

3 UMA ANÁLISE DE SOFTWARE RECONHECIDOS PELO DESIGN DE QUALIDADE

Este capítulo apresenta uma análise de softwares que são reconhecidos pela qualidade do seu design. Foi definida uma estratégia de análise considerando métricas, gráficos e estatística inferencial para verificar o design e a evolução das versões liberadas destes softwares.

3.1 Introdução

Conforme mencionado no Capítulo 1, a fase de *design* é crítica para projetos de desenvolvimento de software, pois nela são tomadas decisões técnicas que orientam o desenvolvimento do sistema [255]. Conforme a metodologia apresentada na Seção 1.6, foi realizada uma busca, seleção e análise de softwares identificados como bons exemplos de *design*. Após buscas em publicações científicas que apontaram para os softwares candidatos, eles foram analisados sob a ótica da evolução da estrutura do seu código-fonte e do histórico de versões liberadas.

Este capítulo apresenta o processo utilizado para realizar a seleção e análise dos softwares que serão utilizados na composição da métrica de *design*. Este capítulo é composto por sete seções, começando com esta introdução, passando pela estratégia de seleção dos softwares que serão analisados, seguindo pela definição da estratégia de análise, chegando nas análises propriamente ditas e finalizando com as considerações finais sobre a análise.

3.2 Seleção dos Softwares de Referência

Muitas publicações científicas foram analisadas para realizarmos o mapeamento sistemático apresentado no Capítulo 2. Como parte destas análises, verificamos que as publicações selecionavam softwares que eram utilizados como modelo para validar as suas propostas. Estes softwares incluem o ArgoUML, Azureus, JDatePicker, JEdit,

JFreeChart, JHotDraw, JTDS, JUnit, JValue, JXLS, QuickUML e Log4J. Em particular, Dirk Riehle [227] analisou o JHotDraw e o classificou como um bom padrão de *design*. Em 2008, ele publicou que JEdit, JHotDraw e JUnit poderiam ser utilizados como *benchmark* para novos sistemas, considerando as qualidades do seu *design*¹.

JHotDraw é uma biblioteca desenvolvida na linguagem de programação Java e utilizada no desenvolvimento de ferramentas de edição de gráficos bi-dimensionais. Em 2010, Kessentini et al. [158] reforçaram a opinião de Dirk Riehle analisando a versão 7.1 e a classificaram como representante da boas práticas de *design* e programação. Em 2013, Bavota et al. [34] analisaram a versão 6.0 sob a ótica de acoplamento. Em 2014, Kaur et al. [156] utilizaram mineração de dados em métricas de código-fonte propostas por Li e Henry [178], Chidamber e Kemerer [73] e Bertrand Meyer [195] na análise das versões 5.2 e 5.3. Em 2015, De Lucia et al. [91] ratificaram a boa qualidade do *design* e avaliaram as versões 5.1, 5.2, 5.4.1 e 6.0. Em 2015, Ebad e Ahmed [99] analisaram as versões 5.1, 5.2 e 5.3 para verificar a estabilidade da arquitetura de softwares orientados a objetos. Em 2016, Ouni et al. [214] avaliaram as versões 6.1 e 6.2 em sua pesquisa. Em 2018, Ghosh e Kuttal [123] avaliaram a versão 7.0.6 para detectar clones de código. Em 2020, Speicher [248] avaliou a versão 5.1 sob a ótica de impactos (negativos e positivos) de potenciais violações de refatorações. Outras pesquisas utilizaram o JHotDraw como referência sem informar a versão analisada [80][218].

JEdit é um projeto *open-source*, desenvolvido na linguagem de programação Java, que implementa um editor de texto para programadores avançados e que representa o esforço de trabalho realizado por mais de 100 pessoas. O código-fonte encontra-se no repositório do SourceForge². Diversas publicações apresentam o JEdit em estudos experimentais que precisam de uma referência de qualidade em projeto de software. Em 2009, Alan e Catal [13] utilizaram o JEdit (versão não informada) na detecção de *outliers* em conjuntos de dados de medição de software que afetam o desempenho dos modelos de previsão de falha de software construídos com base nesses conjuntos de

¹<https://dirkriehle.com/>

²<https://sourceforge.net/projects/jedit/>

dados. Em 2011, Soares et al. [246] utilizaram o JEdit (versão não informada) para analisar a refatoração de código-fonte. Em 2013, Bavota et al. [34] analisaram a versão 2.4.1 para avaliar a percepção de acoplamento dos desenvolvedores. Em 2014, Kaur et al. [156], assim como fizeram para o JHotDraw, analisaram o JEdit nas versões 3.1.0 e 3.2.0. Em 2018, Choi et al. [78] utilizaram o JEdit (versão não informada) em uma investigação sobre o relacionamento entre o padrão de refatoração e métricas de alterações de código.

JUnit é uma biblioteca para desenvolvimento de casos de teste de unidade para software escrito na linguagem de programação Java. Assim como os dois softwares acima, o JUnit na versão 3.8.0 foi classificado como *benchmark* por Riehle [227] e como referência em outras pesquisas. Em 2016, Oruc et al. [211] utilizaram as versões 3.8.0 e 4.1.0 em sua pesquisa para detecção de padrões de *design*. Em 2017, Rani e Chhabra [224] utilizaram as versões 3.8.2, 4.8.1, 4.11.0 e 4.12.0 para detecção de *code smells*. Em 2019, Dwivedi et al. [95] analisaram a versão 5.0 para encontrar um padrão de *design*. Em 2020, Prabha e Shivakumar [218] utilizaram o JUnit (versão não informada) na melhoria de qualidade de *design*.

Para as publicações que não apresentaram a versão utilizada dos respectivos softwares, foram enviados e-mails aos autores perguntando sobre as versões que foram utilizadas. Também foram enviados e-mails aos autores de publicações que consideraram versões que não estavam disponíveis no repositório que foi utilizado nesta pesquisa. Diante desta tentativa de contato, somente não foram adquiridas as versões 6.1 e 6.2 do JHotDraw. Em relação as versões mencionadas do JUnit, somente a versão 5.0 não foi considerada porque foi verificado que o projeto foi subdividido em três subprojetos, tornando a sua arquitetura incomparável com as versões anteriores.

Considerando a boa classificação quanto a qualidade do *design* e do uso dos três softwares (JHotDraw, JEdit e JUnit) em diversas publicações, estes foram selecionados para serem submetidos a análise de evolução da estrutura do seu *design* entre as suas versões. Durante o desenvolvimento de um software, muitas versões podem ser liberadas com o único intuito de corrigir problemas da versão anterior. Estas versões

de correção podem não acrescentar muitos novos elementos para a estrutura do *design* do software. Devido ao grande número de versões disponíveis para os três softwares selecionados, versões que não apresentam muita variação em sua estrutura serão descartadas. Os critérios de análise consideram o número de pacotes, o número de classes e as dependências entre classes e pacotes. Caso uma versão não apresente mudanças no número de pacotes, no número de classes e no número médio de dependências entre classes com relação a sua anterior, esta versão será desconsiderada da análise, sendo mantida a versão mais recente que apresentar mudanças nos critérios.

3.3 Estratégia da análise

Para analisar o *design* do JHotDraw, JEdit e JUnit foram utilizadas diferentes métricas para construir uma compreensão do que aconteceu entre versões consecutivas destes softwares. Começamos analisando métricas de tamanho e complexidade para determinar o aumento da funcionalidade entre as versões de cada sistema. A partir dessa análise, aplicamos métricas de dispersão de alterações, acoplamento e coesão para entender se essas propriedades mudaram significativamente ao longo do tempo.

A seleção das métricas foi baseada em um estudo semelhante realizado para o Apache Ant [30], que apresentou uma estratégia de análise estrutural das versões ao longo do tempo que se assemelha com a necessidade de análise desta pesquisa. Foram desconsideradas as métricas de elegância, MF, EVM e CS, devido a sua baixa frequência de utilização no meio acadêmico e na indústria, evidenciada pelo reduzido número de publicações em que estas métricas são mencionadas. A Tabela 3.1 apresenta as métricas selecionadas separadas por categorias.

Tabela 3.1: Métricas selecionadas.

Tamanho e complexidade	Dispersão das mudanças	Coesão e acoplamento
Número de pacotes	Número de <i>commits</i> de uma classe	CBO - Coupling between objects
Número de classes	Número de classes por <i>commit</i>	AFF - Afferent coupling
Número de atributos	Número de <i>commits</i> de um pacote	EFF - Efferent coupling
Número de métodos	Número de pacotes por <i>commit</i>	LCOM - Lack of cohesion
Número de métodos públicos		
Número de dependências		

3.3.1 Métricas de Tamanho e Complexidade

As métricas de tamanho e complexidade foram selecionadas para descrever o crescimento de um software ao longo das suas versões. Elas são contagens simples e seus dados foram coletados usando a ferramenta de análise estática PF-CDA³ ou acessando o código binário executado na máquina virtual Java usando o BCEL⁴.

A métrica “número de pacotes” conta o número de pacotes que possuem pelo menos uma classe. A métrica “número de classes” conta o número de classes do sistema, incluindo classes internas e aninhadas. A métrica “número de atributos” conta o número de atributos mantidos pelas classes, independente de sua visibilidade (pública, protegida ou privada) e escopo (atributos de classe ou instância). A métrica “número de métodos” conta o número de métodos mantidos pelas classes, sem distinção em relação a sua visibilidade e escopo. Essas métricas estão relacionadas ao tamanho do sistema, como uma medida de dados (para atributos), uma medida funcional (para métodos) ou uma medida mista (dados e funções observadas juntas, para pacotes e classes).

A métrica “número de métodos públicos” conta apenas os métodos públicos mantidos pelas classes que compõem o código-fonte de uma versão, independente de serem métodos de classe ou instância. Esta medida está relacionada ao número de mensagens que podem ser trocadas entre as classes do sistema e, portanto, é uma medida da complexidade. O número de métodos públicos deve ser mantido o mais baixo possível para permitir que os sistemas tenham protocolos de interação entre classe mais simples. Dado que os desenvolvedores precisam entender esses protocolos para alterar ou evoluir o sistema, uma redução do número de métodos públicos tende a levar a sistemas menos propensos a erros e mais fáceis de manter.

A métrica “número de dependências” conta o número de dependências de uso entre classes e se relaciona com o número de links entre classes e, indiretamente, entre pacotes. O número de dependências deve ser mantido tão baixo quanto possível que

³<http://www.dependency-analyzer.org/>

⁴<http://commons.apache.org/proper/commons-bcel/>

as classes componentes de um sistema tenham poucas conexões com outras classes e, portanto, sejam mais autônomas para desempenhar suas funções.

Após a aplicação das métricas de tamanho e complexidade e sua apresentação de forma tabular e individualizada, foram utilizados gráficos *boxplot* para analisar a relação entre as entidades do software que foram medidas. Estes gráficos proporcionam uma identificação visual da concentração dos valores das métricas nas amostras, além de apresentar claramente os valores extremos (*outliers*).

3.3.2 Métricas de Dispersão de Mudanças

As métricas de dispersão de mudanças permitem avaliar a abrangência dos efeitos de uma mudança no código-fonte, permitindo identificar a ocorrência dos padrões (*code smells*) conhecidos como *Shotgun Surgery* e *Scattered Functionality*.

Para os propósitos desta análise, consideramos que o padrão *Shotgun Surgery* é observado se a implementação de uma mudança exigir a alteração de um grande número de classes, ao invés de uma única classe. Isto implica que as classes do sistema não foram organizadas de modo a conter um eixo de mudança cada, sendo necessário alterar um grande número de classes para incorporar uma mudança ao sistema. Embora o número ideal de uma classe para cada alteração não possa ser atingido em todos os casos, o ideal é que o número médio de classes alteradas em cada mudança seja pequeno [171].

De forma análoga, observa-se o padrão *Scattered Functionality* se uma mudança exigir a alteração de classes de um grande número de pacotes distintos, em vez de um único pacote que incorpore a funcionalidade relacionada à alteração necessária [171]. Assim como para as classes, o ideal é que o número médio de pacotes modificados para adicionar uma nova funcionalidade ou corrigir uma falha seja pequeno.

A identificação dos dois padrões é realizada a partir do conjunto de alterações feitas no sistema. Para medir até que ponto uma versão do sistema foi afetada, determina-se quais alterações foram realizadas no contexto desta versão. Em seguida, o número de classes e pacotes envolvidos na implementação de cada alteração são contados. A identificação de alterações não é uma tarefa simples, pois o motivo que

levou a elas pode não ter sido documentado: algumas alterações podem ter sido registradas em um *issue tracking system*, em resposta a demandas e para corrigir erros encontrados pelos usuários, mas outras alterações podem ter sido realizadas em resposta a erros encontrados pelos desenvolvedores, à adição de novas funcionalidades, à refatoração do código-fonte, e assim por diante.

Para identificar as alterações, coletamos informações do sistema de controle de versão e tratamos cada *commit* como uma alteração. No entanto, os *commits* em um sistema de controle de versão estão relacionadas à implementação de uma alteração, não às suas causas, justificativas ou razões. Devido a essa relação, os *commits* podem não estar perfeitamente alinhados com as alterações nos seguintes cenários:

- um *commit* pode incorporar alterações no código-fonte relacionadas a mais de uma alteração nas funcionalidades do sistema; e
- muitos *commits* podem ser necessários para salvar as versões editadas de todas as classes envolvidas em uma única alteração no sistema.

No entanto, os *commits* são a fonte mais confiável de informações sobre alterações disponível em documentos públicos que descrevem o sistema. Portanto, para medir a dispersão das alterações realizadas no sistema, contamos o número de classes e pacotes envolvidos em cada *commit*. Um grande número de *commits* envolvendo um grande número de classes indica a ocorrência do padrão *Shotgun Surgery*. Um grande número de *commits* envolvendo classes residentes em um grande número de pacotes distintos indica a ocorrência do padrão *Scattered Functionality*.

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*) de classes e pacotes que foram afetados por um *commit*. Os *outliers* indicam as ocorrências de pico de número de classes ou pacotes em um *commit*. Na visualização sem *outliers* verificamos a distribuição das ocorrências de *commits*, além de marcar os valores médio e mediano.

3.3.3 Métricas de Coesão e Acoplamento

Para analisar a evolução do acoplamento nos sistemas selecionados, consideramos uma extensão da métrica “*Coupling between Objects*” (CBO) para pacotes. A métrica CBO foi apresentada por Chidamber e Kemerer [73] e calcula o acoplamento de uma classe A contando o número de classes das quais A depende para implementar suas funcionalidades. Sua extensão para medir o acoplamento de pacotes considera as dependências de uso entre pacotes, que derivamos do conceito de dependências de uso entre classes: um pacote A depende de um pacote B se pelo menos uma classe do pacote A depender de pelo menos uma classe do pacote B para desempenhar suas funções. O CBO para um pacote A é calculado como o número de pacotes que contém classes das quais as classes do pacote A dependem. O senso comum sobre *design* sugere que o CBO deve ser minimizado, ou seja, as classes e os pacotes devem depender do menor número possível de outras classes ou pacotes.

A métrica “*Afferent Coupling*” (AFF) é uma medida de acoplamento calculada contando o número de classes de fora de um pacote A que dependem de pelo menos uma classe do pacote A. Um pacote com AFF alto tem forte responsabilidade como parte de um sistema, pois muitas classes dependem dele fornecer as funcionalidades a ele associadas.

A métrica “*Efferent Coupling*” (EFF) é uma medida de acoplamento calculada contando o número de classes de fora de um pacote A do qual as classes residentes no pacote A dependem para implementar suas funcionalidades. Quanto menor o EFF para um pacote, mais independente ele é do restante do sistema. Portanto, mais fácil será testar e reutilizar este pacote em outros sistemas.

Para a análise de coesão, a métrica selecionada é uma adaptação da métrica “*Lack of Cohesion of Methods*” (LCOM) para determinar a coesão de um pacote. A métrica LCOM foi proposta por Henderson-Sellers [137] e calcula a falta de coesão em uma classe de acordo com o uso comum de atributos por seus métodos. Quanto maior o número de método acessando um maior número de atributos, menor será o valor retornado pela métrica. A métrica é definida no intervalo $[0, 1]$ e valores próximos a zero indicam maior coesão, ou seja, grande sinergia entre os métodos e atributos que compõem a classe.

Adaptamos a métrica para calcular a coesão de um pacote: em vez de métodos que fazem referência a atributos, consideramos o número de classes de um pacote que dependem de outras classes do mesmo pacote. A falta de coesão para um pacote é observada quando a maioria das suas classes não depende de outras classes do próprio pacote, resultando em um valor alto de LCOM.

As métricas de coesão e acoplamento foram selecionadas para avaliar o nível de relacionamento entre as entidades que compõem o software. Entretanto, estas métricas são concorrentes. Se um arquiteto pretende maximizar a coesão do software, pode-se considerar colocar as classes em pacotes separados, para que o objetivo de cada pacote se torne claro e estritamente relacionado ao objetivo da classe.

Por outro lado, se o arquiteto pretende minimizar o acoplamento entre os pacotes, ele pode considerar colocar todas as classes em um único pacote para que nenhuma dependência entre pacotes seja possível. Nenhuma dessas soluções parece razoável em um cenário de desenvolvimento de software de larga escala, no qual os desenvolvedores buscam maneiras de agrupar classes relacionadas em pacotes para facilitar a compreensão do sistema e restringir diferentes tipos de alterações a diferentes partes do código-fonte. Portanto, torna-se necessário um equilíbrio entre acoplamento e coesão.

A análise por meio de gráficos *boxplot*, mais uma vez, proporciona a identificação visual da concentração dos valores destas métricas nas amostras, além de destacar as ocorrências de valores extremos. Este valores extremos indicam os picos de cada métrica em cada uma das versões dos softwares. Adicionalmente, podemos visualizar a variação dos valores das métricas para cada versão, além dos valores médio e mediano das métricas entre os softwares e suas versões.

Buscando verificar a semelhança dos valores observados para estas métricas, foi utilizada a análise dos coeficientes de correlação de Spearman entre cada par de métricas. Com esta análise podemos inferir 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 observados para estas métricas.

3.4 A Análise do JHotDraw

As versões do JHotDraw analisadas nesta pesquisa foram obtidas do repositório SourceForge do projeto⁵. Foram encontradas 16 versões, lançadas no período entre 2001 e 2011. A versão 5 perdurou de 2001 a 2004 e quatro versões menores foram disponibilizadas. A versão 6 não teve versões menores encontradas e foi liberada em 2004, na mesma data da versão 5.4.2. Inicialmente, fizemos uma suposição de que estas duas versões fossem iguais, mas a versão 6 apresenta os nomes dos pacotes estruturados de forma totalmente diferente. A versão 7 teve onze versões menores, lançadas no período entre os anos de 2006 e 2011.

A Figura 3.1 apresenta a linha do tempo destas versões. Os arquivos das versões são armazenados no repositório SourceForge. Entretanto, houve grande dificuldade para a aquisição, por meio de download, das versões. Os arquivos do repositório não estavam organizados, dificultando a catalogação. Nas primeiras versões do software não era utilizado o Maven ou qualquer outra ferramenta de construção do sistema. Somente na versão 7 foi encontrado o arquivo de configuração (*pom.xml*). Mesmo assim, em muitas versões, apesar de encontrado o código-fonte da versão entre os arquivos do projeto, não foi possível ter total garantia de que este código era utilizado para compilar e montar os arquivos de distribuição da versão. Os arquivos com explicações sobre o projeto também estavam desatualizados, mostrando códigos de diferentes versões.

O JHotDraw cresceu constantemente ao longo de seus, aproximadamente, 11 anos de existência. A primeira versão analisada do JHotDraw (v5.2.0) tinha 191 classes distribuídas por 11 pacotes, enquanto a última versão (v7.6.0) alcançou 1195 classes em 67 pacotes, representando um aumento de mais de 6 vezes o tamanho original do software. A Tabela 3.2 mostra os valores das métricas de tamanho e complexidade para as 16 versões do JHotDraw. A coluna “Pacotes” mostra o número de pacotes em uma

⁵<https://sourceforge.net/projects/jhotdraw/>

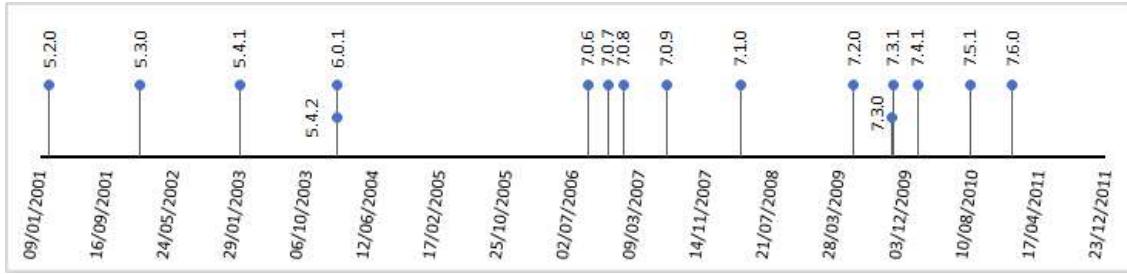


Figura 3.1: Linha do tempo da distribuição das versões do JHotDraw.

versão. A coluna “Classes” mostra o número de classes em uma versão. A coluna “Atributos” mostra o número de atributos distribuídos pelas classes da versão. A coluna “Métodos” mostra o número de métodos distribuídos pelas classes para a versão, enquanto a coluna “Mét. Públicos” mostra quantos desses métodos são públicos. Por fim, a coluna “Dependências” mostra o número de dependências em uma versão..

Tabela 3.2: Tamanho e complexidade das versões do JHotDraw.

Versão	Pacotes	Classes	Atributos	Métodos	Mét. Públicos	Dependências
5.2.0	11	191	218	1.548	1.230	997
5.3.0	11	273	111	2.361	1.834	1.548
5.4.1	17	391	544	3.346	2.560	2.086
5.4.2	17	398	560	3.432	2.593	2.175
6.0.1	17	398	560	3.432	2.593	2.175
7.0.6	16	320	661	2.760	2.169	1.500
7.0.7	16	333	694	2.812	2.200	1.574
7.0.8	16	379	765	3.288	2.600	1.763
7.0.9	17	431	883	3.597	2.837	2.082
7.1.0	47	1.149	2.162	10.026	6.226	4.051
7.2.0	20	391	376	3.729	2.985	2.007
7.3.0	56	1.231	1.922	11.048	6.856	4.533
7.3.1	56	1.233	1.923	11.078	6.875	4.543
7.4.1	65	1.186	1.866	10.716	6.585	4.490
7.5.1	68	1.193	2.029	10.855	6.663	4.566
7.6.0	67	1.195	1.990	10.951	6.748	4.533

Os dados da Tabela 3.2 mostram que o JHotDraw tem crescido constantemente desde suas versões iniciais em termos de número de pacotes e classes. O número médio de classes por pacote é 21,47 (mediana = 22,0, desvio padrão = 2,74), mas esse número não ficou estável ao longo dos anos, variando de 17,36 classes por pacote (versão 5.2.0) até 25,35 classes por pacote (versão 7.0.9). Lanza e Marinescu [172] estudaram 45 programas escritos na linguagem Java e descobriram que o número médio de classes por pacote é 17, variando de um mínimo de 6 a um máximo de 26 em seu conjunto de dados. Assim, as versões do JHotDraw ficaram acima da média do número de classes por pacote, mas dentro do limite máximo. Os maiores aumentos no número de classes entre versões consecutivas foram observados da versão 5.2.0 para 5.3.0 (42,93%) e da 7.0.7

para 7.0.8 (13,81%). As maiores reduções foram encontradas da versão 7.1.0 para 7.2.0 (20,03%) e da versão 7.3.1 para a 7.4.1 (17,13%).

A distribuição de dados ou funcionalidades entre classes, por outro lado, é estável em diferentes versões. O número médio de atributos por classe é 1,55 (mediana = 1,57, desvio padrão = 0,45, mínimo = 0,41 e máximo = 2,08), o número médio de métodos por classe é 8,76 (mediana = 8,66, desvio padrão = 0,35, mínimo = 8,10 e máximo = 9,54) e o número médio de métodos públicos por classe é 6,28 (mediana = 6,52, desvio padrão = 0,64, mínimo = 5,42 e máximo = 7,63). Uma classe depende, em média, de 4,59 outras classes (mediana = 4,71, desvio padrão = 0,76, mínimo = 3,53 e máximo = 5,67) e essa média diminuiu desde a primeira versão, mesmo que seu número absoluto tenha aumentado durante a evolução das versões.

Considerando as versões que foram liberadas durante a evolução do JHotDraw e buscando concentrar as análises nas versões que tiveram variação estrutural, foram realizadas análises das diferenças das versões. Desta forma, versões com o mesmo número de pacotes e classes foram descartadas, mantendo somente a versão mais nova. Além disso, as versões que foram citadas em outras publicações (5.1, 5.2, 5.3, 5.4.1, 6.0, 6.1, 6.2, 7.0.6 e 7.1) foram mantidas, uma vez que outros pesquisadores já as analisaram. A versão 7.2.0 foi desconsiderada, uma vez que o código-fonte da versão disponibilizado no repositório não estava íntegro e correspondendo com a versão intitulada. Isto foi constatado devido à grande variação entre a versão 7.2.0 e as versões anterior e posterior. Na Tabela 3.2 foram hachuradas as versões desconsideradas das análises (5.4.2, 7.2.0) conforme os critérios mencionados.

As versões analisadas são apresentadas nos gráficos de *boxplot* para cada versão do JHotDraw: a Figura 3.3 mostra a distribuição do número de classes por pacote, a Figura 3.4 mostra o número de atributos por classe, a Figura 3.5 mostra o número de métodos por classe e a Figura 3.6 mostra o número de métodos públicos por classe. *Outliers* são apresentados como pontos acima dos limites superiores ou abaixo dos limites inferiores.

Com a evolução das versões, novas funcionalidades foram acrescentadas e, consequentemente, aumentaram o número de pacotes e de relações de dependências

entre as classes. A Figura 3.2 apresenta um grafo para cada uma das versões em análise, sendo os pacotes representados pelos nós e as relações de dependência representadas pelas arestas destes grafos. Observando a evolução das versões representadas pelos grafos, percebemos que o número de dependências cresce até a versão 6.0.1, quando mostra uma grande redução no início da versão 7.0.6. Até a liberação da última versão (7.6.0), observa-se um aumento elevado do número de pacotes e de dependências.

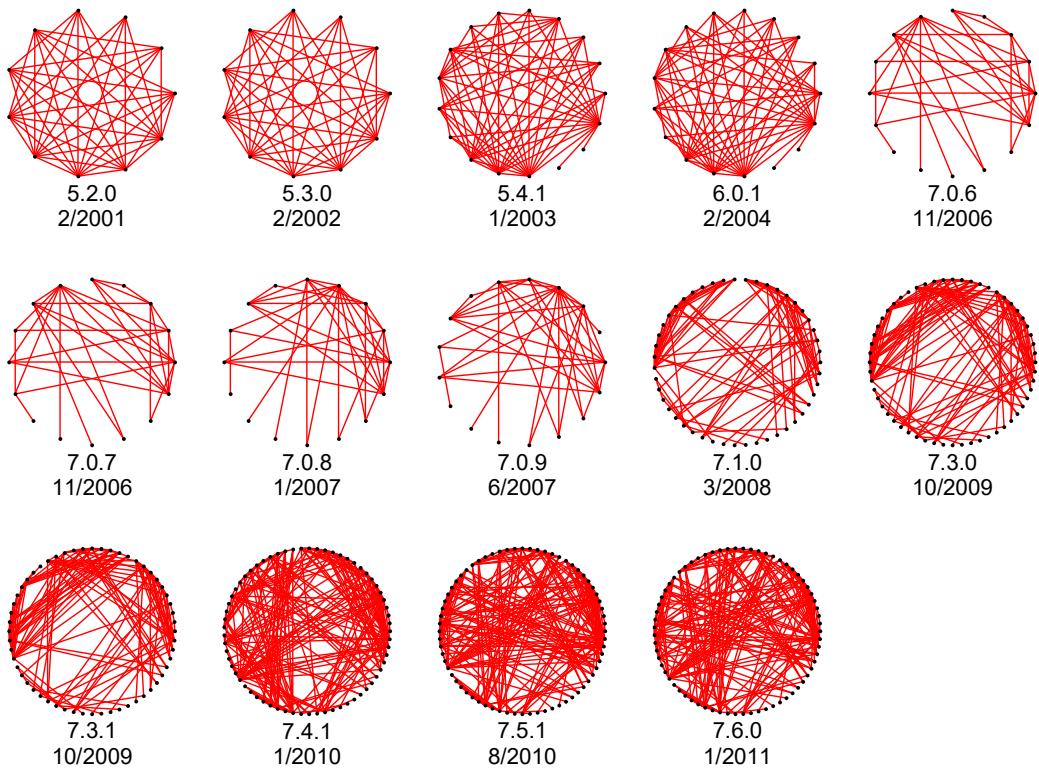


Figura 3.2: Evolução da arquitetura do JHotDraw, com pontos representando os pacotes e linhas representando relações de dependência entre os pacotes.

A Tabela 3.3 apresenta coeficientes de correlação de Spearman entre as métricas de tamanho e complexidade. Uma medida de correlação não-paramétrica é usada porque o número de classes nas versões do JHotDraw, sendo definido como um número inteiro no intervalo $[0, +\infty)$, não pode se parecer com uma distribuição normal. A análise de correlação mostra que as métricas apresentaram correlação forte e positiva, no intervalo

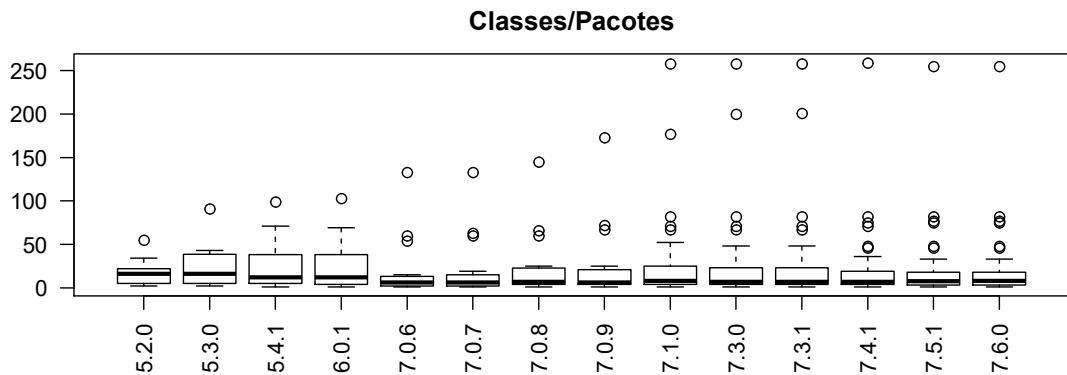


Figura 3.3: *Boxplots* para a distribuição do número de classes por pacote no JHotDraw.

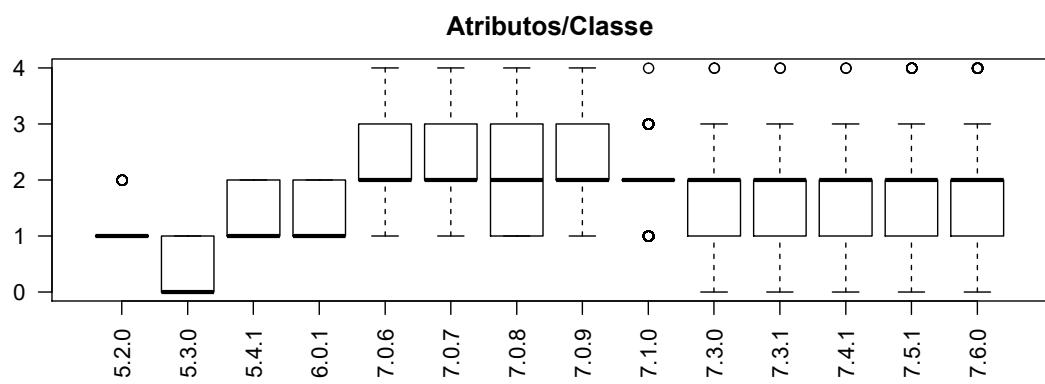


Figura 3.4: *Boxplots* para a distribuição do número de atributos por classe no JHotDraw.

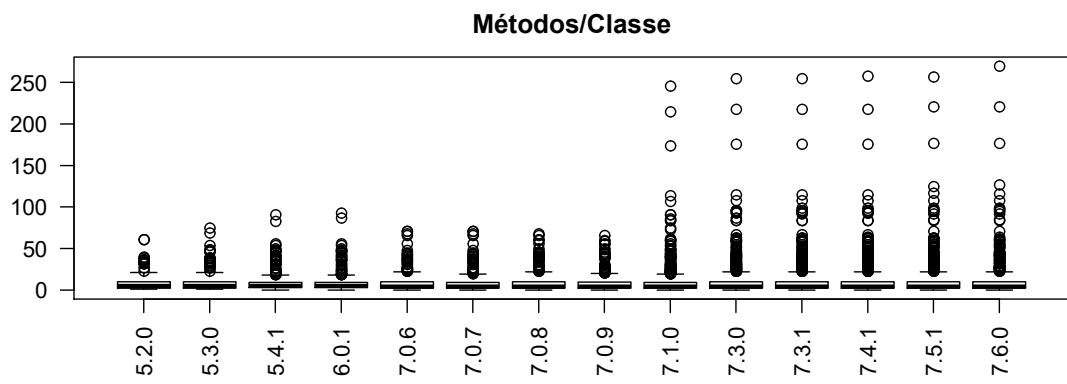


Figura 3.5: *Boxplots* para a distribuição do número de métodos por classe no JHotDraw.

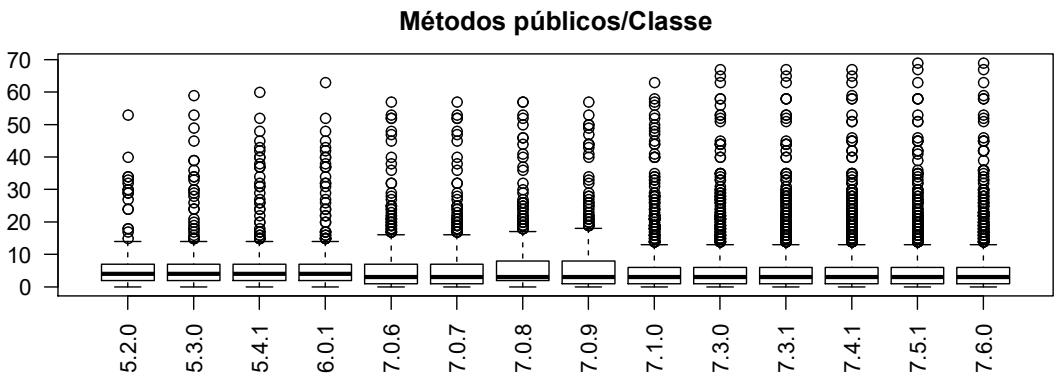


Figura 3.6: *Boxplots* para a distribuição do número de métodos públicos por classe no JHotDraw.

de 0,82 a 1,00. Por este motivo, é possível perceber que todas as métricas são igualmente representativas do tamanho e complexidade do sistema.

Tabela 3.3: Correlação entre as medidas de tamanho e complexidade do JHotDraw.

	Pacotes	Classes	Atributos	Métodos	Métodos Públicos	Dependências
Pacotes	1,00	0,92	0,83	0,92	0,89	0,95
Classes		1,00	0,82	1,00	0,99	0,95
Atributos			1,00	0,82	0,85	0,80
Métodos				1,00	0,99	0,95
Métodos Públicos					1,00	0,93
Dependências						1,00

Analizando a equipe que foi responsável pelo desenvolvimento, foi possível perceber que longo da evolução do sistema, a equipe teve no máximo 6 desenvolvedores. O aumento mais expressivo ocorreu entre 2002 e 2004, quando atingiu o pico. Na maior parte do restante do período, contou com apenas um desenvolvedor, conforme apresentado na Tabela 3.4.

Tabela 3.4: Tamanho da equipe, entrada e saída de desenvolvedores e número de *commits* por ano para o projeto JHotDraw.

Ano	Equipe	Entrada	Saída	Commits
2000	1	1	0	13
2001	1	1	1	12
2002	4	3	0	30
2003	6	3	1	143
2004	5	1	2	22
2005	1	0	4	3
2006	2	1	0	18
2007	1	0	1	78
2008	2	1	0	68
2009	1	0	1	117
2010	2	1	0	103
2011	1	0	1	38
2012	1	0	0	12
2013	1	0	0	5
2014	1	0	0	2
2015	1	0	0	300
2016	1	0	0	128
2017	1	0	0	126

O repositório do sistema de controle de versão⁶ mantém as informações sobre as mudanças no código-fonte (*commits*) em *logs*. Destes *logs*, foram coletadas informações

⁶<https://svn.code.sf.net/p/jhotdraw/svn>

acerca do número de *commits* e do número de desenvolvedores que fizeram estes *commits*. Os *commits* contados continham adições, exclusões e modificações do código-fonte extraído do repositório. Foi possível perceber que apesar da última versão ter sido lançada no ano de 2011, o código-fonte continuou sendo atualizado e apresentou um número recorde de *commits* em 2015, conforme pode ser observado na Tabela 3.4. Entretanto, como a análise do JHotDraw está sendo realizada com base em versões disponíveis para *download*, os *commits* posteriores a 2011 não foram considerados.

A Tabela 3.5 fornece informações sobre o número de classes e pacotes envolvidos nos *commits* registrados no sistema de controle de versão do JHotDraw ao longo do tempo. A coluna “*Commits*” mostra o número de *commits* executados antes de liberar cada versão do sistema. Um total de 608 *commits* foram realizados entre 2000 e 2011. Cada valor da coluna “*Commit*” conta o número de *commits* registrados até que outra versão fosse liberada. Como pode ser visto, os picos de número de *commits* ocorreram nas versões 5.4.1 e 7.3.0.

Tabela 3.5: Número de *commits* e sua distribuição pelas classes e pacotes para o JHotDraw

Versão	Commits	Commit de uma classe		Commit de um pacote		Classes por commit			Pacotes por commit		
		Núm.	%	Núm.	%	Med	μ	σ	Med	μ	σ
5.2.0	13	0	-	0	-	2	23,69	78,21	2	3,54	5,55
5.3.0	13	0	-	0	-	44	89,38	100,88	12	12,46	9,06
5.4.1	100	0	-	0	-	4	21,36	56,44	4	5,96	6,08
6.0.1	74	0	-	0	-	2	24,97	87,12	2	5,05	8,39
7.0.6	23	11	47,83	12	52,17	2	3,78	4,17	1	2,17	1,59
7.0.7	3	0	-	0	-	314	280	74,10	22	20,33	6,66
7.0.8	20	5	25,00	7	35,00	4	17,45	28,05	2	3,2	4,55
7.0.9	48	11	22,92	14	29,17	8	19,4	26,69	3	4,52	4,56
7.1.0	33	7	21,21	12	36,36	11	27,79	65,27	2	5,3	6,07
7.3.0	150	31	20,67	40	26,67	4	16,21	48,35	2	4,20	5,38
7.3.1	7	1	14,29	1	14,29	5	6,86	5,58	2	3,71	2,81
7.4.1	35	5	14,29	8	22,86	9	40,11	89,13	3	8,14	11,13
7.5.1	48	7	14,58	15	31,25	5	20,25	54,37	2	5,52	8,69
7.6.0	41	10	24,39	12	29,27	4	34,61	91,91	3	8,44	13,07

A coluna “*Commit de uma classe*” da Tabela 3.5 conta o número de *commits* que enviam alterações no código-fonte de uma única classe, enquanto a coluna “*Commit de uma classe (%)*” mostra esse número como um percentil do número total de *commits*

enviados em relação a versão. Em média, 14,47% dos *commits* envolveram uma única classe ao longo da evolução da JHotDraw. O percentual de *commits* de uma classe nas primeiras versões era zero até que a versão 7.0.6 teve quase metade dos *commits* com uma classe. As demais versões ficaram abaixo de 25% indicando que a maioria das alterações feitas no sistema envolvia a edição de mais de uma classe. Essa é uma indicação importante de que a dispersão das mudanças não estava sob controle.

A coluna “Classes por *commit*” mostra a mediana entre parênteses, o número médio de classes envolvidas em *commits* e seu desvio padrão (após o sinal \pm). Os elevados valores de desvio padrão calculados ressaltam a ocorrência de poucos *commits* envolvendo um número elevado de classes. A distribuição desta medida nas versões do JHotDraw é apresentada nos gráficos de *boxplot* na Figura 3.7. Esses gráficos são apresentados com *outliers* e sem *outliers*. Somente ocorreram *outliers* maiores que o terceiro quartil nos *boxplot* das versões do JHotDraw. Alguns *commits* afetam um grande número de classes, como, por exemplo, um único *commit* envolvendo mais de 500 classes na versão 7.3.0. Além disto, verificamos que quase todas as versões apresentaram *commits* de aproximadamente metade das classes da versão. Os *outliers* reduzem o espaço vertical disponível para mostrar a parte central das distribuições (entre o primeiro e o terceiro quartis). Assim, apresentamos *boxplots* com *outliers* para mostrar esses valores extremos e sem *outliers* para destacar o IQR curto apresentado pelas distribuições.

O número de *commits* envolvendo apenas uma classe se apresentou de forma variada. Nas primeiras quatro versões, todos os *commits* foram de mais de uma classe. Nas versões seguintes, o pico ocorreu na versão 7.0.6, na qual 47,83% dos *commits* foram de uma classe. O número médio de classes por *commit* também variou de forma expressiva, observando-se o menor valor em 3,78 (versão 7.0.6) e o maior valor em 280 (versão 7.0.7). Entretanto, o alto valor médio deve-se a uma versão que somente teve três *commits* com alterações em um elevado número de classes, indicando uma grande refatoração. Desconsiderando o pico, a média de classes por *commit* foi de 26,60. Também importante, o desvio padrão para os pontos de dados dessa tendência crescente

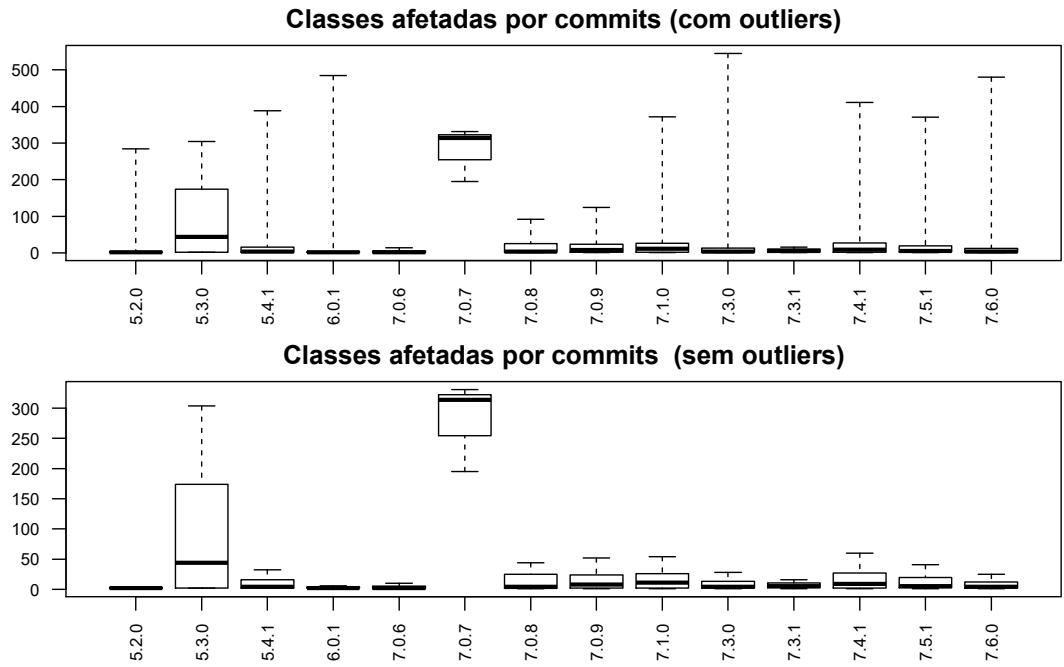


Figura 3.7: Boxplots para a distribuição de número de classes envolvidas em *commits* por versão no JHotDraw.

também aumentou ao longo do tempo, denotando que algumas mudanças podem ter envolvido um grande número de classes (muito maior do que a média já inflada), conforme ilustrado nos gráficos de boxplot com *outliers* na Figura 3.7. Em suma, os dados da Tabela 3.5 indicam a presença do efeito da *Shotgun Surgery*: o número de *commits* envolvendo uma única classe é pequeno e cada alteração afeta muitas classes, variando de 3,78 a 280 classes por *commit*.

Em relação ao número de pacotes envolvidos nos *commits*, a coluna “*Commits* de um pacote”, na Tabela 3.5, conta o número de *commits* envolvendo classes de um único pacote, enquanto a coluna “*Commits* de um pacote (%)” mostra esse número como um percentil do número total de *commits* para a versão. Assim como o *commit* de uma classe, as primeiras quatro versões não tiveram *commits* de um pacote registrados. Em média, somente 19,79% dos *commits* enviados ao sistema de controle de versão envolviam um único pacote. Analisando as últimas 5 versões, a média diminui para, aproximadamente,

17% dos *commits* de uma classe. Esta é uma segunda indicação de que a dispersão das mudanças não estava sob controle: os *commits* envolviam classes de diversos pacotes, aumentando assim os casos de disseminação de alterações no nível do pacote.

A coluna “Pacotes por *commit*” mostra a mediana entre parênteses, o número médio de pacotes envolvidos em *commits* e seu desvio padrão (após o sinal \pm). A distribuição dessa medida entre as versões analisadas é apresentada nos gráficos de *boxplot* da Figura 3.8. A versão 7.0.7 apresentou características muito diferentes das demais, com uma média de 20,33 pacotes por *commit*, mas esta versão teve somente três *commits*. Observando as medianas na Tabela 3.11, todas as versões analisadas tiveram valor menor que 4 pacotes por *commit*, retirando os picos das versão 5.3.0 e 7.0.7.

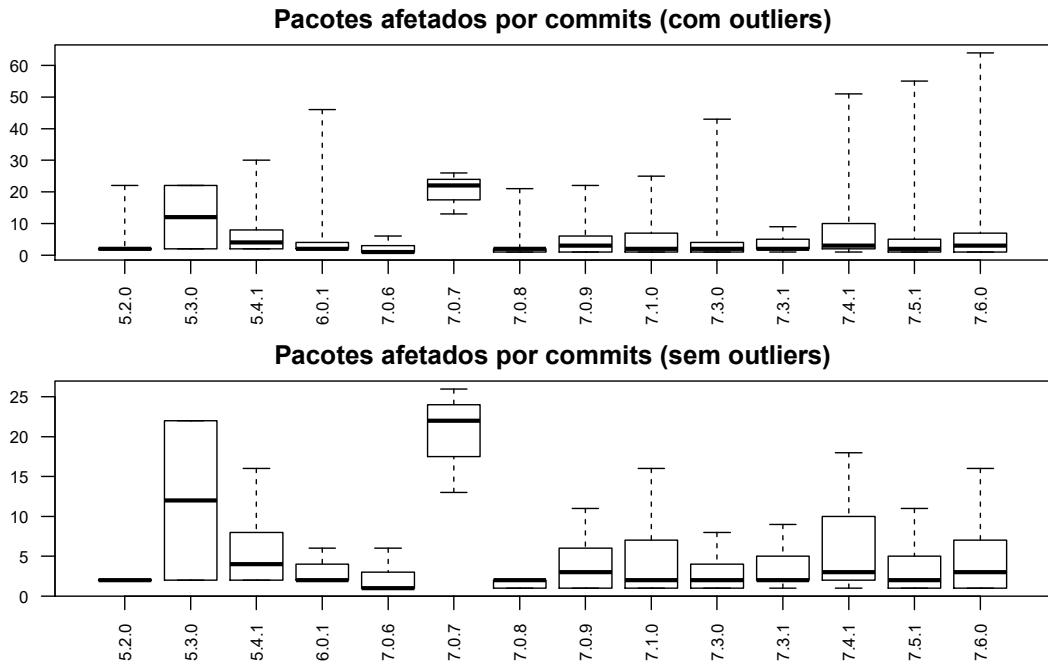


Figura 3.8: *Boxplots* para a distribuição de número de pacotes envolvidos em *commits* por versão para o JHotDraw.

Passando para as métricas de coesão e acoplamento, a Tabela 3.6 apresenta os valores médios dessas métricas nas versões. A distribuição completa dessas métricas é apresentada nos diagramas de *boxplot* apresentados nas Figuras 3.9, 3.10, 3.11 e 3.12.

Tabela 3.6: Métricas de acoplamento e coesão para as versões do JHotDraw.

Versão	CBO ↓	AFF ↓	EFF ↓	LCOM ↓
5.2.0	4.00	25.64	24.18	0.83
5.3.0	4.18	39.55	31.36	0.84
5.4.1	4.18	34.71	27.71	0.83
6.0.1	4.18	35.41	28.12	0.82
7.0.6	2.19	19.38	9.81	0.78
7.0.7	2.19	20.69	10.19	0.79
7.0.8	2.31	22.19	11.25	0.82
7.0.9	2.53	24.29	12.18	0.81
7.1.0	2.28	15.96	8.96	0.78
7.3.0	2.62	15.77	8.82	0.83
7.3.1	2.64	15.82	8.86	0.83
7.4.1	3.66	19.18	10.89	0.83
7.5.1	3.68	18.65	10.74	0.81
7.6.0	3.55	18.45	10.63	0.81

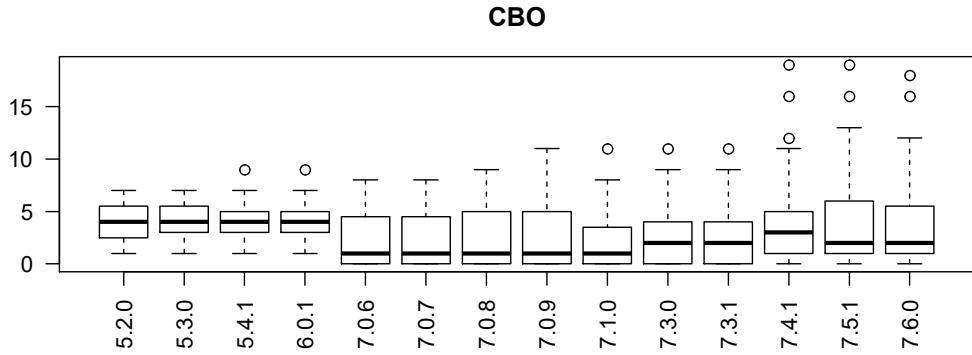


Figura 3.9: Boxplots para CBO por versão do JHotDraw.

A coluna CBO na Tabela 3.6 apresenta a CBO média para os pacotes que compreendem cada versão do sistema, enquanto a coluna AFF apresenta o valor médio para o acoplamento aferente e a coluna EFF apresenta o valor médio para o acoplamento eferente. Esses valores mostram uma tendência decrescente ao longo da evolução do JHotDraw, até a última versão (7.6.0). Também apresentam correlação forte e positiva, mostrando que, pelo menos para o JHotDraw, diferentes medidas de acoplamento são consistentes entre si, variando entre 0,50 a 0,93, conforme a Tabela 3.7.

A coluna LCOM na Tabela 3.6 apresenta o valor médio de LCOM para pacotes que compreendem cada versão do sistema. O LCOM médio se apresentou estável nas

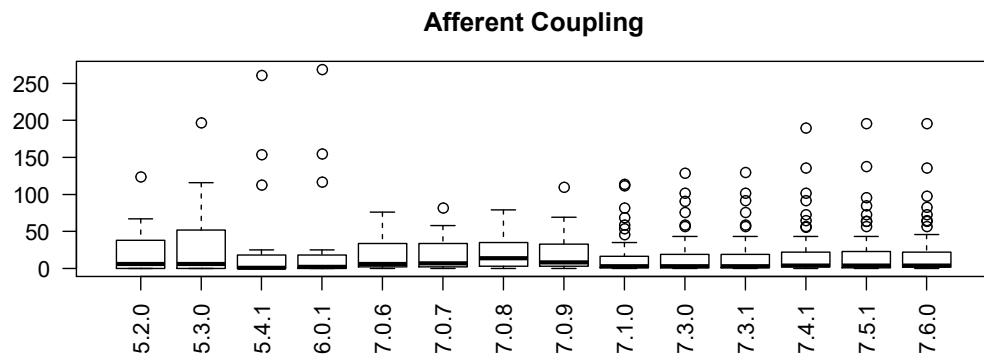


Figura 3.10: *Boxplots* para *Afferent Coupling* por versão do JHotDraw.

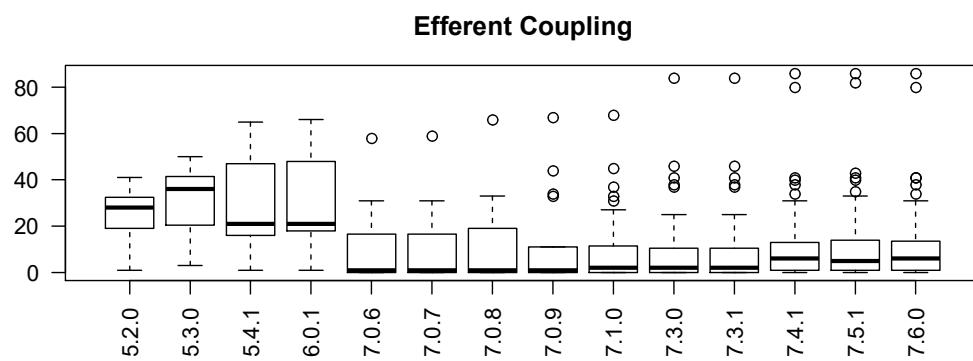


Figura 3.11: *Boxplots* para *Efferent Coupling* por versão do JHotDraw.

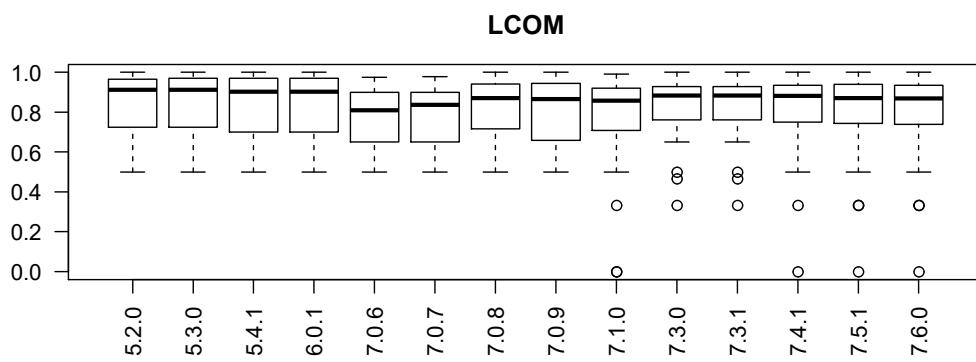


Figura 3.12: *Boxplots* para LCOM por versão do JHotDraw.

versões e com correlação forte com CBO, conforme a Tabela 3.7. Apesar de coesão e acoplamento serem teoricamente competitivos, a métrica LCOM mede a ausência de coesão. Desta forma, pode-se esperar uma correlação positiva e forte entre elas. Os dados indicam que os pacotes mostraram uma coesão estável ao longo do tempo. No entanto, é interessante ver que a coesão permaneceu quase inalterada, mesmo em períodos de variação de acoplamento.

Tabela 3.7: Análise de correlação (Spearman) entre as métricas do JHotDraw

	CBO	AFF	EFF	LCOM
CBO	1,00	0,50	0,70	0,70
AFF		1,00	0,93	0,27
EFF			1,00	0,40
LCOM				1,00

Conforme mencionado nas análises, a evolução da versões do JHotDraw apresentou indícios de *Shotgun Surgery* e *Scattered Functionality*. Estes indícios indicam um descontrole da evolução arquitetural do sistema devido à dispersão das funcionalidades nos pacotes e classes do sistema. As versões 6.1 e 6.2, que foram utilizadas como referências em publicações anteriores, não foram encontradas para *download* na Internet e, portanto, não foram analisadas.

Por fim, apesar de diversas publicações terem classificado uma versão específica do JHotDraw como um bom exemplo de design, estas qualidades não se apresentam claramente nos valores das métricas quando analisamos a evolução do projeto.

3.5 A Análise do JEdit

As versões analisadas nesta pesquisa foram obtidas do repositório do projeto⁷. No repositório foram encontradas 40 versões, lançadas no período entre 2000 e 2018. Dentre estas versões foram liberadas quatro versões principais. A versão 2 teve 13 versões menores disponibilizadas e tudo isto ocorreu em um ano (2000). A versão 3 teve

⁷<https://sourceforge.net/projects/jedit>

7 versões liberadas no intervalo dos anos 2000 e 2001. A versão 4 teve 14 versões menores liberadas em um período de 10 anos (2002 até 2012). A última versão (5) iniciou em 2012 e teve sua sexta versão liberada em 2018.

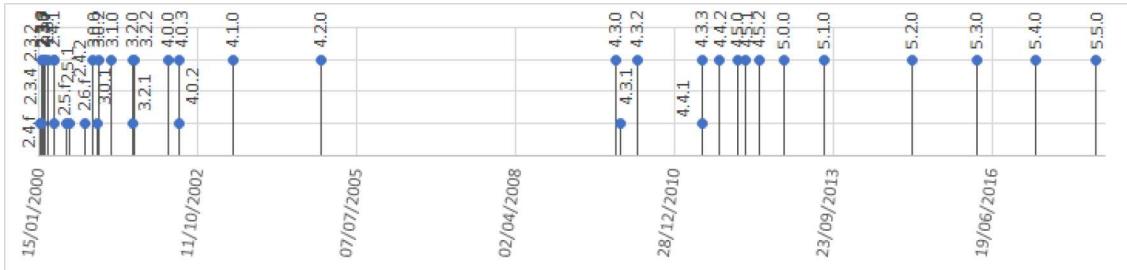


Figura 3.13: Distribuição das versões do JEdit pelo tempo.

A Figura 3.13 apresenta a linha do tempo de liberação destas versões. Muitas versões foram lançadas no início do período, indicando a possibilidade de pequenas melhorias ou correções. O intervalo de tempo entre as versões 4.2.0 e 4.3.0 foi em torno de 5 anos. Para as últimas seis versões, o intervalo de tempo seguiu uma regularidade maior. Mesmo possuindo um elevado número de versões, a estrutura dos arquivos encontrados no repositório estava organizada, além de disponibilizar o código-fonte e a versão compilada (JAR) para download. Desta forma, a identificação das versões foi mais confiável.

O JEdit cresceu constantemente ao longo de seus, aproximadamente, 20 anos de existência. A Tabela 3.8 mostra os valores das métricas de tamanho e complexidade para as 40 versões do JEdit. A coluna “Pacotes” mostra o número de pacotes em uma versão. A coluna “Classes” mostra o número de classes em uma versão. A coluna “Atributos” mostra o número de atributos distribuídos pelas classes da versão. A coluna “Métodos” mostra o número de métodos distribuídos pelas classes para a versão, enquanto a coluna “Mét. Públicos” mostra quantos desses métodos são públicos. Por fim, a coluna “Dependências” mostra o número de dependências em uma versão.

Tabela 3.8: Tamanho e Complexidade das Versões do JEdit.

Versão	Pacotes	Classes	Atributos	Métodos	Mét. Públicos	Dependências
2.3.2	2	23	39	91	64	52
2.3.3	2	23	39	91	64	52
2.3.4	2	23	39	91	64	52
2.3.5	2	23	39	90	64	52
2.3.6	2	23	39	90	64	52
2.3.7	2	23	39	90	64	52
2.3.f	2	23	39	90	64	52
2.4.1	2	23	39	90	64	52
2.4.2	2	23	39	90	64	52
2.4.f	2	23	39	90	64	52
2.5.1	2	29	51	131	77	64
2.5.f	2	29	51	130	77	64
2.6.f	2	28	50	129	77	64
3.0.0	14	415	654	3.406	2.002	1.952
3.0.1	14	417	658	3.436	2.023	1.962
3.0.2	14	417	658	3.436	2.023	1.962
3.1.0	14	440	700	3.555	2.094	2.035
3.2.0	16	508	828	4.098	2.368	2.420
3.2.1	16	508	828	4.100	2.370	2.422
3.2.2	16	508	825	4.102	2.371	2.425
4.0.0	19	606	1.003	4.823	2.821	2.899
4.0.2	19	606	1.003	4.825	2.821	2.899
4.0.3	19	606	1.003	4.825	2.822	2.899
4.1.0	20	644	477	5.014	2.990	3.197
4.2.0	23	805	1.425	6.110	3.630	4.151
4.3.0	29	1.132	2.263	7.990	4.936	5.891
4.3.1	29	1.132	2.263	7.992	4.938	5.892
4.3.2	29	1.132	2.263	7.998	4.939	5.888
4.3.3	29	1.132	2.263	7.999	4.940	5.888
4.4.1	29	1.193	2.417	8.186	5.081	6.159
4.4.2	29	1.194	2.420	8.193	5.084	6.167
4.5.0	30	1.216	2.440	8.271	5.098	6.299
4.5.1	30	1.218	2.444	8.294	5.114	6.305
4.5.2	30	1.220	2.448	8.304	5.117	6.315
5.0.0	34	1.267	2.555	8.638	5.279	6.497
5.1.0	35	1.266	2.541	8.686	5.298	6.519
5.2.0	35	1.277	2.569	8.771	5.360	6.586
5.3.0	35	1.289	2.601	8.843	5.395	6.710
5.4.0	35	1.355	2.776	9.206	5.618	7.015
5.5.0	35	1.366	2.804	9.277	5.668	7.074

Analisando a evolução do software, é possível perceber que o JEdit tem crescido constantemente desde suas versões iniciais em termos de número de pacotes e classes. O número médio de classes por pacote é 28,04 (mediana = 31,82, desvio padrão = 11,66), mas esse número não ficou estável ao longo dos anos, variando de 11,5 classes/pacote nas primeiras 10 versões até 41,17 na versão 4.4.2. A primeira versão analisada (2.3.2) tinha 23 classes localizadas em dois pacotes e a última versão analisada (5.5.0) alcançou 1.366 classes em 35 pacotes, representando um aumento de mais de 59 vezes em relação

ao tamanho original do software. Durante quase todas as versões menores da versão 2 ocorreu um crescimento pequeno. Somente nas três últimas versões menores houve aumento máximo de seis classes. Não somente o número de classes manteve esta estabilidade – o número de atributos, métodos, métodos públicos e dependências também se mantiveram estáveis.

Na evolução para o grupo de versões 3, o número de classes apresentou crescimento em, aproximadamente, 15 vezes, assim como as demais métricas de tamanho. O grupo das versões 4 também apresentou aumento, alcançando na última versão 4 (4.4.2) aproximadamente o dobro do número de classes da sua primeira versão (4.0.0). Por fim, o grupo de versões 5 agregou mais classes, mas nada na mesma magnitude dos incrementos das versões anteriores (menor do que 7% entre as versões).

A distribuição de dados ou funcionalidades entre classes, por outro lado, é estável entre diferentes versões. O número médio de atributos por classe é 1,78 (mediana = 1,7, desvio padrão = 0,24, mínimo = 0,74 e máximo = 2,05), o número médio de métodos por classe é 6,31 (mediana = 6,86, desvio padrão 1,66, mínimo = 3,91 e máximo = 8,24) e o número médio de métodos públicos por classe é 3,9 (mediana = 4,2, desvio padrão = 0,82, mínimo = 2,66 e máximo = 4,85). Uma classe depende, em média, de 4,11 outras classes (mediana = 4,78, desvio padrão = 1,32, mínimo = 2,21 e máximo = 5,21) e essa média aumentou desde a primeira versão.

Considerando as versões que foram liberadas durante a evolução do JEdit e buscando concentrar as análises nas versões que tiveram variação estrutural, foram realizadas análises das diferenças entre as versões. Desta forma, as versões com o mesmo número de pacotes e classes iguais foram desconsideradas, mantendo somente a versão mais nova. Entretanto, as versões que foram citadas pelas publicações analisadas (2.4.1, 3.1.0, 3.2.0) foram mantidas, uma vez que outros pesquisadores já as analisaram. Sendo assim, na Tabela 3.8, foram hachuradas as versões que foram desconsideradas das análises seguintes (2.3.2, 2.3.3, 2.3.4, 2.3.5, 2.3.6, 2.3.7, 2.3.f, 2.4.2, 2.5.1, 3.0.1, 3.2.1, 4.0.0, 4.0.2, 4.3.0, 4.3.1 e 4.3.2), conforme os critérios mencionados.

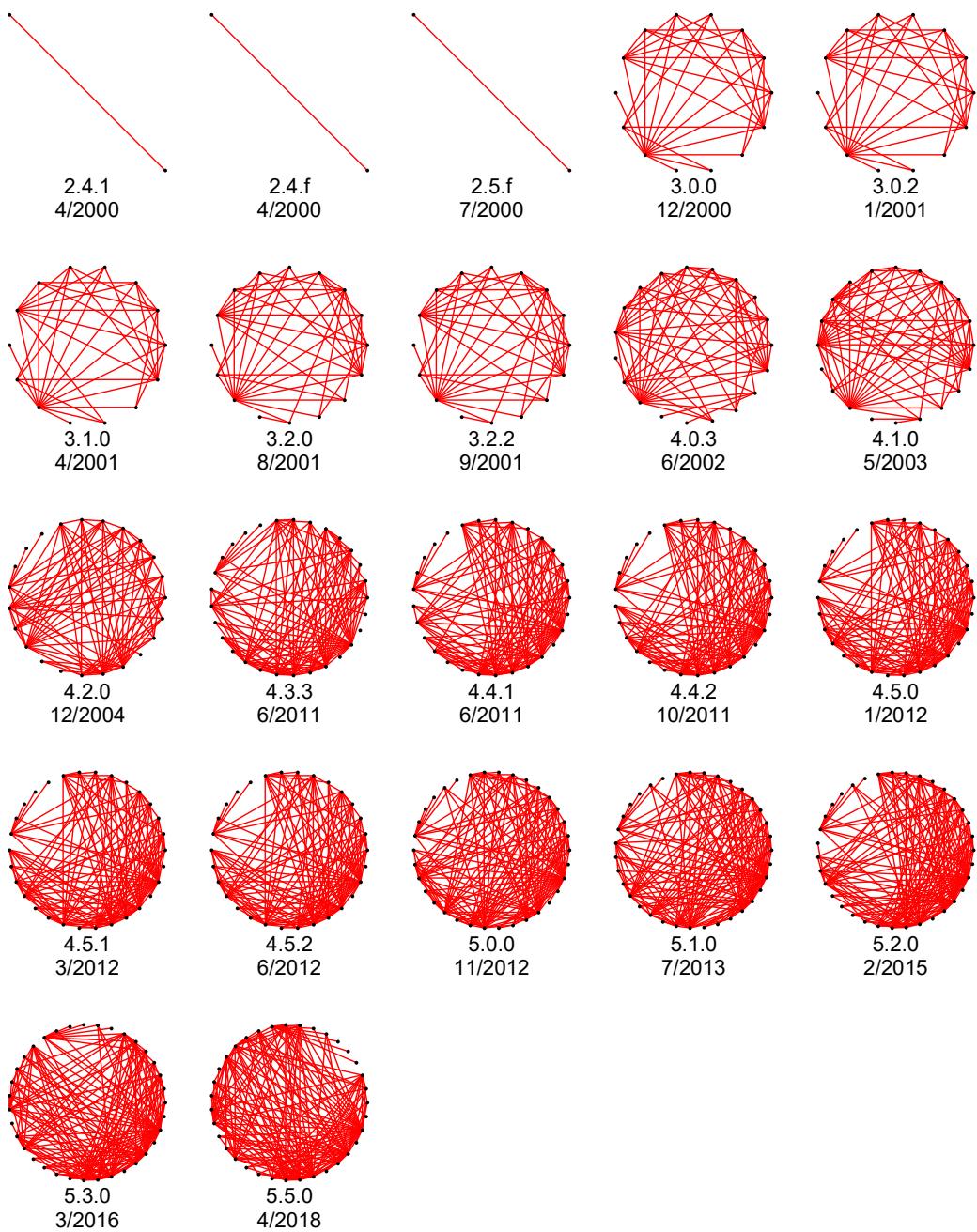


Figura 3.14: Evolução da arquitetura do JEdit.

As versões analisadas são apresentadas nos gráficos de *boxplot* para cada versão do JEdit: a Figura 3.15 mostra a distribuição do número de classes por pacote, a Figura

3.16 mostra o número de atributos por classe, a Figura 3.17 mostra o número de métodos por classe e a Figura 3.18 mostra o número de métodos públicos por classe. *Outliers* são apresentados como pontos acima dos limites superiores ou abaixo dos limites inferiores.

Com a evolução das versões, novas funcionalidades foram acrescentadas e, com isto, aumentaram o número de pacotes e as dependências entre as classes. A Figura 3.14 apresenta um grafo para cada uma das versões em análise, sendo os pacotes representados pelos nós e as relações de dependência representadas pelas arestas nestes grafos. Analisando esta evolução, é possível verificar uma mudança drástica entre as versões 2 e 3: a partir da versão 3 ocorreu um crescimento substancial no número de dependências, classes e pacotes.

A Tabela 3.9 apresenta coeficientes de correlação de Spearman entre as métricas de tamanho e complexidade. Uma medida de correlação não-paramétrica é usada porque o número de classes nas versões do JEdit, sendo definido como um número inteiro no intervalo $[0, +\infty)$, não pode se parecer com uma distribuição normal. A análise de correlação mostra que as métricas apresentaram correlação muito forte e positiva, no intervalo de 0,98 a 1,00. Por este motivo, é possível perceber que as métricas são igualmente representativas do tamanho e complexidade do sistema.

Tabela 3.9: Análise de correlação entre as medidas de tamanho/complexidade e o número de classes do JEdit

Métrica correlacionada	Pacotes	Classes	Atributos	Métodos	Métodos Públicos	Dependências
Pacotes	1,00	0,99	0,98	0,99	0,99	0,99
Classes		1,00	0,99	0,99	1,00	1,00
Atributos			1,00	0,98	0,99	0,99
Métodos				1,00	0,99	0,99
Métodos Públicos					1,00	1,00
Dependências						1,00

Analizando a equipe responsável pelo desenvolvimento, foi possível perceber que ela contou com no máximo 31 desenvolvedores. A primeira versão contou com 7 desenvolvedores, mas nas versões seguintes apresentou grande rotatividade. A partir de 2012, o número de desenvolvedores passou a diminuir a cada ano, apresentando somente

um desenvolvedor em 2020, conforme apresentado na Tabela 3.10. Considerando que a última versão em análise foi liberada em 2018, todos os *commit* registrados após a liberação desta versão foram descartados.

Tabela 3.10: Tamanho da equipe, entrada e saída de desenvolvedores e número de *commits* por ano para o projeto JEdit.

Ano	Equipe	Entrada	Saída	Commits
2000	7	7	0	100
2001	16	10	1	695
2002	26	15	5	1.315
2003	29	14	11	1.694
2004	25	7	11	533
2005	21	9	13	773
2006	29	13	5	1.274
2007	27	7	9	1.488
2008	28	12	11	1.413
2009	31	10	7	1.536
2010	20	4	15	1.403
2011	18	5	7	858
2012	24	11	5	961
2013	14	3	13	368
2014	7	2	9	201
2015	13	6	0	264
2016	6	0	7	181
2017	9	3	0	128
2018	6	2	5	41
2019	3	1	4	43
2020	1	1	3	12

O repositório do sistema de controle de versão⁸ mantém as informações sobre as mudanças no código-fonte (*commits*) em *logs*. Destes *logs*, foram coletadas informações acerca do número de *commits* e do número de desenvolvedores que fizeram estes *commits*. Os *commits* contados continham adições, exclusões e modificações do código-fonte extraído do repositório. O pico de *commits* foi identificado no ano de 2003, com 1.694 *commits*. O número médio de *commits* por desenvolvedor foi calculado como 34,98. O maior número de *commits* por desenvolvedor foi calculado como 70,15 no ano de 2010. O menor número de *commits* por desenvolvedor foi calculado como 6,83 no ano de 2018, conforme a Tabela 3.10.

⁸<https://svn.code.sf.net/p/jedit/svn>

A Tabela 3.11 fornece informações sobre o número de classes e pacotes envolvidos nos *commits* registrados no sistema de controle de versão do JEdit ao longo do tempo. A coluna “*Commits*” mostra o número de *commits* executados antes de liberar cada versão do sistema. Um total de 15.205 *commits* foram realizados entre 2000 e 2018 (ano da liberação da versão 5.4.0). Cada valor da coluna “*Commit*” conta o número de *commits* registrados até que outra versão fosse liberada. Como pode ser visto, o pico de número de *commits* ocorreu na versão 4.3.3 (8.164), sendo, aproximadamente, cinco vezes maior que o segundo maior valor, que ocorreu na versão 4.1.0. A versão 4.4.1 não foi apresentada por não ter tido *commits* registrados especificamente para esta versão, uma vez que a versão 4.3.3 apresentou a mesma data de geração de versão que a 4.4.1.

Tabela 3.11: Número de *commits* e sua distribuição pelas classes e pacotes para o JEdit

Versão	Commits	Commit de uma classe		Commit de um pacote		Classes por commit			Pacotes por commit		
		Núm.	%	Núm.	%	Med	μ	σ	Med	μ	σ
2.4.1	36	12	33,33	31	86,11	2	4,39	5,11	1	1,25	0,69
2.4.f	6	2	33,33	6	100	2	2	1,1	1	1	0
2.5.f	15	7	46,67	14	93,33	2	2,27	2,09	1	1,13	0,52
2.6.f	15	3	20	8	53,33	6	8,13	7,8	1	2,07	1,49
3.0.0	25	6	24	22	88	3	4,04	3,77	1	1,12	0,33
3.0.2	15	6	40	13	86,67	2	2,4	1,72	1	1,13	0,35
3.1.0	47	18	38,3	38	80,85	2	3,72	3,88	1	1,47	1,04
3.2.0	172	69	40,12	148	86,05	2	3,73	4,32	1	1,21	0,64
3.2.2	55	17	30,91	39	70,91	3	14,36	46,35	1	2,38	3,75
4.0.3	1167	365	31,28	602	51,59	3	4,65	5,29	1	2,22	1,71
4.1.0	1642	676	41,17	1003	61,08	2	3,81	6,11	1	1,86	1,46
4.2.0	1088	383	35,2	615	56,53	2	4,1	5,95	1	1,96	1,55
4.3.3	8164	4486	54,95	6169	75,56	1	2,99	9,77	1	1,48	1,33
4.4.2	282	180	63,83	222	78,72	1	2,53	7,14	1	1,3	0,75
4.5.0	526	320	60,84	412	78,33	1	3,65	30,86	1	1,48	3,36
4.5.1	217	150	69,12	185	85,25	1	1,75	1,87	1	1,25	0,78
4.5.2	175	105	60	129	73,71	1	2,52	4,6	1	1,34	0,67
5.0.0	348	197	56,61	281	80,75	1	4,17	21,1	1	1,5	2,01
5.1.0	259	163	62,93	219	84,56	1	2,05	2,25	1	1,27	0,83
5.2.0	367	190	51,77	282	76,84	1	2,57	4	1	1,44	1,38
5.3.0	271	145	53,51	214	78,97	1	2,49	3,79	1	1,39	1,07
5.4.0	197	96	48,73	131	66,5	2	3,46	5,46	1	1,71	1,39
5.5.0	116	62	53,45	84	72,41	1	2,09	1,65	1	1,41	0,75

A coluna “*Commit de uma classe*” da Tabela 3.11 conta o número de *commits* que enviam alterações no código-fonte de uma única classe, enquanto a coluna “*Commit de uma classe (%)*” mostra esse número como um percentil do número total de *commits*

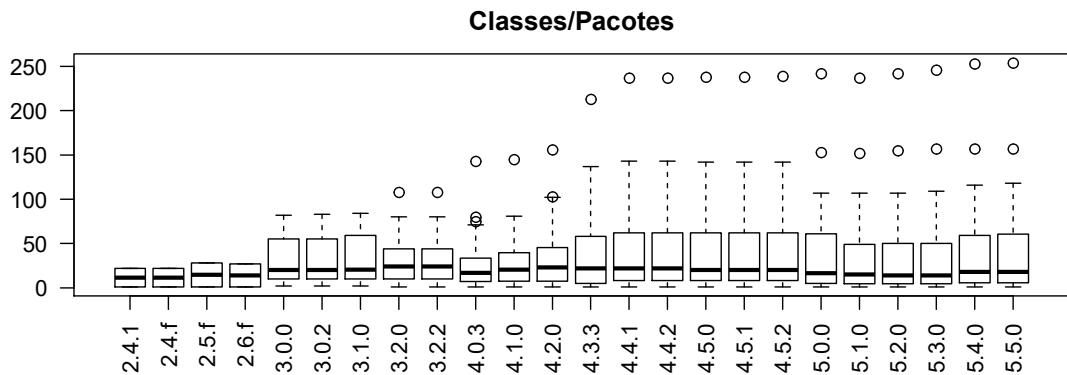


Figura 3.15: *Boxplots* para a distribuição do número de classes por pacote para o JEdit.

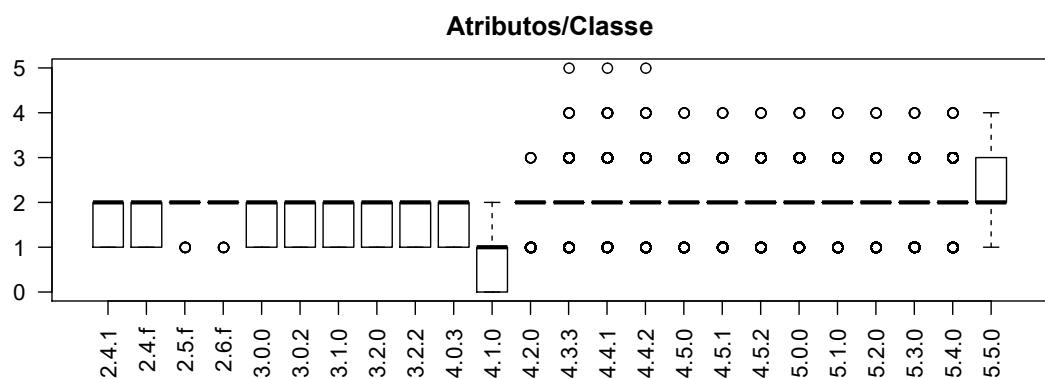


Figura 3.16: *Boxplots* para a distribuição do número de atributos por classe no JEdit.

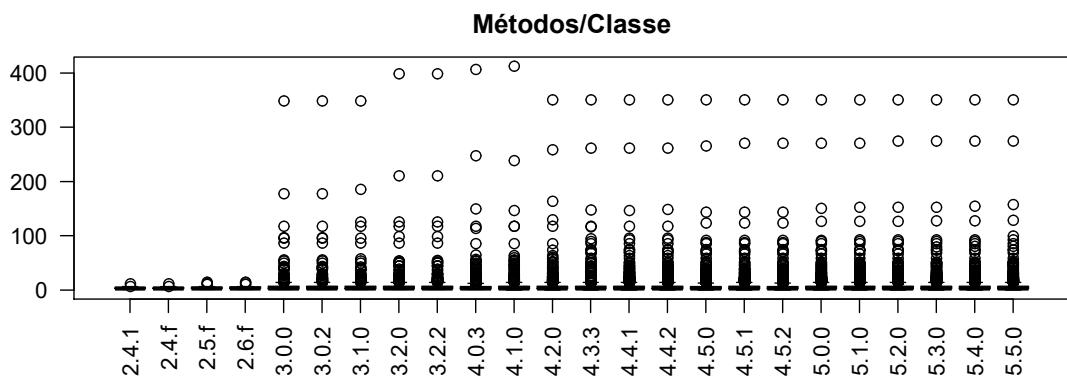


Figura 3.17: *Boxplots* para a distribuição do número de métodos por classes para o JEdit.

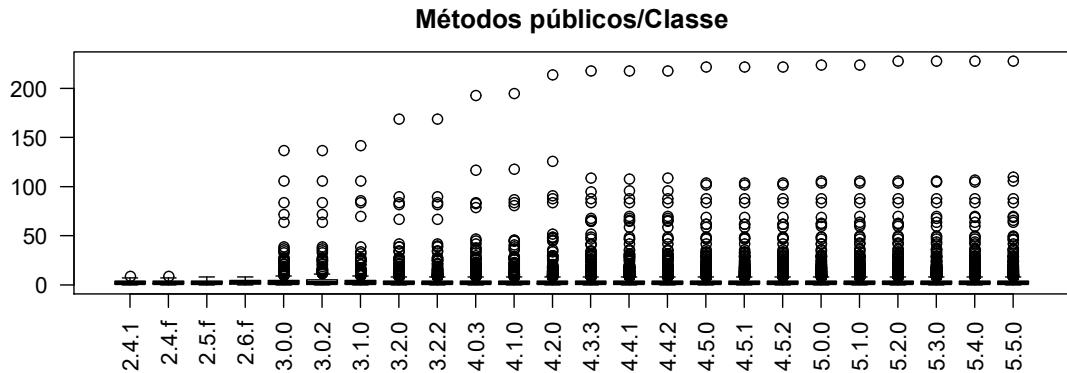


Figura 3.18: *Boxplots* para a distribuição do número de métodos públicos por classes para o JEdit.

enviados na versão. Aproximadamente, 50% dos *commits* envolveram uma única classe ao longo da evolução do JEdit. Levando em conta que durante as últimas 10 versões o percentual de *commits* de uma classe foi superior a 50%, podemos considerar que essa é uma indicação importante de que a dispersão das mudanças estava sob controle.

A coluna “Classes por *commit*” mostra a mediana entre parênteses, o número médio de classes envolvidas em *commits* e seu desvio padrão (após o sinal \pm). Os elevados valores de desvio padrão calculados ressaltam a ocorrência de poucos *commits* envolvendo um número elevado de classes. A distribuição desta medida nas versões do JEdit é apresentada nos gráficos de *boxplot* na Figura 3.19. Esses gráficos são apresentados com *outliers* e sem *outliers*. Somente ocorreram *outliers* maiores que o terceiro quartil nos *boxplot* das versões do JEdit. Alguns *commits* afetam um grande número de classes, como, por exemplo, um único *commit* envolvendo mais de 700 classes na versão 4.5.0.

Somente em seis versões do projeto ocorreram *commits* envolvendo 100 ou mais classes. Podemos considerar que estes *commits* foram correspondentes a uma refatoração de larga escala do projeto. O número médio de classes por *commit* por versão foi calculado como 3,87 e se apresentou de forma estável, com exceção do pico na versão 3.2.2, aproximadamente cinco vezes maior que a média. Também importante, o desvio padrão para o número de classes por *commit* durante essa evolução de versões

variou entre 1,0 e 10,0 na maior parte das versões, apresentando valores muito maiores em apenas três versões. Esta é mais um indicativo de que algumas mudanças envolveram um grande número de classes, mas que estes extremos não são a maioria.

Observando o gráfico sem *outliers*, é possível verificar que o número de classes por *commit* se apresentou de forma variada ao longo do tempo. Até a versão 4.3.3 identificamos 5 versões com *commits* que envolveram mais de 10 classes, de um total de 23 versões analisadas. A partir da versão 4.3.3, o número de classes por *commit* reduziu drasticamente variando em torno de três e mantendo este padrão pelas próximas oito versões, conforme ilustrado nos gráficos de *boxplot* com *outliers* na Figura 3.19. A versão 4.4.1 não foi apresentada por não ter tido *commits* registrados para si: a versão 4.3.3 apresentou a mesma data de geração de versão que a 4.4.1 e os *commits* foram considerados para a versão 4.3.3.

No geral, os dados da Tabela 3.11 indicam efeitos limitados da *Shotgun Surgery*: o número de *commits* envolvendo uma única classe é elevado e cada alteração afeta poucas classes, variando de 1,75 a 14,36 classes por *commit*, sendo este limite superior considerado como uma ocorrência isolada.

Em relação ao número de pacotes envolvidos nos *commits*, a coluna “*Commits de um pacote*” da Tabela 3.11 conta o número de *commits* envolvendo classes de um único pacote, enquanto a coluna “*Commits de um pacote (%)*” mostra esse número como um percentil do número total de *commits* para a versão. Assim como o número de *commits* de uma classe, o número de *commits* de um pacote foi elevado: em média, 76,65% dos *commits* enviados ao sistema de controle de versão envolveram um único pacote. Esta é uma segunda indicação de que a dispersão da mudança estava sob controle: os *commits* envolviam poucas classes distribuídas em poucos pacotes.

A coluna “*Pacotes por commit*” mostra a mediana entre parênteses, o número médio de pacotes envolvidos em *commits* e seu desvio padrão (após o sinal \pm). A distribuição dessa medida entre os versões analisadas é apresentada nos gráficos de *boxplot* da Figura 3.20. A versão 4.5.0 apresentou um *commit* que alterou quase 80

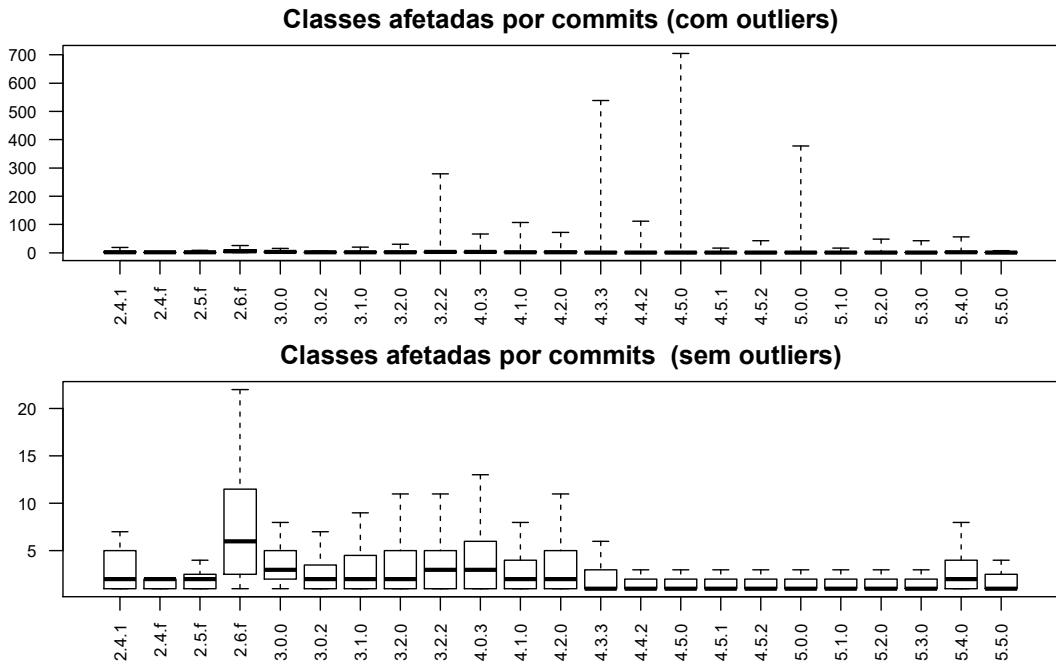


Figura 3.19: *Boxplots* para a distribuição de número de classes envolvidas em *commits* por versão do JEdit.

pacotes. Observando as medianas na Tabela 3.11, todas as versões analisadas tiveram valor igual a 1 pacote por *commit*.

Todas as versões analisadas ficaram abaixo de uma média de 2,38 pacotes alterados em *commits*.

Passando para as métricas de coesão e acoplamento, a Tabela 3.12 apresenta os valores médios dessas métricas nas versões do JEdit selecionadas para análise. A distribuição completa dessas métricas é apresentada nos gráficos de *boxplot* apresentados nas Figuras 3.21, 3.22, 3.23 e 3.24.

Tabela 3.12: Métricas de acoplamento e coesão para as versões do JEdit.

Versão	CBO ↓	AFF ↓	EFF ↓	LCOM ↓
2.4.1	0,50	0,50	1,00	0,89
2.4.f	0,50	0,50	1,00	0,89
2.5.f	0,50	0,50	1,00	0,92
2.6.f	1,00	4,50	1,50	0,92
3.0.0	3,50	23,93	17,64	0,85
3.0.2	3,50	24,07	17,71	0,85
continua na próxima página...				

...continuação da página anterior - Tabela 3.12				
Versão	CBO ↓	AFF ↓	EFF ↓	LCOM ↓
3.1.0	3,57	25,29	18,29	0,85
3.2.0	3,62	26,62	18,75	0,86
3.2.2	3,62	26,69	18,75	0,86
4.0.3	4,00	28,63	19,26	0,85
4.1.0	4,40	30,15	19,60	0,86
4.2.0	4,57	33,00	21,43	0,86
4.3.3	5,28	40,48	24,21	0,85
4.4.1	5,55	42,83	25,55	0,86
4.4.2	5,62	42,90	25,66	0,86
4.5.0	5,60	42,40	25,70	0,85
4.5.1	5,60	42,43	25,70	0,85
4.5.2	5,60	42,47	25,73	0,85
5.0.0	5,44	38,68	24,03	0,85
5.1.0	5,49	37,74	23,60	0,83
5.2.0	5,60	38,14	24,20	0,84
5.3.0	5,80	39,37	24,74	0,83
5.4.0	6,03	41,94	25,80	0,84
5.5.0	6,03	42,14	25,86	0,84

A coluna CBO na Tabela 3.12 apresenta a CBO média para os pacotes que compreendem cada versão do sistema, enquanto a coluna AFF apresenta o valor médio para o acoplamento aferente e a coluna EFF apresenta o valor médio para o acoplamento eferente. De todas as métricas, somente LCOM apresentou melhoria no valores com a evolução das versões. Por este motivo, foi identificada correlação negativa entre o LCOM e as demais métricas. As métricas CBO, AFF e EFF apresentaram correlação bastante forte e positiva entre si, conforme a Tabela 3.13.

A coluna LCOM na Tabela 3.12 apresenta o valor médio de LCOM para pacotes que compreendem cada versão do sistema. O LCOM médio se apresentou estável nas versões e com correlação forte e negativa com CBO, conforme a Tabela 3.13. Os dados indicam que os pacotes mostraram um acoplamento (CBO) variável ao longo do tempo. No entanto, é interessante ver que a coesão permaneceu quase inalterada, mesmo em períodos de variação de acoplamento.

Tabela 3.13: Análise de correlação entre as métricas utilizando o método de Spearman para o JEdit.

	CBO	AFF	EFF	LCOM
CBO	1,00	0,91	0,97	-0,68
AFF		1,00	0,95	-0,47
continua na próxima página...				

...continuação da página anterior - Tabela 3.13				
	CBO	AFF	EFF	LCOM
EFF			1,00	-0,61
LCOM				1,00

Conforme mencionado nas análises, a evolução da versões do JEdit não apresentou fortes indícios de *Shotgun Surgery* e *Scattered Functionality*. Desta forma, indicam um controle da evolução arquitetural do sistema em relação a dispersão das funcionalidades entre os pacotes e classes do sistema. Por fim, diversas publicações classificaram versões específicas do JEdit como um bom exemplo de *design* e quando analisamos a evolução do projeto arquitetural encontramos evidências sobre estas qualidades.

3.6 A Análise do JUnit

As versões analisadas nesta seção foram obtidas do repositório do projeto JUnit⁹. Neste repositório foram encontradas 28 versões, lançadas no período entre 1999 e 2020. Dentre estas versões, foram liberadas três versões principais. A versão 2 teve duas versões menores disponibilizadas no ano de 1999. A versão 3 teve oito versões liberadas no intervalo entre os anos 2000 e 2006. A versão 4 teve dezoito versões menores liberadas durante quatorze anos (entre 2006 e 2020). Identificamos a existência de uma versão 5, mas a partir desta versão o projeto do JUnit foi completamente modificado, dividindo a ferramenta em três subprojetos e dificultando a comparação com as versões anteriores.

A Figura 3.25 apresenta a linha do tempo de liberação das versões supracitadas. Houve concentração na liberação das versões em alguns períodos. Até a versão 3.8.1 foram liberadas nove versões em, aproximadamente, três anos. Em seguida, a versão 3.8.2 foi liberada após a versão 4.0.0, que seria sua evolução. Contando com a liberação da versão 4.0.0, foram liberadas nove versões entre 2006 e 2009. A versão 4.9.0 teve um intervalo de três anos em relação a versão anterior. Após a versão 4.11.0, que teve um intervalo maior que a anterior, ocorreu uma nova concentração de cinco versões no

⁹<https://sourceforge.net/projects/junit>

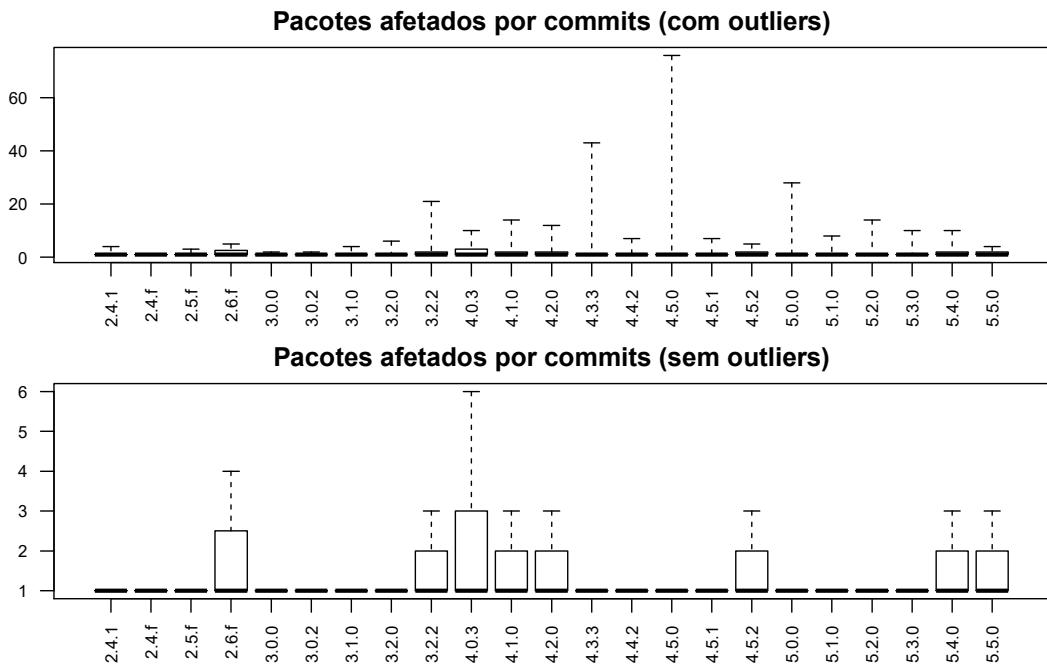


Figura 3.20: *Boxplots* para a distribuição de número de pacotes envolvidos em *commits* por versão do JEdit.

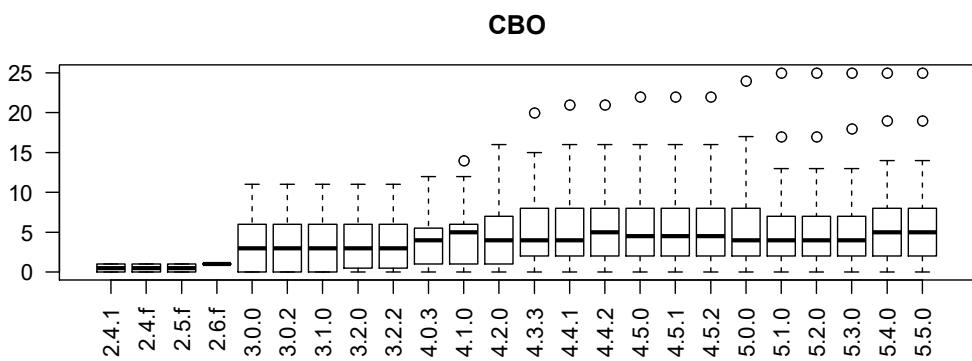


Figura 3.21: *Boxplots* para CBO por versão do JEdit.

intervalo de um ano. Por fim, a última versão em análise (4.13.0) foi a que teve o maior intervalo (5 anos) com relação à versão anterior.

Os períodos de concentração podem indicar que novas versões foram criadas com objetivo de correção de *bugs*. Por exemplo, a versão 3.8.2 parece ter sido uma versão de

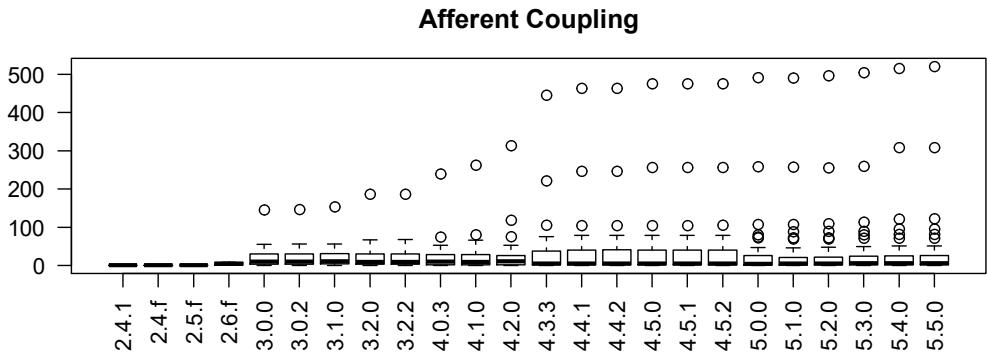


Figura 3.22: *Boxplots* para *Afferent Coupling* por versão do JEdit.

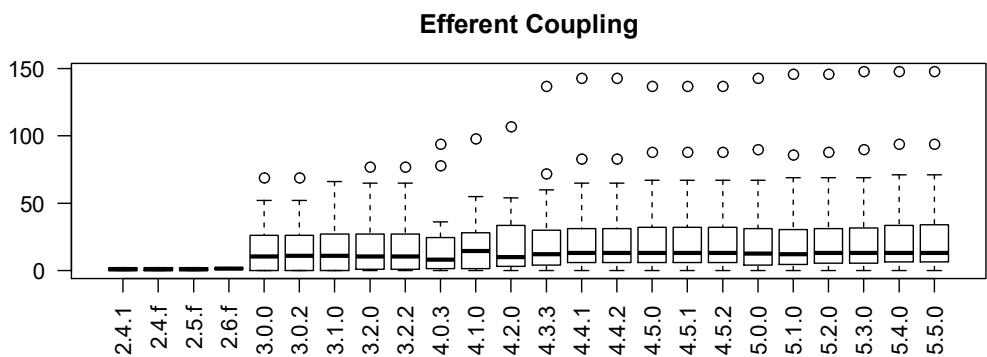


Figura 3.23: *Boxplots* para *Efferent Coupling* por versão do JEdit.

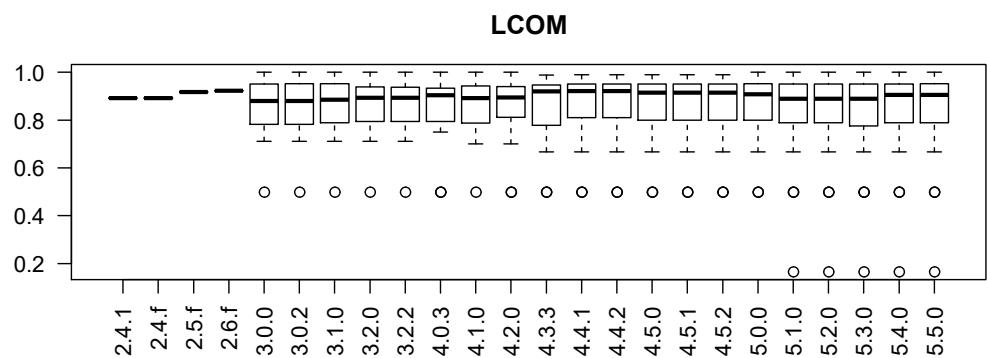


Figura 3.24: *Boxplots* para LCOM por versão do JEdit.

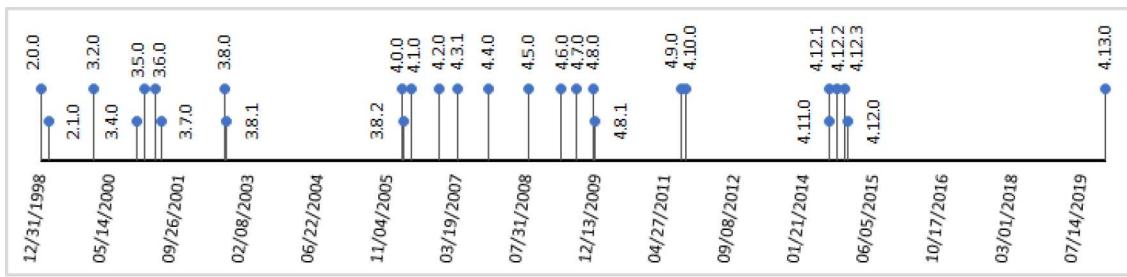


Figura 3.25: Distribuição das versões do JUnit ao longo do tempo.

correção de erros em relação à versão 3.8.1, ainda que uma nova versão (4.0.0) estivesse disponível. Em relação aos intervalos maiores entre versões, podemos considerar que representam a introdução de novas funcionalidades e maiores alterações no sistema. Mesmo possuindo um elevado número de versões, a estrutura dos arquivos encontrados no repositório do projeto estava organizada. O código-fonte e o compilado (JAR) estavam disponíveis para download. Desta forma, a identificação das versões do JUnit foi mais confiável do que para os softwares analisados nas seções anteriores.

O JUnit cresceu constantemente ao longo de seus, aproximadamente, 20 anos de existência. A Tabela 3.14 mostra os valores das métricas de tamanho e complexidade para as 28 versões do JUnit. A coluna “Pacotes” mostra o número de pacotes em uma versão. A coluna “Classes” mostra o número de classes em uma versão. A coluna “Atributos” mostra o número de atributos distribuídos pelas classes da versão. A coluna “Métodos” mostra o número de métodos nas classes da versão, enquanto a coluna “Métodos Públicos” mostra quantos desses métodos são públicos. Por fim, a coluna “Dependências” mostra o número de dependências em uma versão.

Tabela 3.14: Tamanho e complexidade das versões do JUnit.

Versão	Pacotes	Classes	Atributos	Métodos	Mét. Públicos	Dependências
2.0.0	4	38	61	203	123	84
2.1.0	4	38	61	213	129	91
3.2.0	6	88	153	443	240	218
3.4.0	7	78	135	443	242	205
3.5.0	7	100	169	555	325	277
3.6.0	7	103	175	566	332	280
3.7.0	7	99	167	564	334	274
3.8.0	6	101	169	594	358	275

continua na próxima página...

...continuação da página anterior - Tabela 3.14						
Versão	Pacotes	Classes	Atributos	Métodos	Mét. Públcos	Dependências
3.8.1	6	100	167	591	358	276
3.8.2	6	102	169	616	372	278
4.0.0	11	92	170	555	391	341
4.1.0	11	96	181	560	393	359
4.2.0	12	98	183	569	399	362
4.3.1	12	99	185	588	416	370
4.4.0	20	154	293	859	639	571
4.5.0	26	188	365	1.051	729	702
4.6.0	28	204	404	1.134	785	779
4.7.0	29	225	447	1.212	828	839
4.8.0	30	230	459	1.224	837	853
4.8.1	30	230	459	1.227	837	853
4.9.0	31	250	492	1.276	859	883
4.10.0	31	252	493	1.299	869	889
4.11.0	28	233	462	1.305	830	766
4.12.0	30	286	582	1.627	973	1.027
4.12.1	30	268	529	1.509	927	940
4.12.2	30	283	576	1.606	968	1.012
4.12.3	30	286	582	1.627	973	1.027
4.13.0	32	348	728	1.869	1.087	1.283

Analizando a evolução do software apresentada na Tabela 3.14, é possível perceber que o JUnit tem crescido constantemente desde suas versões iniciais em termos de número de pacotes e classes. O número médio de classes por pacote é 10,36 (mediana = 9,18, desvio padrão = 3,21, mínimo = 7,23 e máximo = 17,00), mas esse número não ficou estável ao longo dos anos, variando de 7,23 classes por pacote (na versão 4.5.0) até 17 classes por pacote na versão 3.8.2. A partir da versão 4.0.0, foi identificada uma redução do número de classes por pacote, sendo mantida até a última versão liberada.

A primeira versão analisada (2.0.0) tinha 38 classes localizadas em 4 pacotes e a última versão (4.13.0) alcançou 348 classes distribuídas em 32 pacotes, representando um aumento em mais de 9 vezes o tamanho da primeira versão do software em termos no número de classes. Ao longo do desenvolvimento das versões 3, as métricas não apresentaram grande crescimento, mas na versão 4 iniciou o crescimento de todas as métricas de tamanho com grande intensidade. O maior aumento identificado no número de classes foi entre as versões 3.2.0 e 2.1.0 (131%). A segunda versão que agregou mais classes foi a versão 4.4.0, 55% a mais do que a anterior (4.3.1). A evolução da primeira versão analisada (2.0.1) até a última versão analisada (4.13.0) representou um aumento em mais de 9 vezes o tamanho original do software.

A distribuição de dados ou funcionalidades entre as classes, por outro lado, é estável em diferentes versões. O número médio de atributos por classe é 1,86 (mediana = 1,89, desvio padrão = 0,15, mínimo = 1,61 e máximo = 2,09). O número médio de métodos por classe é 5,59 (mediana = 5,60, desvio padrão = 0,26, mínimo = 5,03 e máximo = 6,04). O número médio de métodos públicos por classe é 3,56 (mediana = 3,50, desvio padrão = 0,36, mínimo = 2,73 e máximo = 4,25). Por fim, uma classe depende, em média, de 3,28 outras classes (mediana = 3,55, desvio padrão = 0,52, mínimo = 2,21 e máximo = 3,82).

Buscando concentrar as análises nas versões que apresentaram variação estrutural, as versões com o mesmo número de pacotes e classes foram desconsideradas, mantendo somente a versão mais nova. As versões que foram citadas pelas publicações analisadas (3.8.0, 3.8.2, 4.1.0, 4.8.1, 4.11 e 4.12) também foram mantidas, uma vez que outros pesquisadores já as analisaram. Sendo assim, na Tabela 3.14, foram hachuradas as versões que foram desconsideradas das análises seguintes (2.0.0 e 4.8.0).

As versões analisadas são apresentadas nos gráficos de *boxplot* para cada versão do JUnit: a Figura 3.27 mostra a distribuição do número de classes por pacote, a Figura 3.28 mostra o número de atributos por classe, a Figura 3.29 mostra o número de métodos por classe e a Figura 3.30 mostra o número de métodos públicos por classe. Os *outliers* são apresentados como pontos sobre os limites superiores ou abaixo dos limites inferiores. A Figura 3.26 apresenta um grafo para cada uma das versões em análise, sendo os pacotes representados pelos nós e as relações de dependência representadas pelas arestas destes grafos. Analisando esta evolução de versões, é possível verificar o crescimento das relações de dependências ao longo das versões.

A Tabela 3.15 apresenta coeficientes de correlação de Spearman entre as métricas de tamanho e complexidade. Uma medida de correlação não-paramétrica é usada porque o número de classes nas versões do JUnit, sendo definido como um número inteiro no intervalo $[0, +\infty)$, não pode se parecer com uma distribuição normal. A análise de correlação mostra que as métricas apresentaram correlação muito forte e positiva, no

intervalo de 0,87 a 0,99. Por este motivo, é possível perceber que as métricas são igualmente representativas do tamanho e complexidade do sistema.

Tabela 3.15: Análise de correlação entre as medidas de tamanho/complexidade e o número de classes do JUnit

Métrica correlacionada	Pacotes	Classes	Atributos	Métodos	Métodos Públicos	Dependências
Pacotes	1,00	0,87	0,95	0,87	0,95	0,96
Classes		1,00	0,94	0,98	0,93	0,93
Atributos			1,00	0,95	0,98	0,99
Métodos				1,00	0,96	0,94
Métodos Públicos					1,00	0,99
Dependências						1,00

O repositório do sistema de controle de versão do projeto JUnit¹⁰ mantém as informações sobre as mudanças no código-fonte (*commits*) em *logs*. Destes *logs*, foram coletadas informações acerca do número de *commits* e do número de desenvolvedores que fizeram estes *commits*. Apesar de analisarmos versões que foram liberadas em 1999, somente foram encontrados registros de *logs* do controle de versões iniciando em 11 de dezembro de 2000. Considerando que a última versão em análise foi liberada em janeiro de 2020, os *commits* registrados após a liberação desta versão foram descartados.

Analizando a equipe responsável pelo desenvolvimento do JUnit, é possível perceber que ao longo da evolução do sistema a equipe teve até 40 desenvolvedores. Nos primeiros dez anos do projeto, o número máximo de desenvolvedores foi de quatro (em 2002). Após o ano de 2009, é possível identificar uma grande rotatividade entre os desenvolvedores. A partir de 2017, o número de desenvolvedores na equipe foi reduzido, apresentando somente 8 desenvolvedores em 2020, conforme a Tabela 3.16.

Tabela 3.16: Tamanho da equipe, entrada e saída de desenvolvedores e número de *commits* por ano para o projeto JUnit.

Ano	Equipe	Entrada	Saída	Commits
2000	1	1	0	5
2001	3	2	0	23
continua na próxima página...				

¹⁰<https://svn.code.sf.net/p/junit/svn>

...continuação da página anterior- Tabela 3.16				
Ano	Equipe	Entrada	Saída	Commits
2002	4	1	0	48
2003	1	0	3	3
2004	3	2	0	24
2006	3	2	2	34
2007	3	2	2	34
2008	2	0	1	141
2009	2	0	0	98
2010	10	9	1	242
2011	6	3	7	56
2012	19	17	4	124
2013	38	28	9	312
2014	33	24	29	239
2015	40	32	25	161
2016	22	17	35	38
2017	28	24	18	78
2018	9	6	25	50
2019	8	5	6	44
2020	8	4	4	33

O pico de *commits* foi identificado no ano de 2013, com 312 *commits*. O número médio de *commits* por desenvolvedor foi calculado como 12,93. O maior número de *commits* por desenvolvedor foi calculado como 70,15 no ano de 2008. O menor número de *commits* por desenvolvedor foi calculado como 3 no ano de 2003.

A Tabela 3.17 fornece informações sobre o número de classes e pacotes envolvidos nos *commits* feitos no sistema de controle de versão do JUnit ao longo do tempo. A coluna *Commits* mostra o número de *commits* feitos antes de liberar cada versão do sistema. Um total de 1.753 *commits* foram enviados ao sistema de controle de versão de 2000 a 2020. Como não foram encontrados registros de *commits* antes do ano 2000, a Tabela 3.17 inicia na versão 3.5.0. Cada valor da coluna *Commit* conta o número de *commits* identificados até que outra versão fosse liberada. O pico de número de *commits* por versão ocorreu na versão 4.11.0 (662 *commits*), representando mais do que o dobro que o segundo pico (266 *commits*), que ocorreu na última versão (4.13.0).

Tabela 3.17: Número de *commits* e sua distribuição pelas classes e pacotes para o JUnit

Versão	Commits	Commit de uma classe		Commit de um pacote		Classes por commit			Pacotes por commit		
		Núm.	%	Núm.	%	Med	μ	σ	Med	μ	σ
3.5.0	8	2	25	3	37,5	2,5	15	22,88	2,5	3,75	3,45

continua na próxima página...

...continuação da página anterior - Tabela 3.17

Versão	Commits	Commit de uma classe		Commit de um pacote		Classes por commit			Pacotes por commit		
		Núm.	%	Núm.	%	Med	μ	σ	Med	μ	σ
3.6.0	2	0	0	0	0	13,5	13,5	6,36	6,5	0,71	
3.8.0	55	27	49,09	32	58,18	2	4,96	9,98	1	2,25	2,18
3.8.1	11	3	27,27	7	63,64	2	2,45	1,37	1	1,73	1,19
3.8.2	4	2	50	2	50	3,5	26	46,73	2,5	5,5	7,14
4.0.0	28	15	53,57	15	53,57	1	12,43	44,49	1	3,11	5,88
4.1.0	5	1	20	1	20	4	7	7,75	2	5,8	8,01
4.2.0	15	6	40	6	40	2	4,93	8,58	2	2,8	2,81
4.3.1	25	4	16	4	16	5	20,72	65,44	3	4,96	8,35
4.4.0	51	3	5,88	4	7,84	7	13,98	18,64	4	5,76	5,05
4.5.0	152	58	38,16	63	41,45	2	7,66	39,78	2	3,44	7,49
4.6.0	44	6	13,64	7	15,91	3,5	16,84	55,15	2,5	5,66	10,3
4.7.0	174	51	29,31	60	34,48	3	5,76	8,76	2	3,37	3,44
4.8.1	33	7	21,21	9	27,27	3	5,64	7,5	2	4	4,97
4.9.0	146	45	30,82	48	32,88	3	8,12	13,58	2	4,64	6,37
4.10.0	34	12	35,29	12	35,29	3,5	10,38	16,04	3	5,68	7
4.11.0	662	246	37,16	260	39,27	2	13,7	40,09	2	5,77	9,79
4.12.0	7	4	57,14	4	57,14	1	2,43	2,15	1	2,43	2,15
4.12.2	21	6	28,57	6	28,57	3	14,9	29,78	3	6,67	9,99
4.12.3	10	6	60	6	60	1	1,7	1,25	1	1,6	0,97
4.13.0	266	130	48,87	144	54,14	2	9,27	35,46	1	4	10,32

A coluna “Commit de uma classe” da Tabela 3.17 conta o número de *commits* que enviam alterações no código-fonte de uma única classe, enquanto a coluna “Commit de uma classe (%)” mostra esse número como um percentil do número total de *commits* realizados nessa versão. Em média, 32,71% dos *commits* envolveram uma única classe ao longo da evolução da JUnit. Entretanto, quando consideramos somente as últimas 5 verões analisadas, esta média sobe para, aproximadamente, 46%. Por esta elevação do percentual nas últimas versões alcançar um valor próximo dos 50% dos *commits*, podemos considerar que essa é uma indicação importante de que a dispersão das mudanças estava sob controle.

A coluna “Classes por commit” mostra a mediana entre parênteses, o número médio de classes envolvidas em *commits* e seu desvio padrão (após o sinal \pm). Os elevados valores de desvio padrão calculados ressaltam a ocorrência de poucos *commits* envolvendo um número elevado de classes. A distribuição desta medida nas versões do JUnit é apresentada nos gráficos de boxplot na Figura 3.31. Esses gráficos são apresentados com *outliers* e sem *outliers*. Confirmando a observação realizada com base nos valores dos desvios padrão, alguns *commits* afetam um grande número de classes,

como, por exemplo, um único *commit* envolvendo em torno de 500 classes na versão 4.5.0. Em seis versões ocorreram *commits* envolvendo 100 ou mais classes. Podemos considerar que estes *commits* correspondem a refatorações do projeto JUnit, ocorrendo em versões com mudança do segundo número dentro da linha de versões 4 (4.0.0, 4.3.1, 4.5.0, 4.6.0, 4.11.0 e 4.13.0).

O número médio de classes por *commit* por versão foi calculado em 10,35 e se apresentou de forma um pouco organizada, com exceção do pico na versão 3.8.2, aproximadamente 2,5 vezes maior que a média. Também importante, o desvio padrão para os pontos de dados dessa evolução de versões se apresentou variando entre 1,25 e 65,44. Assim, é possível perceber que os *commits* envolvendo um grande número de classes não são a maioria. Como os *outliers* reduzem o espaço vertical disponível para mostrar a parte central das distribuições (entre o primeiro e o terceiro quartis), apresentamos um boxplot com *outliers* para mostrar os valores extremos e um boxplot sem *outliers* para destacar o IQR curto apresentado por todas as versões.

Mesmo no boxplot sem *outliers*, é possível verificar que o número de classes por *commit* se apresentou de forma variada entre as versões. Até a versão 4.3.1 identificamos cinco versões com *commits* que envolveram mais de 10 classes, de um total de 15 versões analisadas. Na versão 4.4.0 ocorreu uma elevação e, em seguida, o teto de 20 classes foi respeitado até a última versão. Desta forma, dentre as 21 versões analisadas, somente três ultrapassaram as 20 classes, conforme ilustrado nos gráficos de boxplot com *outliers* na Figura 3.31.

Em relação ao número de pacotes envolvidos nos *commits*, a coluna *Commits* de um pacote na Tabela 3.17 conta o número de *commits* envolvendo classes de um único pacote, enquanto a coluna *Commits* de um pacote (%) mostra esse número como um percentil do número total de *commits* para a versão. Assim como o número de *commits* de uma classe, o número de *commits* de um pacote não foi tão elevado: em média, 39,53% dos *commits* enviados ao sistema de controle de versão envolveram um único pacote. Entretanto, quando consideramos as últimas 5 versões analisadas, esta média sobre para, aproximadamente, 48%. Por esta elevação do percentual nas últimas versões alcançar um

valor próximo dos 50% dos *commits*, temos uma segunda indicação de que a dispersão da mudança estava sob controle.

A coluna “Pacotes por *commit*” mostra a mediana entre parênteses, o número médio de pacotes envolvidos em *commits* e seu desvio padrão (após o sinal \pm). A distribuição dessa medida entre as versões é apresentada nos gráficos de boxplot da Figura 3.32. A versão 4.5.0, que registrou o maior *outlier* na análise de classes afetadas por *commits*, também apresentou o maior *outlier* de, aproximadamente, 90 pacotes em um *commit*. Observando as medianas na Tabela 3.17, todas as versões analisadas ficaram abaixo de 7 pacotes por *commit*.

Passando para as métricas de coesão e acoplamento, a Tabela 3.18 apresenta os valores médios dessas métricas para cada uma das versões. A distribuição completa dessas métricas é apresentada nos diagramas de boxplot apresentados nas Figuras 3.33, 3.34, 3.35 e 3.36.

Tabela 3.18: Métricas para as Versões do JUnit

Versão	CBO ↓	AFF ↓	EFF ↓	LCOM ↓
2.0.0	0,75	2,50	2,25	0,70
2.1.0	0,75	2,50	2,25	0,70
3.2.0	1,17	5,50	5,83	0,87
3.4.0	1,29	4,71	5,29	0,85
3.5.0	1,29	6,14	6,43	0,88
3.6.0	1,43	6,14	6,57	0,88
3.7.0	1,43	6,14	6,57	0,88
3.8.0	1,50	7,33	7,50	0,82
3.8.1	1,50	7,33	7,50	0,82
3.8.2	1,50	6,83	7,50	0,82
4.0.0	2,55	8,55	8,45	0,81
4.1.0	2,55	8,64	8,45	0,81
4.2.0	2,42	8,08	7,83	0,81
4.3.1	2,67	8,33	8,08	0,81
4.4.0	2,80	7,40	8,35	0,76
4.5.0	3,35	8,31	8,62	0,80
4.6.0	3,50	8,89	8,79	0,79
4.7.0	3,66	9,24	9,10	0,79
4.8.1	3,67	9,10	9,00	0,78
4.9.0	3,71	9,48	9,16	0,82
4.10.0	3,74	9,61	9,19	0,82
4.11.0	3,86	9,64	8,71	0,81
4.12.0	3,97	11,23	9,57	0,78
4.12.1	3,90	10,47	9,17	0,78
4.12.2	3,97	11,13	9,57	0,78
4.12.3	3,97	11,23	9,57	0,78
4.13.0	4,12	12,50	10,47	0,79

A coluna CBO na Tabela 3.18 apresenta a CBO média para os pacotes que compreendem cada versão do sistema, enquanto a coluna AFF apresenta o valor médio para o acoplamento aferente e a coluna EFF apresenta o valor médio para o acoplamento eferente. Os dados indicam que os pacotes mostraram um acoplamento crescente ao longo do tempo. As métricas CBO, AFF e EFF apresentaram correlação bastante forte e positiva entre si, conforme a Tabela 3.19.

A coluna LCOM na Tabela 3.18 apresenta o valor médio de LCOM para os pacotes de cada versão do sistema. É interessante ver que a coesão sofreu pequenas alterações entre as versões, mesmo em transições que apresentam grande variação de acoplamento.

Tabela 3.19: Análise de correlação (Spearman) entre as métricas.

	CBO	AFF	EFF	LCOM
CBO	1,00	0,98	0,98	-0,41
AFF		1,00	0,98	-0,37
EFF			1,00	-0,38
LCOM				1,00

Conforme mencionado nas análises, a evolução da versões do JUnit apresentou indícios de *Shotgun Surgery* e *Scattered Functionality*. Desta forma, indicam ausência de controle da evolução arquitetural do sistema em relação à dispersão das funcionalidades entre os pacotes e classes do sistema. Apesar de diversas publicações terem classificado uma versão específica do JUnit como um bom exemplo de design, quando analisamos a evolução do projeto arquitetural, isto não foi identificado.

3.7 Considerações Finais

Diante das análises que foram realizadas nos três sistemas (JHotDraw, JEdit e JUnit) que foram indicados por outros pesquisadores como exemplos de qualidade, foi possível perceber que quando uma versão é analisada de forma isolada, ela pode apresentar bons valores para métricas estruturais. Entretanto, quando analisamos a evolução do sistema, considerando o acréscimo de funcionalidades e a liberação de novas versões ao longo de diversos anos, percebemos que o resultado é um pouco diferente.

Analisando sob o aspecto do controle de alterações dos sistemas e a dispersão de alterações, os três sistemas não apresentaram as mesmas características. O JHotDraw apresentou resultados que indicam que o controle da evolução não estava sendo realizado da melhor forma, ou seja, as alterações envolviam diversas classes e pacotes com frequência. O JEdit e o JUnit apresentaram resultados que indicam que o controle da evolução estava sendo realizado, ou seja, as alterações não envolviam diversas classes e pacotes com frequência. O JEdit apresentou melhores resultados, quando comparados com o JUnit.

Analisando sob o aspecto da evolução do acoplamento, os três sistemas apresentaram as mesmas características. Foi possível identificar que a métrica CBO apresentou piora nas análises das últimas versões, ou seja, aumentou o acoplamento entre os pacotes, na medida que foram lançadas novas versões para os três sistemas. Em relação à coesão, a métrica LCOM apresentou leve melhoria para JHotDraw, se manteve bastante estável para JEdit e piorou para o JUnit.

Analisando sob o aspecto da evolução do coesão, os três sistemas não apresentaram as mesmas características. A métrica LCOM apresentou leve melhoria para JHotDraw, se manteve bastante estável para JEdit e piorou para o JUnit.

Desta forma, os sistemas JEdit e JUnit apresentaram um evolução estrutural adequada para seguir como exemplo de qualidade de *design*.

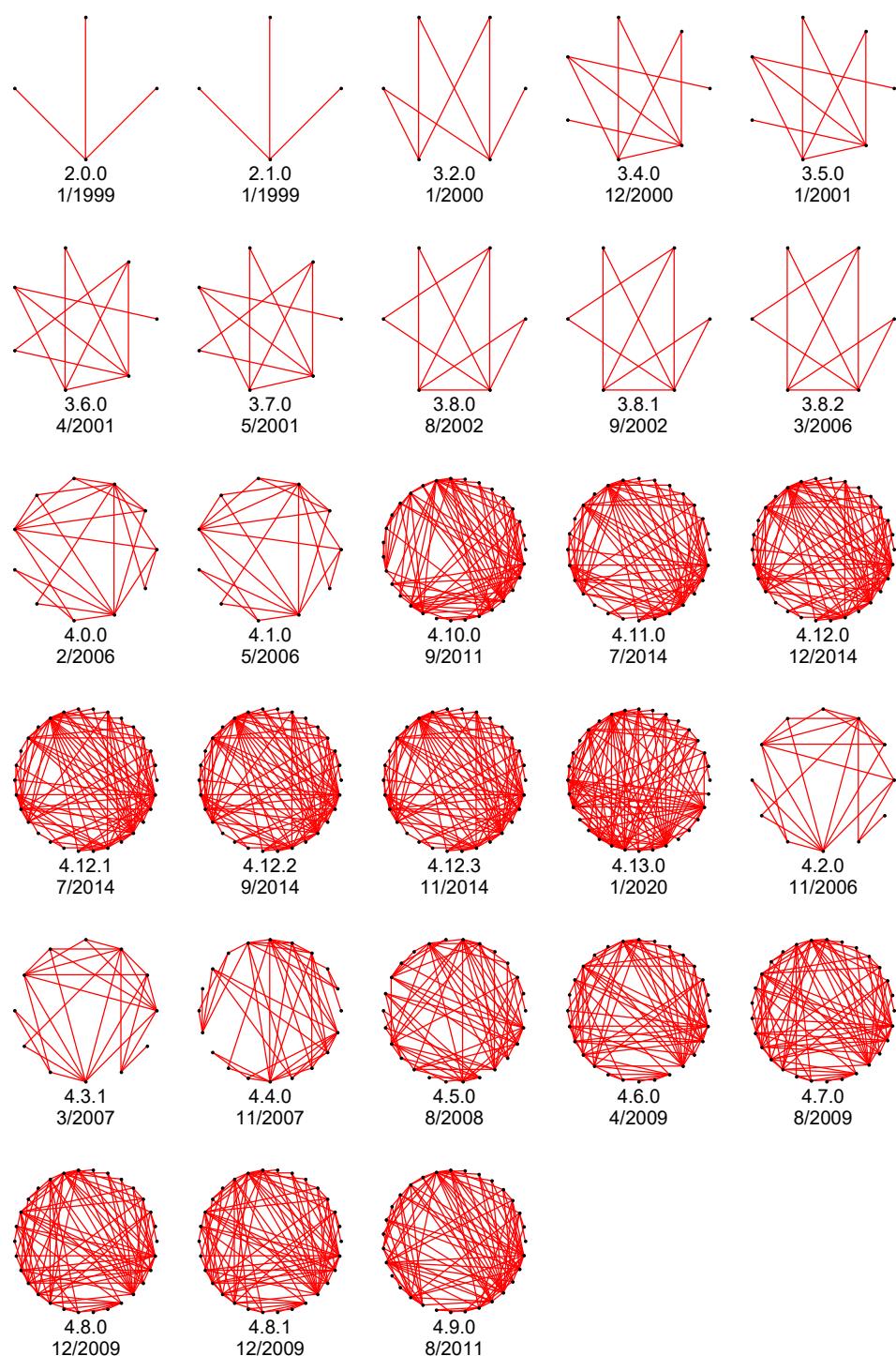


Figura 3.26: Evolução da arquitetura do JUnit

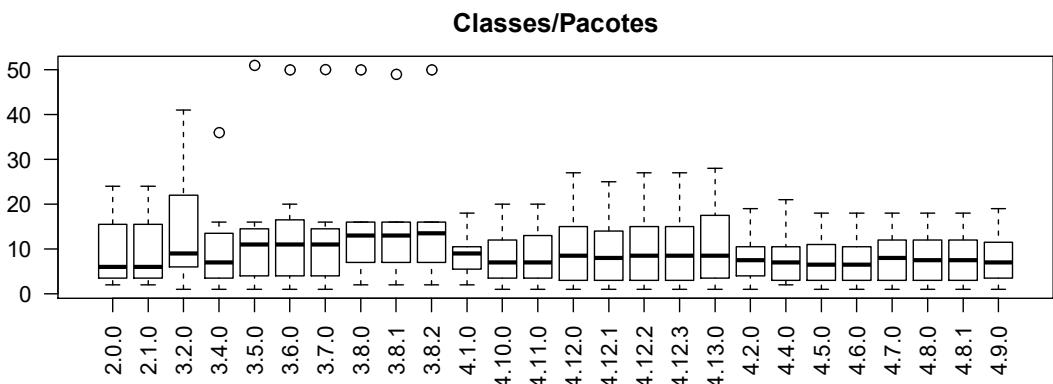


Figura 3.27: *Boxplots* para a distribuição do número de classes por pacote para o JUnit.

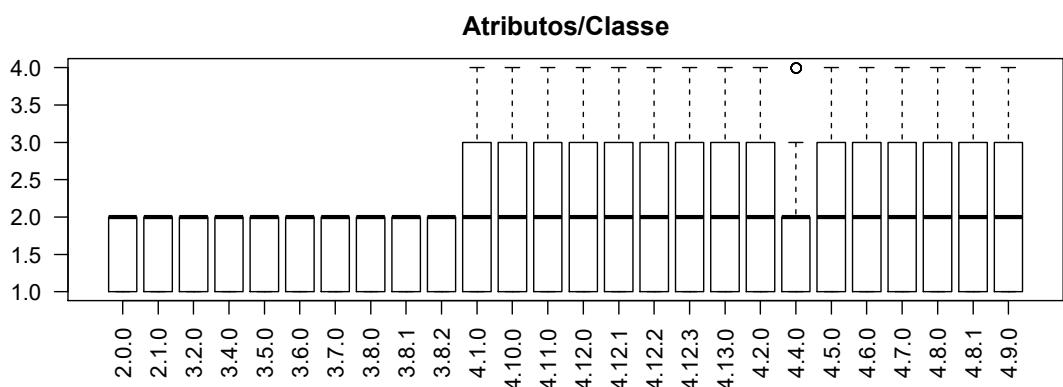


Figura 3.28: *Boxplots* para a distribuição do número de atributos por classe para o JUnit.

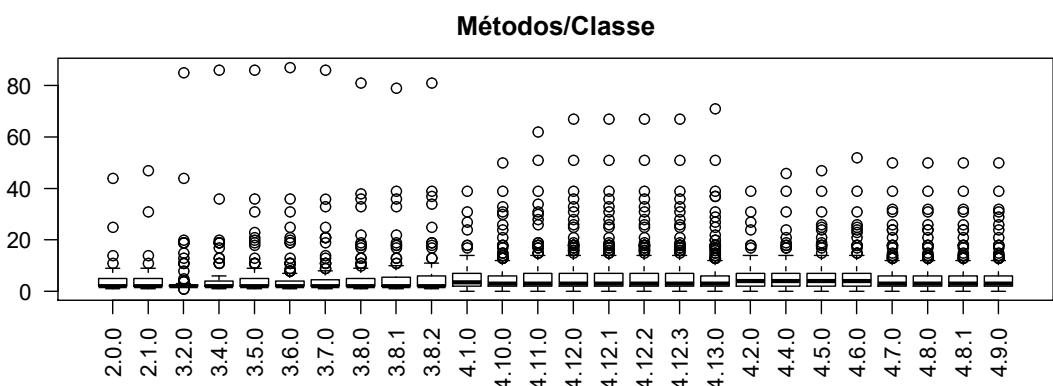


Figura 3.29: *Boxplots* para a distribuição do número de métodos por classe para o JUnit.