeadamente por utilização de funções implementadas nelas.

A direcionalidade do grafo reflete a relação de uso/importação entre as classes, isto é, se o módulo A depende do módulo B, então existe uma aresta orientada de A para B. Desta forma, é possível mapear o fluxo de dependências e compreender melhor a organização interna da framework.

Figura 1: JUnit - Grafo de Dependências

3 Análise do Grafo

3.1 Grau (Degree)

Para primeira análise estrutural do grafo, foi gerada, em `R` utilizando a biblioteca `igraph`, uma imagem em que os nós são representados com cores distintas consoante o seu grau. Isto é, o número de dependências associadas a cada um. A representação visual resultante na Figura 2 facilita a identificação imediata de módulos críticos, permitindo distinguir entre os pacotes mais isolados e aqueles que se encontram fortemente conectados dentro da rede.

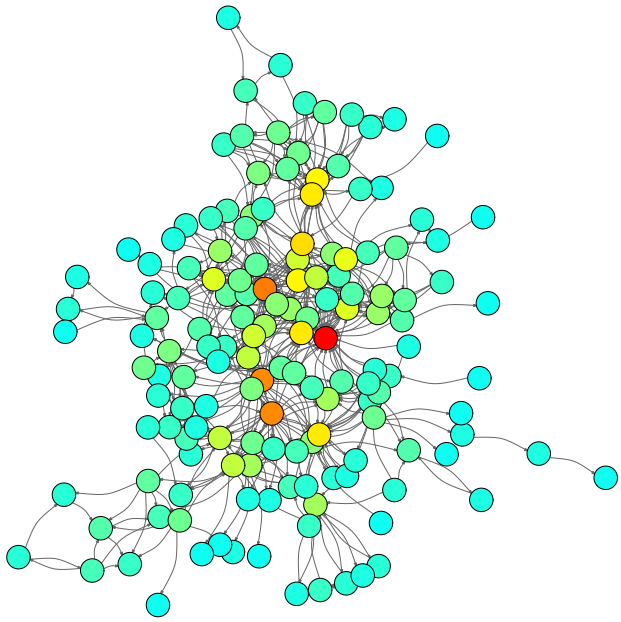

Figura 2: Visualização dos Graus

Nós com grau elevado foram destacados com cores mais intensas, uma vez que representam pontos centrais e críticos da arquitetura. Alterações nestes pacotes podem ter impacto em cascata sobre outros, o que aumenta o risco e a complexidade da manutenção e atualização. Por outro lado, nós com grau reduzido indicam módulos mais independentes, geralmente mais fáceis de testar e reutilizar.

No contexto do JUnit, esta abordagem é particularmente relevante, pois

permite identificar quais partes da framework têm um papel fundamental e quais atuam como extensões específicas. Tal distinção é essencial para garantir a modularidade da framework, evitando que funcionalidades de uso específico dependam fortemente de componentes centrais ou vice-versa.

Assim, a coloração por grau não só facilita a leitura visual do grafo, como também fornece dados sobre a dependência e a importância relativa de cada pacote dentro da rede.

3.2 Intermediação (Betweenness)

Outra métrica considerada na análise do grafo de dependências foi a centralidade de intermediação. Esta medida avalia a frequência com que um nó aparece nos caminhos mínimos entre pares de outros nós. Isto é, indica o quão intermediário um pacote é na comunicação ou ligação entre diferentes partes da framework.

Na Figura 3, os nós foram coloridos de acordo com o valor da sua intermediação, com cores mais intensas indicando valores mais elevados, ou seja, módulos que funcionam como pontes críticas entre diferentes partes da rede. Estes módulos não têm necessariamente o maior número de conexões ou grau, mas desempenham um papel estratégico na conectividade global da estrutura.

Esta análise é especialmente importante, pois permite identificar módulos de intermediação, que embora possam não ser centrais em termos de dependência direta, são fundamentais para manter a coerência e a integração entre diferentes componentes. Um módulo com alta centralidade de intermediação, se modificado ou removido, pode eliminar caminhos importantes entre outros módulos, afetando a integridade da framework como um todo.

Além disso, esta métrica ajuda a distinguir entre nós "populares"(com alto grau) e nós "estratégicos"(com alta intermediação), contribuindo para uma compreensão mais refinada da arquitetura. Em projetos de larga escala como o JUnit, esta distinção pode ser crucial para decisões de refatoração, modularização ou otimização do desempenho da framework.

Assim, a visualização baseada no betweenness complementa a análise de grau, permitindo uma visão mais completa dos pontos estruturais sensíveis da rede de dependências.

3.3 Diâmetro (Diameter)

O diâmetro da rede ilustra a maior distância entre quaisquer dois nós no grafo de dependências do JUnit. O diâmetro representa o caminho mais longo dentro da rede entre dois módulos conectados, sendo um indicador da profundidade estrutural da framework.

No JUnit, o maior diâmetro encontrado foi de 9. Esta informação permite identificar módulos periféricos, que se encontram mais distantes do núcleo do sistema, e também compreender o grau de propagação de uma alteração — ou seja, até onde uma dependência pode afetar o restante da rede.

No contexto desta análise, um diâmetro muito elevado pode sinalizar complexidade excessiva ou baixa coesão entre partes da framework, enquanto um

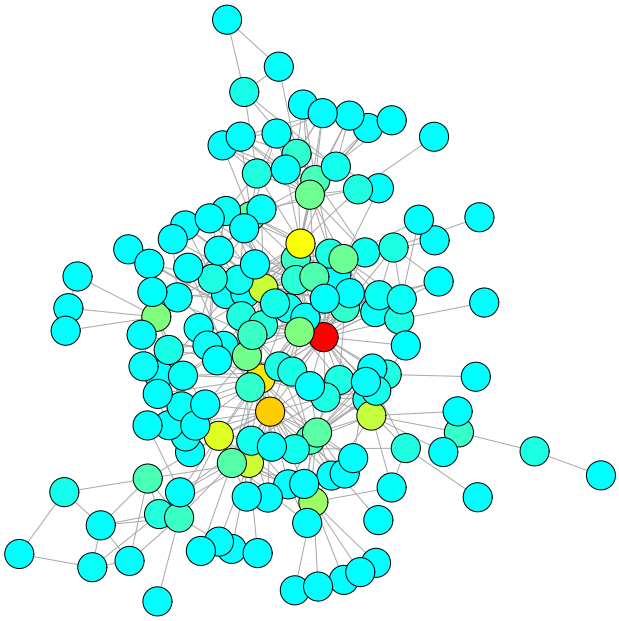

Figura 3: Visualização da Intermediação

diâmetro mais curto tende a indicar melhor integração e modularidade.

3.4 Modularidade (Modularity)

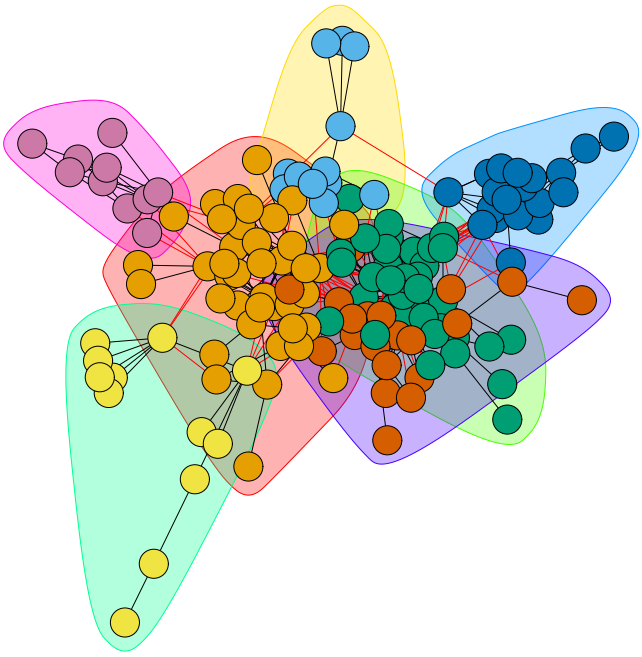

Figura 4: Visualização da Modularidade

A imagem na Figura 4 ilustra a divisão do grafo em comunidades ou subgrupos coesos, onde os nós dentro de um mesmo grupo estão mais densamente conectados entre si do que com nós de outros grupos. Cada comunidade é representada por uma cor distinta, permitindo observar como a arquitetura do JUnit se organiza em termos de blocos funcionais ou temáticos.

A modularidade é uma métrica fundamental em redes complexas, pois reflete o grau de separação funcional entre os componentes. Uma modularidade elevada sugere que a framework está bem estruturada em termos de coesão interna e baixa acoplagem externa, características desejáveis numa arquitetura eficiente. Além disso, quantifica o nível de transitividade. Deste modo, a modularidade representa o quanto determinados pacotes dependem uns dos outros.

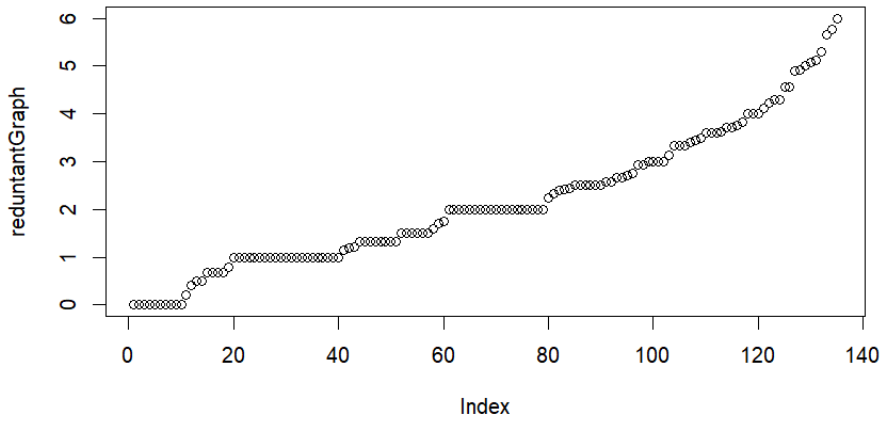

Figura 5: Gráfico de Redundância

3.5 Reduncância (Redundancy)

A análise de redundância, presente na Figura 5 mostra a existência de caminhos alternativos entre os módulos no grafo, ou seja, em que medida a rede mantém conectividade mesmo na ausência de determinados nós ou arestas.

Em termos de arquitetura de software, a redundância pode ter duas interpretações:

- Quando moderada, é positiva, pois garante resiliência da estrutura — a remoção de um módulo não quebra a comunicação entre os restantes.
- Quando excessiva, pode indicar dependências duplicadas ou desnecessárias, que aumentam a complexidade sem aumentar valor.

No caso do JUnit, observar a redundância permite entender quais módulos são verdadeiramente críticos (com pouca redundância ao seu redor) e quais têm dependências potencialmente redundantes, podendo ser simplificados.

4 Conclusões

A análise do grafo de dependências do JUnit revelou-se uma abordagem eficaz para compreender a arquitetura interna desta framework muito utilizada para testes em aplicações Java. A representação em grafo permitiu visualizar com clareza as relações entre classes e identificar padrões estruturais que, de outro modo, poderiam passar despercebidos numa inspeção apenas textual ou documental.

Através do estudo das diferentes medidas de centralidade — como o grau, a centralidade de intermediação, o diâmetro, a modularidade e a redundância — foi possível avaliar a importância relativa de cada módulo dentro da rede, tanto em termos de conectividade direta quanto de papel estratégico na comunicação entre os demais componentes.

Estas análises forneceram informações importantes, como:

- A identificação de módulos centrais e críticos, cuja modificação pode impactar grande parte da framework;
- A deteção de estruturas modulares, que favorecem a escalabilidade e a manutenção;
- A evidência de módulos de intermediação, cuja posição na rede os torna fundamentais para a coesão do sistema;
- A avaliação da robustez estrutural da rede, com base na existência ou não de caminhos alternativos (redundância);
- E a compreensão da profundidade e dispersão das dependências, por meio do diâmetro da rede.

Num todo, estas métricas não só permitem avaliar a qualidade da arquitetura do JUnit como a de qualquer outro sistema. A aplicação destes conceitos ao desenvolvimento de um software dá-nos uma maior clareza estrutural, permitindo assim garantir uma arquitetura de software robusta.

Referências

[1] Dan Amlund Thomsen: Adjacency matrix code visualizer - junit 4.11. `https://danamlund.dk/adjmatrix/junit-4.11.html` (2019), accessed: 2025-05-30

[2] The JUnit Team: Junit 4. `https://junit.org/junit4/` (2021), accessed: 2025-05-30