UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
ESCOLA DE INFORMÁTICA APLICADA

ANÁLISE DA QUALIDADE DO *DESIGN* DA BIBLIOTECA *JODA-TIME*

João Gabriel Passos Cosendey Brouck
Lucas Carvalho de Vasconcellos

Orientador:

Márcio de Oliveira Barros

RIO DE JANEIRO, RJ - BRASIL
AGOSTO, 2023

ANÁLISE DA QUALIDADE DO *DESIGN* DA BIBLIOTECA *JODA-TIME*

JOÃO GABRIEL PASSOS COSENDEY BROUCK
LUCAS CARVALHO DE VASCONCELLOS

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação.

Aprovada por:

Márcio de Oliveira Barros, D.Sc. – PPGI/UNIRIO

Jefferson Elbert Simões, D.Sc. – BSI/UNIRIO

RIO DE JANEIRO, RJ - BRASIL
Agosto, 2023

Brouck, João Gabriel.
Análise da Qualidade do *Design* da Biblioteca *Joda-Time* / João Gabriel Brouck. – Rio de Janeiro, 2023.

Carvalho de Vasconcellos, Lucas.
Análise da Qualidade do *Design* da Biblioteca *Joda-Time* / Lucas Carvalho de Vasconcellos. – Rio de Janeiro, 2023.

Orientador: Márcio de Oliveira Barros
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal do Estado do Rio de Janeiro,
Graduação em Sistemas de Informação, 2023

Agradecimentos

Agradeço ao meu orientador Márcio Barros pela paciência e pelo bom direcionamento ao longo de todo o processo de desenvolvimento deste trabalho. Também agradeço a todos as outras pessoas que me ajudaram de forma direta ou indireta neste processo, através de sugestões, conversas e aconselhamentos relacionados ao projeto e ao meio acadêmico.

RESUMO

Na área de Engenharia de Software existe o desafio de construir software com qualidade. Os aspectos de qualidade são principalmente relevantes nas etapas de arquitetura e design do software, pois as decisões técnicas ali tomadas terão impacto na evolução e manutenção do software. Dito isso, criaram-se métricas para avaliar a qualidade de um software ao longo da sua evolução.

Este trabalho tem como objetivo analisar a evolução da biblioteca Joda-Time, amplamente utilizada para representar datas em programas escritos com a linguagem de programação Java, do ponto de vista de três conjuntos de métricas distintos: tamanho e complexidade, dispersão de mudanças e, por fim, coesão e acoplamento. Com esta análise, esperamos contribuir com uma pesquisa de longo prazo, que analisa diversos softwares utilizando a mesma estrutura, e identificar padrões que orientem o desenvolvimento de software no futuro.

Palavras-chave: Joda-Time, *design* de software, métricas de projeto de software.

ABSTRACT

Software Engineering is the science that addresses the challenge of building software with quality. Quality aspects are particularly relevant in the architecture and design stages of the software development life-cycle, as the technical decisions made in this stage impact the evolution and maintenance of the software project in the long run. That said, metrics have been created to evaluate the quality of a software's design throughout its many versions.

This work aims to analyze the evolution of the *Joda-Time* library, widely used to represent dates in programs written in the Java programming language. This analysis is carried from the perspective of three distinct sets of metrics: size and complexity, change dispersion, and cohesion and coupling. By performing this analysis, we hope to contribute to a long-term research project that studies various software using the same framework to identify patterns that lead to architectural erosion and to provide best practices for software design.

Keywords: Joda-Time, software design, software design metrics.

Conteúdo

Lista de Figuras

Lista de Tabelas

1. Introdução

1.1 Motivação

A erosão arquitetônica é o processo pelo qual a arquitetura de um sistema de software se degrada de forma incremental, a medida que desenvolvedores fazem mudanças que violam as intenções dos arquitetos em tempo de *design*. Este processo, ao longo de anos, faz com que uma versão atualizada do software difira significativamente do seu *design* original.

A erosão atrapalha a manutenção e a evolução do software, pois violações dos princípios de arquitetura acabam adicionando complexidade ao *design* do software, tornando-o mais difícil de entender. Além disso, a erosão torna o software progressivamente mais complicado de manter, gerando assim um cenário em que temos mais erosão. Ainda assim, todos os softwares estão sujeitos à erosão.

Para entender como a erosão se infiltra nos projetos de software, precisamos estudar como estes softwares evoluem ao longo do tempo. A existência de projetos de código aberto em desenvolvimento e uso há mais de uma década e cujo histórico de código-fonte é mantido em sistemas de controle de versão permite analisar como estes softwares evoluíram ao longo do tempo.

Esta análise pode nos levar a padrões que nos permitam, a longo prazo, entender como a arquitetura dos softwares evolui e, possivelmente, identificar padrões que permitam entender como a erosão começa e quais são os fatores que influenciam a sua intensidade ao longo da vida do software.

1.2 Objetivos

O objetivo deste projeto é analisar a evolução de um software de código aberto em termos do seu tamanho, sua complexidade, da dispersão das alterações realizadas em seu código-fonte ao longo de suas diversas versões, da coesão e do acoplamento das suas classes. Com esta análise, esperamos construir um resumo baseado em métricas que explique a evolução da arquitetura do software, sem que o código-fonte, em todo o seu nível de detalhe, precise ser examinado.

Para este projeto, escolhemos a biblioteca *Joda-Time*. Esta biblioteca foi desenvolvida para a linguagem de programação Java e implementa diversos métodos para manipulação de datas e horários. Ela foi amplamente utilizada, dadas as limitações da biblioteca padrão para manipulação de datas da linguagem Java. No entanto, a partir da versão 8.0 do Java, a biblioteca *Joda-Time* foi incorporada à biblioteca padrão do Java.

1.3 Organização do texto

Este documento está dividido em quatro capítulos, a começar por esta introdução.

No Capítulo 2 são apresentadas as estratégias adotadas para a realização deste trabalho, incluindo o racional para seleção do software e as estratégias utilizadas para realizar as análises pretendidas. Depois, as métricas que foram selecionadas para realizar as análises do software também são descritas.

O Capítulo 3 apresenta as análises propriamente ditas, realizadas sobre o código-fonte e a versão pré-compilada do *Joda-Time* através da aplicação das métricas descritas no Capítulo 2. Os resultados são reportados por meio de tabelas e gráficos gerados usando a ferramenta de análise estatística R.

Por fim, no Capítulo 4 são apresentadas as conclusões do trabalho, expondo os resultados encontrados. O capítulo é concluído com sugestões de trabalhos futuros no sentido de continuação e aprimoramento deste trabalho.

2. Uma Estratégia para a Análise do *Design* de um Software

2.1 Introdução

A fase de *design* é uma etapa crítica em projetos de desenvolvimento de software, pois nela são tomadas decisões técnicas que orientam o trabalho da equipe nas etapas subsequentes. Devido à importância desta etapa, decidimos analisar o *design* da biblioteca Joda-Time para entender como se deu a sua construção e sua evolução ao longo do tempo.

Joda-Time é uma biblioteca desenvolvida para a linguagem de programação Java com o objetivo de fornecer classes e métodos mais eficientes do que as classes do pacote *java.util* para o tratamento de datas e horários. A partir da versão 8 da linguagem de programação, esta biblioteca foi incorporada ao JDK (*Java Development Kit*). O principal motivo da escolha desta biblioteca para a análise foi o seu uso em um grande número de projetos, sendo classificada como a biblioteca mais utilizada na categoria de *date time utilities* no *Maven Repository*[1].

2.2 Estratégia de Análise

Diversas versões da biblioteca Joda-Time foram submetidas à análise de evolução da estrutura do seu *design*. Diferentes métricas foram utilizadas para construir uma compreensão do que aconteceu entre as versões consecutivas deste software. Começamos analisando métricas de tamanho e complexidade para determinar o aumento da funcionalidade entre as versões da biblioteca. Em seguida, aplicamos métricas de dispersão de alterações, acoplamento e coesão para entender se essas propriedades sofreram alterações significativas ao longo do tempo.

[1] https://mvnrepository.com/open-source/date-time-utilities

A escolha das métricas utilizadas neste trabalho foi baseada em análises realizadas para a biblioteca Apache Ant [2]. O trabalho anterior apresenta uma metodologia de análise das versões ao longo do tempo similar à realizada nesta pesquisa. As métricas de EVM e MQ não foram utilizadas devido ao seu baixo uso na indústria e no meio acadêmico, que é observado pelo reduzido número de publicações que fazem menção a elas. Estas métricas também foram desconsideradas pela incapacidade de extrairmos conclusões a partir delas. A Tabela 2.1 apresenta as métricas utilizadas nesta análise, separadas por categoria. As próximas subseções apresentam detalhes sobre estas métricas.

Tamanho e Complexidade	Dispersão das Mudanças	Coesão e Acoplamento
Número de pacotes	Número de *commits* de uma classe	CBO - *Coupling between objects*
Número de classes	Número de classes por *commit*	AFF - *Afferent coupling*
Número de atributos	Número de *commits* de um pacote	EFF - *Efferent coupling*
Número de métodos	Número de pacotes por *commit*	LCOM - *Lack of cohesion*
Número de métodos públicos		
Número de dependências		

Tabela 2.1: Métricas selecionadas para análise

2.3 Métricas de Tamanho e Complexidade

As métricas de tamanho e complexidade foram escolhidas para expor o crescimento do software sob análise ao longo das suas versões. Elas se baseiam em contagens e seus dados foram coletados usando a ferramenta de análise estática de código-fonte PF-CDA[2] ou acessando o código binário do programa Java utilizando a biblioteca BCEL4[3].

A métrica "número de pacotes" é responsável por contar o número de pacotes que possuem pelo menos uma classe. A métrica "número de classes" conta o número de classes do software, inclusive classes internas e aninhadas. A métrica "número de atributos" mede o número de atributos mantidos pelas classes, independente de seus modificadores (pública, protegida ou privada) e escopo (atributos de classe ou instância). A métrica "número de métodos" mede o número de métodos que são mantidos pelas classes, sem diferenciar os métodos por visibilidade e escopo. Essas métricas estão associadas ao tamanho do software.

[2]http://www.dependency-analyzer.org/
[3]http://commons.apache.org/proper/commons-bcel/

A métrica "número de métodos públicos" é responsável por contar os métodos que são públicos das classes que representam o código-fonte de uma versão do software, sejam eles métodos de classe ou instância. Esta medida está relacionada ao número de trocas de mensagens que podem ser feitas entre as classes do software e, dessa forma, é uma métrica da complexidade. O número de métodos públicos deve permanecer o mais baixo possível para que os softwares possam ter interações mais simples entre classes. Levando em consideração que os desenvolvedores precisam ter conhecimento sobre estas interações entre classes para que seja feita alguma alteração ou evolução no software, uma redução do número de métodos públicos geralmente facilita a manutenção do software e faz com que estes sejam menos suscetíveis a erros.

A métrica "número de dependências" mede o número de dependências de uso entre classes. Este número deve ser mantido o mais baixo possível, de maneira que as classes que compõem um software tenham poucas conexões com outras classes e, portanto, sejam mais independentes para desempenhar suas funções. É esperado que classes independentes sejam mais fáceis de manter ou evoluir porque poucas (se alguma) classes dependem das suas interfaces e do seu comportamento.

Após calcularmos as métricas descritas acima, utilizamos gráficos *boxplot* para analisar e entender a relação entre as entidades que compõem o software. Estes gráficos são uma representação visual dos valores das métricas, além de destacar os valores extremos (*outliers*) que elas podem assumir em algumas versões.

Tentando analisar a semelhança dos valores obtidos para estas métricas, analisamos os coeficientes de correlação entre cada par de métricas de tamanho. Com esta análise, podemos entender se duas métricas variam de forma semelhante (o valor de uma métrica aumenta quando o valor de uma segunda métrica aumenta – correlação positiva), se o comportamento é invertido (o valor de uma métrica é reduzido quando o valor de uma segunda métrica aumenta – correlação negativa) ou se não há relação entre os valores obtidos para as métricas. A convergência entre as métricas nos permite selecionar uma dentre aquelas que apresentam comportamento parecido para caracterizar o tamanho do software, diminuindo o número de métricas que precisam ser analisadas para caracterizar o software.

2.4 Métricas de Dispersão de Mudanças

As métricas de dispersão de mudanças são utilizadas para avaliar o alcance das alterações que precisam ser implementadas quando ocorre uma mudança no código-fonte, permitindo identificar a ocorrência de padrões indesejados no projeto e codificação do software, como *shotgun surgery* e *scattered functionality*.

O padrão *shotgun surgery* ocorre quando a implementação de uma mudança no código-fonte de uma classe desencadeia um efeito de alteração em cascata em outras classes. Isto indica que as classes não foram estruturadas de forma independente, exigindo a alteração de um grande número de classes para implementar uma mudança no software. Ainda que o número ideal de uma classe por alteração (que indica que apenas uma classe estaria envolvida na implementação da funcionalidade alterada) não possa ser obtido em grande parte dos casos, o número médio de classes alteradas em cada mudança deve ser pequeno.

O padrão *scattered functionality* ocorre quando uma mudança exige a alteração de classes dentro de um grande número de pacotes distintos, em vez da mudança afetar somente um pacote, que seria responsável pela funcionalidade alterada. Assim como no caso do *shotgun surgery*, o ideal é que o número de pacotes afetado pela adição de uma funcionalidade ou correção de um erro seja pequeno.

Para a identificação das alterações realizadas em uma versão do software, as informações são coletadas do sistema de controle de versão e cada *commit* é considerado uma alteração. Entretanto, os *commits* em um sistema de controle de versão estão relacionados à implementação de uma alteração, não às suas causas, justificativas ou razões. Em função dessa relação, os *commits* podem não estar perfeitamente alinhados com as alterações nos seguintes casos:

- um *commit* pode possuir alterações no código-fonte relacionadas a mais de uma alteração nas funcionalidades do software;
- muitos *commits* podem ser necessários para completar a alteração de todas as classes envolvidas em uma única funcionalidade no software.

Apesar destas limitações, os *commits* são os dados mais confiáveis sobre as alterações do software em domínio público. Deste modo, para medir a dispersão das alterações realizadas no software iremos obter o número de classes e pacotes modificados em cada *commit*. Para ocorrer o padrão *shotgun surgery*, devemos

observar um grande número de *commits* que envolvam um grande número de classes. Já o padrão *scattered functionality* é observado caso um grande número de *commits* altere um grande número de pacotes distintos.

Com este foco, a análise por meio de gráficos *boxplot*, considerando as classes e os pacotes afetados por *commits*, proporciona a identificação visual da concentração das amostras, além de sinalizar as ocorrências de valores extremos (*outliers*) do número de classes e pacotes afetados por um *commit*. A visualização sem *outliers*, por sua vez, permite ver os detalhes das distribuições de *commits*, que podem ficar obscurecidos pela escala necessária para que os valores extremos sejam visíveis.

2.5 Métricas de Coesão e Acoplamento

Para realizar a análise referente a evolução do acoplamento no software, consideramos uma extensão da métrica "*coupling between objects*" (CBO) para pacotes. Esta métrica foi apresentada por Chidamber e Kemerer [3] e calcula o acoplamento de uma classe A contabilizando o número de classes das quais A depende para a implementação de suas funcionalidades. A adaptação desta métrica para o acoplamento de pacotes leva em consideração as dependências de uso entre pacotes, que derivamos das dependências de uso entre classes: um pacote A depende de um pacote B se ao menos uma classe do pacote A depender de pelo menos uma classe do pacote B para desempenhar suas funções. O CBO para o pacote A é estimado como o número de pacotes que possuem classes com as quais as classes do pacote A tem dependência. O CBO deve ser reduzido em um *design*, ou seja, as classes e os pacotes devem ter uma baixa dependência de outras classes ou de outros pacotes.

A métrica "*Afferent coupling*" (AFF) é uma medida de acoplamento calculada a partir da contagem do número de classes que não estão em um pacote A que tem dependência com pelo menos uma classe do pacote A. Um pacote com AFF elevado tem alta responsabilidade em um projeto de software, pois diversas classes tem dependência das funcionalidades oferecidas por ele. O lado esquerdo da Figura 2.1 representa o tipo de dependência considerado no cálculo da métrica AFF, ou seja, as dependências de uma classe externa ao pacote A por classes do pacote A.

A métrica "*Efferent coupling*" (EFF) é uma medida de acoplamento calculada a partir da contagem do número de classes que não estão em um pacote A e com

as quais as classes presentes no pacote A tem dependência para implementar suas funcionalidades. Um pacote com EFF baixo apresenta maior independência do restante do software. Dessa maneira, se torna mais simples e fácil de testar e reaproveitar este pacote em outros softwares. O lado direito da Figura 2.1 representa o tipo de dependência considerado no cálculo da métrica EFF, ou seja, as dependências de uma classe do pacote A por classes externas ao pacote A.

Vale ressaltar que as métricas "*Efferent coupling*" (EFF) e "*Afferent coupling*" (AFF) consideram apenas as dependências diretas entre classes para o seu cálculo. Dessa forma, as dependências indiretas entre classes não são contabilizadas para o cálculo das métricas.

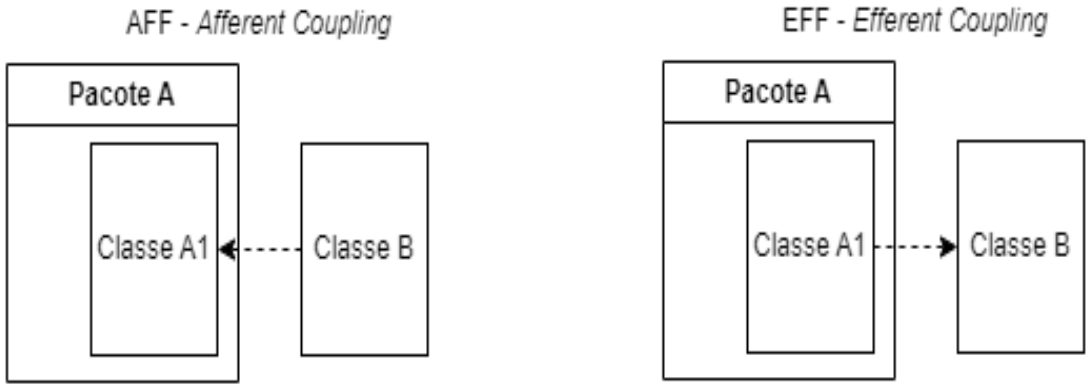

Figura 2.1: Métricas Afferent Coupling (AFF) e Efferent Coupling (EFF).

Para a análise de coesão de pacotes, escolhemos uma adaptação da métrica "*Lack of cohesion of methods*" (LCOM). A métrica LCOM foi proposta por Henderson-Sellers [4] e calcula a falta de coesão em uma classe de acordo com o uso conjunto de atributos por seus métodos. Quanto mais elevado o número de métodos acessando um elevado número de atributos da classe, menor será o valor da métrica. O cálculo considera um conjunto de métodos M_i, $1 \leq i \leq m$, que acessa um conjunto de atributos A_j, $1 \leq j \leq a$. Seja $\mu(A_j)$ o número de métodos que referencia um atributo A_j. LCOM é calculado conforme a Equação 2.1.

$$\mathrm{LCOM} = \frac{\frac{1}{a}(\sum_{j=1}^{a} \mu(A_j)) - m}{(1 - m)} \tag{2.1}$$

A métrica é estabelecida no intervalo [0, 1] e valores mais próximos a zero indicam maior coesão, ou seja, forte união entre os métodos e atributos da classe. Foi realizada a adaptação desta métrica para calcular a coesão de um pacote:

no lugar de métodos que fazem referência a atributos, utilizamos o número de classes do pacote que dependem de outras classes desse mesmo pacote. Podemos observar a falta de coesão de um pacote quando a maioria das suas classes não tem dependência com outras classes do mesmo pacote, gerando um alto valor de LCOM.

As métricas de coesão e acoplamento foram escolhidas para fazer a análise da relação entre classes e pacotes que formam o software. Porém, estas métricas são concorrentes. Se um arquiteto pretende maximizar a coesão do software, ele pode escolher colocar as classes em pacotes distintos, para que a função de cada pacote seja clara e puramente relacionada ao objetivo da classe. Porém, se o arquiteto pretende minimizar o acoplamento entre os pacotes, ele pode escolher colocar todas as classes em somente um pacote para que não haja nenhuma dependência entre pacotes. Essas abordagens não parecem plausíveis no desenvolvimento de software de larga escala, no qual os desenvolvedores procuram agrupar classes relacionadas em pacotes para facilitar o entendimento do software. Dessa maneira, se torna fundamental um equilíbrio entre acoplamento e coesão.

Foram utilizados gráficos *boxplot* para apresentar os valores dessas métricas e mostrar a ocorrência de *outliers*. Com isso, conseguimos analisar o valor da variação das métricas, da mediana e dos quartis de cada versão e fazer a comparação entre elas. Para a tentativa de analisar a semelhança dos valores observados, foi utilizada a análise dos coeficientes de correlação entre os pares de métricas de coesão e acoplamento. Como na análise das métricas de tamanho, a identificação de alta correlação entre métricas nos permite reduzir o universo de análise ao selecionar uma entre as que possuem comportamento semelhante para caracterizar um aspecto do *software*.

Para analisar a variação dos resultados destas métricas, ao considerar os pacotes de cada versão do software, utilizamos o teste estatístico não-paramétrico de *Wilcoxon-Mann-Whitney* com 95% de confiança. Dado o resultado ($p\text{-}value < 0{,}05$) podemos encontrar a probabilidade dos resultados das métricas de coesão e acoplamento dos pacotes apresentarem medianas iguais. Assim, identificamos se existem diferenças significativas entre os valores das métricas na evolução de cada versão.

2.6 Trabalhos Relacionados

A abordagem de pesquisa utilizada neste trabalho já foi aplicada em trabalhos anteriores do grupo de pesquisa de Engenharia de Software baseada em Buscas (*Search-based Software Engineering*, ou SBSE) do PPGI/UNIRIO. Entre eles, destacamos os trabalhos de Barros et al. [2] e Araújo [1].

Barros et al. [2] estudaram a implementação da ferramenta Apache Ant [4] para identificar se a otimização da distribuição das suas classes em pacotes pode ajudar a recuperar a arquitetura de um software de código aberto e larga escala. Eles apresentaram a evolução do software ao longo dos últimos treze anos, baseando a sua análise nas métricas de tamanho, complexidade, dispersão de mudanças, coesão e acoplamento descritas neste capítulo. Após a apresentação das métricas, os autores concluíram que são necessários melhores modelos para conduzir uma pesquisa baseada em otimização da estrutura do projeto de um software. Apesar das limitações dos modelos mostrados na análise, eles foram úteis para realçar certas limitações e permitir aos pesquisadores melhorar a base de conhecimento sobre bons princípios de *design* de software. Por exemplo, o dogma de aumentar a coesão e diminuir o acoplamento não parece funcionar se levado a cenários extremos. Dessa maneira, é necessário melhorar os modelos do que é considerado como um bom *design*.

Araújo [1] analisou três projetos de software – *JHotDraw*[5], *JEdit*[6] e *JUnit*[7] – de acordo com as métricas utilizadas neste trabalho. Ao avaliar a evolução das várias versões dos softwares, o autor concluiu que a classificação de bom exemplo de *design* não deveria ser aplicada para estes três softwares. O autor verificou que o *JEdit* apresentou a melhor evolução estrutural entre os três projetos selecionados. Também foi constatado que os resultados obtidos não eram uniformes. Sendo assim, não foi possível encontrar um padrão de *design* relacionado com a distribuição de classes em pacotes quando analisados os três softwares.

[4] https://ant.apache.org/
[5] https://sourceforge.net/projects/jhotdraw/
[6] http://www.jedit.org/
[7] https://junit.org/

2.7 Considerações Finais

Neste capítulo apresentamos a estratégia de análise que será aplicada sobre a biblioteca *Joda-Time* e as métricas que serão utilizadas nas análises apresentadas no próximo capítulo. No Capítulo 3, será apresentada a análise de diversas versões da biblioteca *Joda-Time* e a análise do *software* com base no seu código-fonte e nos *commits* registrados no seu sistema de controle de versões.

3. A Análise do Joda-Time

Neste capítulo apresentaremos uma análise detalhada da evolução estrutural da biblioteca *Joda-Time* com base nas métricas apresentadas no Capítulo 2.

3.1 Versões Analisadas

As versões do *Joda-Time* analisadas foram obtidas dos repositórios GitHub do projeto[1]. Foram encontradas 45 versões, lançadas no período entre 2011 e 2021. Levando em consideração as versões liberadas e focando na evolução estrutural do projeto, as versões que mantiveram o mesmo número de pacotes, classes, atributos, métodos e métodos públicos das versões imediatamente anteriores foram descartadas (0.9.0, 0.9.5, 1.0.0, 1.1.0, 1.2.0, 1.2.1, 1.3.0, 1.4.0, 1.5.0, 1.5.1, 1.5.2, 1.6.0, 2.8.1, 2.8.2, 2.9.1, 2.9.2, 2.9.3, 2.9.7, 2.9.8, 2.9.9, 2.10.1, 2.10.2, 2.10.3, 2.10.4, 2.10.5, 2.10.6, 2.10.8, 2.10.9, 2.10.10). Além disso, notamos que diferentes versões foram lançadas na mesma data. Nestes casos, selecionamos para a análise apenas as versões mais recentes de cada data.

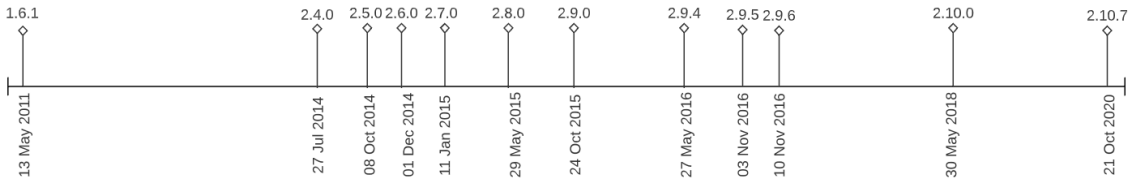

Figura 3.1: Linha do tempo de distribuição das versões da biblioteca Joda-Time.

A Figura 3.1 apresenta a *timeline* destas versões. Escolhemos a versão 1.6.1 como primeira versão do histórico por ser a última de uma série de versões com a

[1]https://github.com/JodaOrg/joda-time

mesma estrutura (16 versões descartadas) e a versão 2.10.7 como a última por ser a primeira de uma série de versões com a mesma estrutura (17 versões descartadas). Ao final, selecionamos 12 versões para análise entre as versões 1.6.1 e 2.10.7.

3.2 Análise das Métricas de Tamanho e Complexidade

A Tabela 3.1 mostra que o *Joda-Time* tem crescido lentamente ao longo do tempo desde a sua versão inicial. A primeira versão analisada (1.6.1) apresenta 222 classes distribuídas em 7 pacotes, enquanto a última versão analisada (2.10.7) possui 247 classes distribuídas em 7 pacotes. A primeira versão possui uma média de 31,71 classes por pacote, enquanto a última versão tem uma média de 35,28 classes por pacote. Com isso, podemos observar que o aumento no número de classes não foi muito significativo ao longo das versões.

Versão	Pacotes	Classes	Atributos	Métodos	Mét. Públicos	Dependências
1.6.1	7	222	330	3.987	3.250	1.574
2.4.0	7	239	377	4.363	3.488	1.717
2.5.0	7	243	386	4.404	3.516	1.729
2.6.0	7	244	388	4.411	3.518	1.734
2.7.0	7	245	391	4.422	3.525	1.744
2.8.0	7	246	393	4.425	3.527	1.745
2.9.0	7	246	393	4.435	3.533	1.751
2.9.4	7	247	398	4.439	3.535	1.753
2.9.5	7	247	398	4.445	3.543	1.755
2.9.6	7	247	398	4.446	3.543	1.755
2.10.0	7	247	398	4.484	3.580	1.784
2.10.7	7	247	398	4.486	3.581	1.785

Tabela 3.1: Tamanho e complexidade das versões do Joda-Time.

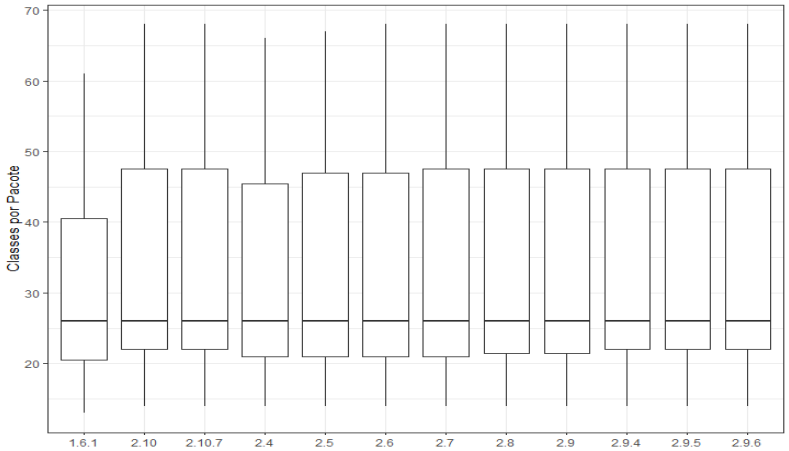

Figura 3.2: *Boxplot* da distribuição de classes por pacote do Joda-Time.

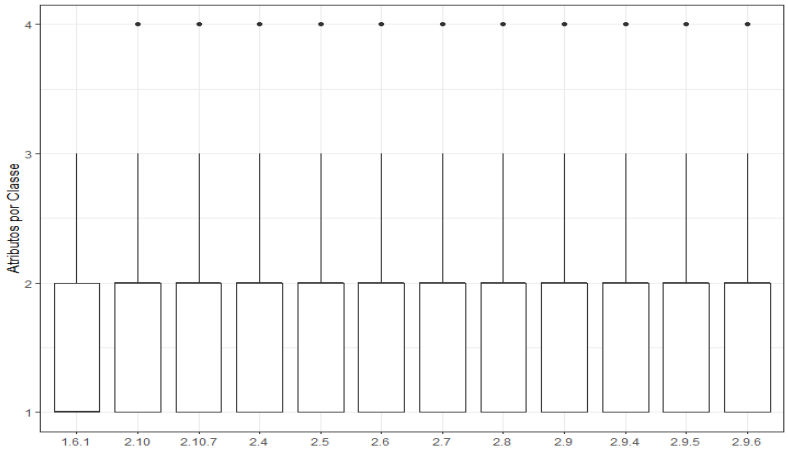

Figura 3.3: *Boxplot* da distribuição de atributos por classe do Joda-Time.

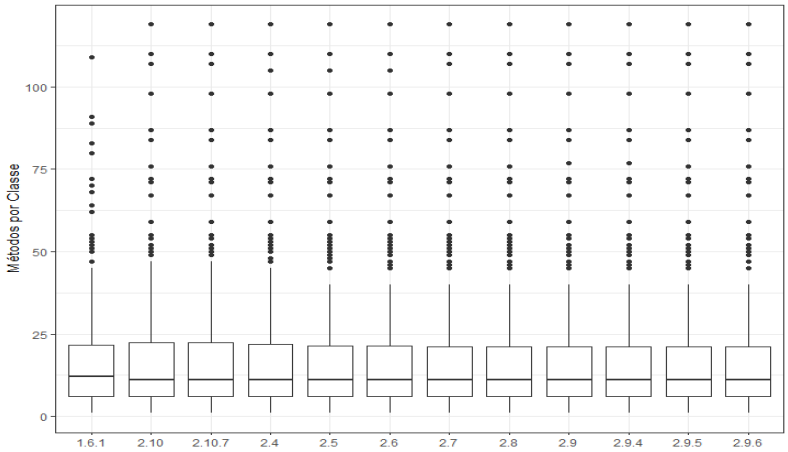

Figura 3.4: *Boxplot* da distribuição de métodos por classe do Joda-Time.

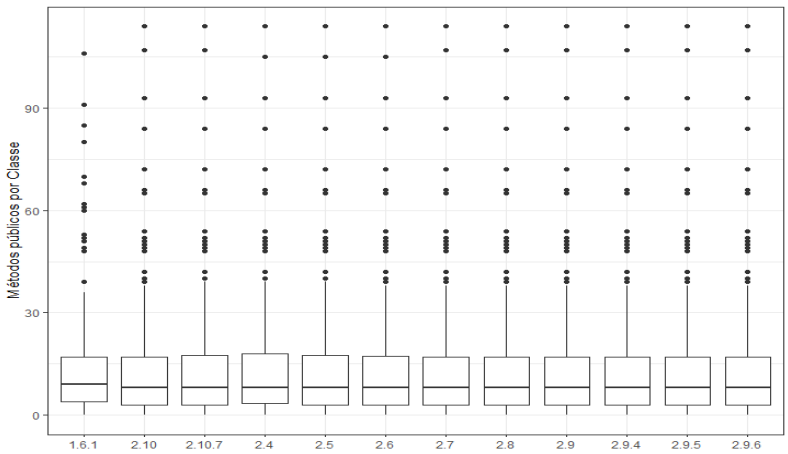

Figura 3.5: *Boxplot* da distribuição de métodos públicos por classe do Joda-Time.

A Figura 3.2 apresenta a distribuição de classes por pacote nas 12 versões da biblioteca. A Figura 3.3 apresenta a distribuição de atributos por classe nas

diferentes versões. A Figura 3.4 apresenta a distribuição de métodos por classe. Por fim, a Figura 3.5 apresenta a distribuição dos métodos públicos por classe.

Os *boxplots* permitem uma análise visual do valor central, dispersão, simetria e valores discrepantes (*outliers*) do conjunto de dados. O valor central é representado na linha central do retângulo (mediana). A dispersão é representada pelo intervalo inter-quartílico, que é a diferença entre o terceiro e primeiro quartil, ou seja, a diferença entre as linhas de base e topo da vela central do *boxplot*. Uma distribuição simétrica terá a linha da mediana no meio do retângulo. Quando a linha da mediana estiver mais próxima do primeiro quartil, o conjunto de dados será assimétrico positivo; quando esta linha estiver mais próxima do terceiro quartil, o conjunto de dados será assimétrico negativo.

Pelos gráficos acima, observa-se pouca variação ao longo do tempo em todas as características da biblioteca, seja o seu número de classes, atributos, métodos e métodos públicos. O tamanho do software não mudou muito entre as versões, ou seja, não houve uma versão com grande crescimento e posterior redução. Com relação aos atributos, percebe-se que a grande maioria das classes tem um único atributo e que poucas classes chegam a ter quatro atributos. Observa-se um cenário diferente para os métodos, com a grande maioria das classes apresentando poucos métodos (entre um e vinte métodos, sendo zero a quinze públicos), mas algumas classes concentrando mais de trinta métodos e chegando a mais de uma centena de métodos públicos em alguns casos.

3.2.1 Análise das Métricas de Dispersão de Mudanças

O repositório do sistema de controle de versão armazena as informações sobre as alterações realizadas no código-fonte (*commits*) em *logs*. A partir destes *logs*, foram coletadas informações sobre o número de *commits* e os desenvolvedores que realizaram estas alterações no código. Os *commits* contêm adições, exclusões e modificações no código-fonte da biblioteca *Joda-Time*.

A Tabela 3.2 apresenta informações sobre o número de classes envolvidas nos *commits* ao longo do tempo. A coluna "*Commits*" apresenta o número de *commits* realizados antes de liberar cada versão. No total, foram realizados cerca de 1.244 *commits*. Podemos ver que o maior número de *commits* ocorreu logo na primeira versão, que apresenta os *commits* das 16 versões anteriores a ela e que foram descartadas por compartilharem a mesma estrutura. A coluna "*Commits* de uma classe" apresenta o número de *commits* que modificaram apenas uma única

Versão	Commits	Commits de uma classe	(%) commits de uma classe
1.6.1	950	401	42,21%
2.4.0	166	70	42,16%
2.5.0	14	1	7,14%
2.6.0	7	3	42,85%
2.7.0	6	3	50,00%
2.8.0	15	9	60,00%
2.9.0	15	9	60,00%
2.9.4	16	4	25,00%
2.9.5	7	2	28,57%
2.9.6	5	3	60,00%
2.10.0	28	16	57,14%
2.10.7	15	9	60,00%

Tabela 3.2: Número de commits em classes das versões do Joda-Time.

classe. Já a coluna "(%) de *commits* de uma classe" apresenta esse número como um percentil do número total de *commits* da versão.

O número de *commits* envolvendo apenas uma classe variou entre as versões, tendo o seu maior valor na versão 1.6.1 e o menor na versão 2.5.0. O percentual de *commits* envolvendo apenas uma classe teve uma pequena variação ao longo das versões. Apenas nas versões 2.5.0, 2.9.4 e 2.9.5 foram observados valores de variação dissonantes das demais versões. De modo geral, observamos um grande número de *commits* de apenas uma classe, o que é positivo para *design* do software e afasta a recorrência do padrão *shotgun surgery*.

Versão	Commits	Mediana	Média	Desvio Padrão
1.6.1	950	2	5,79	19,43
2.4.0	166	2	5,39	32,45
2.5.0	14	2	2,50	1,01
2.6.0	7	2	2,42	1,81
2.7.0	6	1,50	1,50	0,54
2.8.0	15	1	1,53	0,74
2.9.0	15	1	2,00	2,29
2.9.4	16	2	3,93	7,80
2.9.5	7	2	2,42	1,27
2.9.6	5	1	1,80	1,30
2.10.0	28	1	3,28	8,67
2.10.7	15	1	1,86	1,59

Tabela 3.3: Estatísticas de commits em classes das versões do Joda-Time.

A Tabela 3.3 nos fornece informações a respeito dos *commits* de cada versão em relação às classes afetadas por estes commits. Podemos observar que é calculada a mediana, média e desvio padrão do número de classes afetadas pelos *commits* em cada versão do software. O valor elevado de desvio padrão e média na versão 2.4.0 mostra a ocorrência de poucos *commits* envolvendo um grande número de classes, embora a maior parte dos *commits* envolva apenas uma ou duas classes, como pode ser visto pela mediana. Assim, é provável que a versão 2.4.0 tenha promovido uma grande refatoração do projeto, registrada nos *commits* que envolvem um grande número de classes.

Versão	Commits	Commits de um pacote	(%) commits de um pacote
1.6.1	950	596	62,73%
2.4.0	166	86	51,80%
2.5.0	14	2	14,28%
2.6.0	7	4	57,14%
2.7.0	6	3	50,00%
2.8.0	15	10	66,66%
2.9.0	15	9	60,00%
2.9.4	16	4	25,00%
2.9.5	7	4	57,14%
2.9.6	5	3	60,00%
2.10.0	28	18	64,28%
2.10.7	15	10	66,66%

Tabela 3.4: Número de commits em pacotes das versões do Joda-Time.

A Tabela 3.4 apresenta as informações sobre o número de pacotes envolvidos nos *commits* ao longo do tempo. A coluna "Commits de um pacote" contabiliza o número de *commits* que alteraram o código-fonte de classes de um único pacote. Já a coluna "(%) de *commits* de um pacote" apresenta esse número como uma porcentagem do número total de *commits* em cada versão. Podemos observar que a média de *commits* que alteraram somente um pacote durante o desenvolvimento foi de 52,98%, o que significa que a dispersão dos *commits* em pacotes foi controlada, com exceção das versões 2.9.4 e 2.5.0 em que a porcentagem de *commits* de um único pacote foi baixa (25,00% e 14,28%, respectivamente). No entanto, cabe notar que estas duas versões têm poucos *commits*, especialmente quando comparadas com as primeiras versões do software.

A Tabela 3.5 fornece informações estatísticas sobre os *commits* e os pacotes envolvidos neles. Podemos observar que na segunda versão do projeto (2.4.0) temos o desvio padrão mais elevado, indicando que houve um maior número

Versão	Commits	Mediana	Média	Desvio Padrão
1.6.1	950	1	1,72	1,58
2.4.0	166	1	1,87	3,16
2.5.0	14	2	1,85	0,36
2.6.0	7	1	1,57	0,78
2.7.0	6	1,5	1,50	0,54
2.8.0	15	1	1,33	0,48
2.9.0	15	1	1,60	1,05
2.9.4	16	2	1,87	0,61
2.9.5	7	1	2,00	1,41
2.9.6	5	1	1,60	0,89
2.10.0	28	1	1,64	1,54
2.10.7	15	1	1,46	0,74

Tabela 3.5: Estatísticas sobre commits em pacotes das versões do Joda-Time.

de mudanças em pacotes diferentes durante as versões iniciais do projeto, o que também pode ser fundamentado pelo número de *commits* nessas versões. Esta observação coaduna com a possível refatoração observada anteriormente, dado que não apenas muitas classes foram alteradas, mas também estas classes estavam distribuídas entre diferentes pacotes.

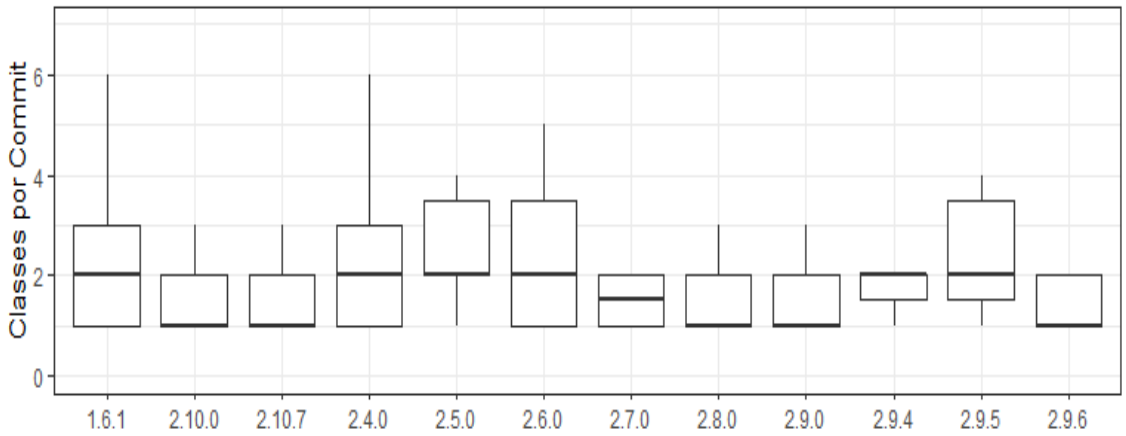

Figura 3.6: *Boxplot* da distribuição de classes por commits do Joda-Time.

A Figura 3.6 representa o número de classes por *commit* em cada versão. Nela observamos que 50% dos *commits* alteraram somente uma classe nas versões 2.8.0, 2.9.0, 2.10.0 e 2.10.7; os demais *commits* alteraram duas ou três classes. Já nas versões 1.6.1 e 2.4.0 temos uma disparidade maior que nas outras versões: 25% dos *commits* afetaram entre 1 e 2 classes e 50% afetaram entre duas ou três classes.

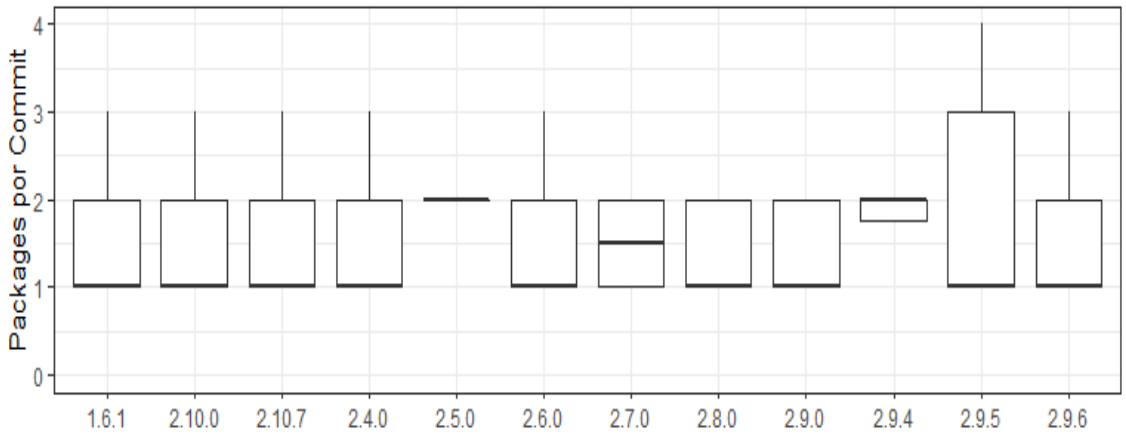

Figura 3.7: *Boxplot* da distribuição de pacotes por commits do Joda-Time.

A Figura 3.7 representa o número de pacotes por *commit* em cada versão. Nela, observamos que 50% dos *commits* das versões 1.6.1, 2.4.0, 2.6.0, 2.8.0, 2.9.0, 2.9.5, 2.9.6, 2.10.0 e 2.10.7 alteraram somente um pacote. Já na versão 2.5.0, 100% dos *commits* alteraram dois pacotes. A versão 2.9.5 apresenta a maior disparidade: 50% dos *commits* alteraram apenas um pacote, 25% dos *commits* alteraram entre um e três pacotes e os 25% de *commits* restantes alterarem de três a quatro pacotes.

3.2.2 Análise das Métricas de Coesão e Acoplamento

A Tabela 3.6 apresenta a média de cada métrica de coesão e acoplamento por versão. A distribuição destas métricas é apresentada nos boxplots das Figuras 3.8, 3.9, 3.10 e 3.11.

Versão	CBO	AFF	EFF	LCOM	CBO	AFF	EFF	LCOM
1.6.1	3,42	41,14	26,28	0,91	-	-	-	-
2.4.0	3,42	44,28	26,42	0,92	1,00	0,74	0,94	0,84
2.5.0	3,42	44,28	26,42	0,93	1,00	1,00	1,00	0,84
2.6.0	3,42	44,42	26,42	0,93	1,00	0,94	1,00	1,00
2.7.0	3,42	44,85	26,42	0,93	1,00	0,89	1,00	0,84
2.8.0	3,42	44,85	26,42	0,93	1,00	1,00	1,00	1,00
2.9.0	3,42	45,00	26,42	0,93	1,00	0,94	1,00	0,84
2.9.4	3,42	45,00	26,42	0,93	1,00	1,00	1,00	1,00
2.9.5	3,42	45,14	26,42	0,93	1,00	0,89	1,00	0,84
2.9.6	3,42	45,14	26,42	0,93	1,00	1,00	1,00	0,84
2.10.0	3,42	45,28	26,71	0,93	1,00	0,89	0,84	1,00
2.10.7	3,42	45,28	26,71	0,93	1,00	1,00	1,00	1,00

Tabela 3.6: Métricas de acoplamento e coesão para as versões do Joda-Time.

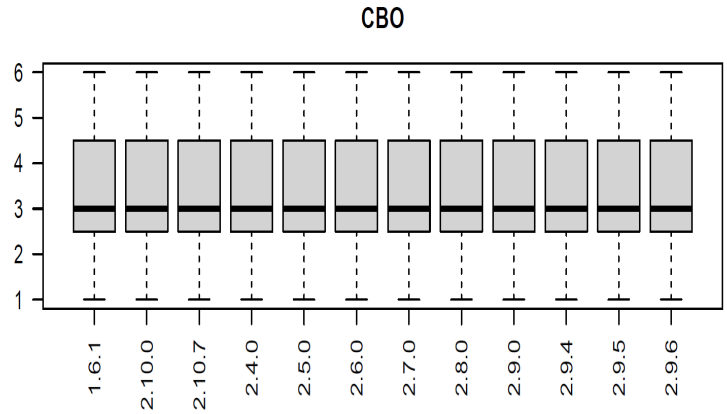

Figura 3.8: *Boxplot* da métrica CBO.

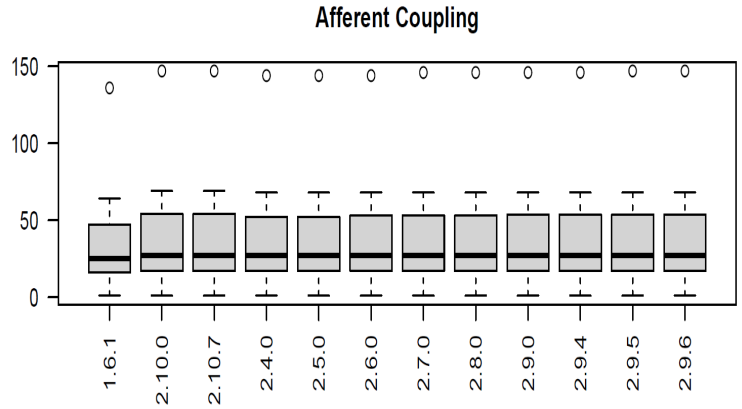

Figura 3.9: *Boxplot* da métrica Afferent Coupling (AFF).

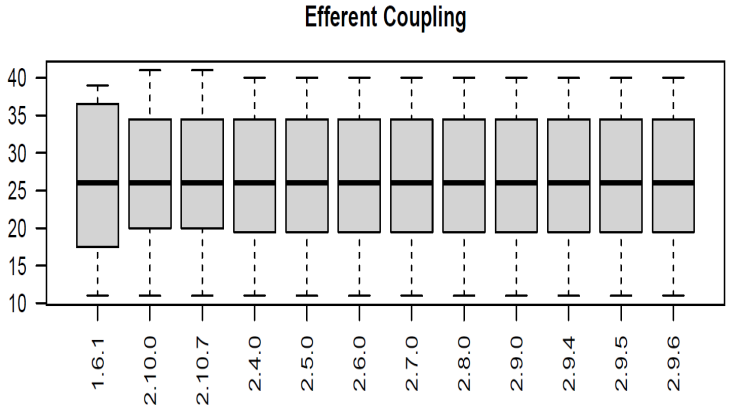

Figura 3.10: *Boxplot* da métrica Efferent Coupling (EFF).

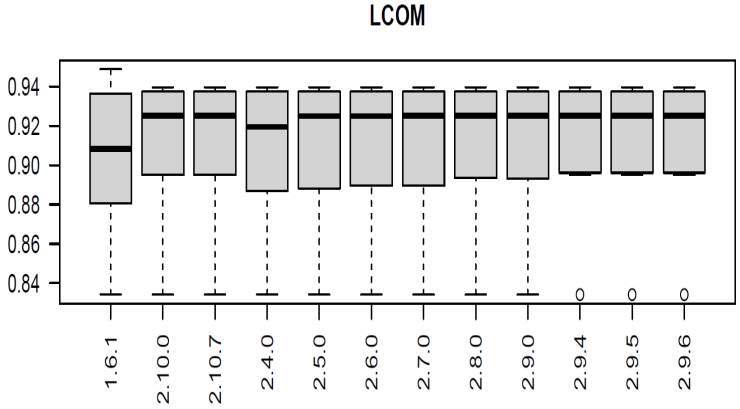

Figura 3.11: *Boxplot* da métrica *Lack of Cohesion* (LCOM).

A primeira coluna da Tabela 3.6 ("CBO") apresenta a média da métrica CBO em cada pacote que compõe a versão, a coluna AFF apresenta a média do acoplamento aferente, a coluna EFF apresenta a média do acoplamento eferente e a coluna LCOM apresenta a média da falta de coesão por classe.

Podemos observar a falta de variação nos valores da coluna referente ao CBO, indicando que houve pouca (se alguma) variação na estrutura de dependências entre as classes ao longo das versões. Por outro lado, o valor médio do acoplamento aferente cresceu ao longo das versões, indicando que houve um crescimento nas dependências de classes entre pacotes. Também podemos observar que o acoplamento eferente não apresentou uma variação grande. Em função disso, podemos dizer que o número de pacotes independentes se manteve estável entre as diferentes versões. Ao olharmos para a coluna LCOM, podemos observar que o número de pacotes com falta de coesão foi alto. A falta de coesão de um pacote é observada quando a maioria de suas classes não depende de outras classes do mesmo pacote.

As colunas à direita na Tabela 3.6 mostram o *p-value* resultante da aplicação de testes de *Wilcoxon-Mann-Whitney* entre as colunas do lado esquerdo da tabela. Os valores altos para os *p-values* mostram que não é possível identificar variações significativas nas medianas dos valores das métricas para os pacotes, quando comparamos diferentes versões. Isto indica que a coesão e o acoplamento não mudaram significativamente ao longo do desenvolvimento do software.

A Tabela 3.7 apresenta a correlação não-paramétrica (coeficiente de *Spearman*) entre as três métricas que apresentam alguma variação na Tabela 3.6. Optamos por não utilizar o coeficiente de correlação de *Pearson* pois os dados analisados

	AFF	EFF	LCOM
AFF	1,00	0,57	-0,14
EFF		1,00	0,21
LCOM			1,00

Tabela 3.7: Análise de correlação (Spearman) entre as métricas do Joda-Time.

não tem proximidade com a distribuição normal. Assim, utilizamos o coeficiente de *Spearman*, que é uma correlação não-paramétrica e, em função disso, não exige que os dados submetidos a ela tenham distribuição próxima à normal. Podemos observar uma correlação mediana e positiva entre as métricas AFF e EFF, como era de se esperar entre duas métricas que representam a mesma característica: o acoplamento. Assim, a medida que aumenta o número de classes de um pacote A que dependem de classes de outros pacotes, aumenta o número de classes presentes em outros pacotes que dependem de classes do pacote A. A correlação entre as métricas AFF/EFF e LCOM, por outro lado, é fraca, variando de sinal entre as métricas. Assim, não existe uma relação forte entre coesão e acoplamento no software sob análise.

3.3 Trabalhos Relacionados

Araújo [1] analisou três projetos de software considerados bons exemplos de *design*: `JHotDraw`[2], `JEdit`[3] e `JUnit`[4]. O autor chegou a conclusão de que todos tiveram um crescimento constante ao longo de suas diversas versões e a distribuição de dados ou funcionalidades entre classes se manteve estável entre as versões. Farzat et al. [2] estudaram a implementação da ferramenta `Apache Ant` [5] e chegaram à conclusão de que o projeto teve um crescimento constante ao longo de seus anos e a distribuição de dados ou funcionalidades entre as classes se manteve estável entre as diferentes versões. Assim, percebemos que as conclusões observadas para trabalhos anteriores reafirmam o crescimento constante dos projetos de software e, nos casos estudados, a estabilidade da distribuição dos atributos e métodos ao longo de suas diferentes versões.

[2]https://sourceforge.net/projects/jhotdraw/
[3]http://www.jedit.org/
[4]https://junit.org/
[5]https://ant.apache.org/

Fazendo a comparação dos resultados relacionados a tamanho e complexidade de software do `Joda-Time` com as outras análises (`JHotDraw`, `JEdit`, `JUnit` e `Apache Ant`), vemos que diferentemente dos outros o `Joda-Time` não cresceu de forma significativa ao longo de suas versões e manteve aproximadamente o mesmo tamanho até a última versão analisada. Por outro lado, ao compararmos as métricas de complexidade, observamos que a distribuição de dados ou funcionalidades entre classes se manteve estável ao longo das versões analisadas, em alinhamento com os outros software.

A análise das informações obtidas do sistema de controle de versão mostrou um comportamento similar ao encontrado na análise do `JEdit`, que não demonstrou indícios de *shotgun surgery*, ou seja, foi evidenciado um baixo número de classes alteradas em cada *commit*. `Joda-Time` também não demonstrou o efeito do anti-padrão *scattered functionality*, pois grande parte dos *commits* envolvia poucos pacotes. É importante destacar que estas características são muito diferentes do observado no `JHotDraw` e no `JUnit`, ambos apresentando indícios de *shotgun surgery* e *scattered functionality*.

Analisando as métricas de coesão e acoplamento, percebemos que o `Joda-Time` se comportou de forma parecida com o `JEdit` em relação à coesão, pois as métricas se mantiveram estáveis ao longo de suas versões. Já as métricas de acoplamento tiveram um resultado mais próximo ao do `JHotDraw` para o qual foram obtidos resultados neutros. O caso do `Joda-Time` é marcante, pois o valor da métrica de acoplamento não se alterou até a segunda casa decimal ao longo das versões, provavelmente em função do número reduzido de alterações e do tamanho reduzido do próprio projeto.

3.4 Considerações Finais

Este capítulo apresentou a análise do *design* do software `Joda-Time`, comparando-o com outros software analisados anteriormente utilizando as mesmas perspectivas. O próximo capítulo apresenta as conclusões deste trabalho, suas contribuições, limitações e perspectivas de trabalho futuro.

4. Conclusão

4.1 Contribuições

A principal contribuição deste trabalho foi a aplicação das métricas de tamanho, complexidade, dispersão de mudanças, coesão e acoplamento utilizadas para analisar a biblioteca `Apache Ant` [2] no contexto de uma nova biblioteca. Em função desta aplicação, temos um melhor entendimento de como se deu a evolução da biblioteca `Joda-Time` ao longo do tempo e de suas versões. Observamos que a biblioteca cresceu pouco, seja em tamanho ou complexidade, ao longo do tempo. Vemos também que as métricas de coesão, acoplamento e dispersão de mudanças se mantiveram estáveis ao longo deste período.

Com isso, chegamos à conclusão de que a biblioteca possui um bom *design* e teve uma boa evolução, dado que durante a sua evolução não apresentou padrões indesejados, como *Shotgun Surgery*, e o crescimento não foi desordenado, mostrando um cuidado durante o desenvolvimento de novas funcionalidades. Porém, a análise também mostrou que o *Joda-Time* pode não ser um fiel representante dos programas desenvolvidos em Java, no geral, dado que novas funcionalidades não foram introduzidas com muita frequência.

4.2 Limitações

As análises realizadas neste projeto de pesquisa utilizaram uma biblioteca desenvolvida com a linguagem de programação Java. Dessa forma, não é possível podemos estender os resultados encontrados para outras bibliotecas, escritas com outras linguagens de programação. Além disso, `Joda-Time` é uma biblioteca pequena, com poucas dezenas de classes. Em função disso, pode haver vieses

nas conclusões, em particular no que se refere às alterações realizadas de forma conjunta entre pacotes.

Não foi possível extrair os dados de maneira eficiente através de alguma API do GitHub que, por exemplo, retornasse os dados necessários para o cálculo das métricas. Assim, tivemos que utilizar uma automação que fizesse um tratamento prévio das informações coletadas do GitHub para que pudéssemos utilizar no cálculo das métricas.

4.3 Trabalhos futuros

Estudos futuros podem ser planejados e realizados a partir dos resultados apontados neste trabalho. Uma sugestão seria utilizar as métricas deste trabalho para realizar um novo estudo com outras bibliotecas escritas na linguagem de programação Java. Outra análise interessante seria utilizar as métricas deste trabalho para bibliotecas escritas em outras linguagens de programação e comparar os resultados obtidos.

Bibliografia

[1] ARAUJO JUNIOR, L. A. O. D. *Extended modularization quality: Uma métrica para orientar a distribuição de classes em pacotes no design de software*. Programa de Pós-Graduação em Informática, Universidade Federal do Estado do Rio de Janeiro (PPGI/UNIRIO), Outubro. 2022.

[2] BARROS, M., FARZAT, F., TRAVASSOS, G. "Learning from optimization: A case study with apache ant", *Information and Software Technology* v. 57, Agosto. 2014.

[3] CHIDAMBER, S. R., KEMERER, C. F. "A metrics suite for object oriented design", *IEEE Trans. Softw. Eng.* v. 20, n. 6, pp. 476–493, June. 1994.

[4] HENDERSON-SELLERS, B., *Object-oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 1996.