6
Estudos Experimentais

A ferramenta QCDTool (Capítulo 5) foi utilizada no contexto de dois estudos experimentais de domínio distintos, com características, níveis de complexidade diferentes. O primeiro estudo avalia as estratégias de detecção propostas para o modelo (Capítulo 4) de acordo com o grau de correlação com que detectam as mesmas classes com problemas de *design* que suas correspondentes para código. Parte dos resultados está disponível em (Macía et al, 2008). Além disso, como uma análise complementar da qualidade dos resultados obtidos, também foram avaliadas as medidas de acurácia, precisão e *recall*. O estudo envolveu um conjunto de nove sistemas com tamanhos e domínios distintos e desenvolvidos por pessoas atuando em ambientes diferentes. O segundo estudo teve como objetivo avaliar a usabilidade do modelo da qualidade QMOOD em diagramas de classes. Este modelo da qualidade foi aplicado em uma seqüência de versões do framework JHotdraw (2006) utilizando a ferramenta QCDTool.

A avaliação de abordagens e técnicas da engenharia de software é uma tarefa notoriamente complicada. A organização destes estudos baseou-se na estrutura proposta por Wohlin et al (2000) para estudos experimentais em engenharia de software. Esta estrutura foi usada para organizar os estudos em termos de seus objetivos, definição, planejamento e execução. A estrutura proposta por Wohlin et al (2000) vem ganhando popularidade, e muitos estudos experimentais dentro da comunidade de engenharia de software têm sido descritos seguindo esse padrão. Tal estrutura sugere a realização das seguintes tarefas: definição, planejamento, operação, análise e interpretação, validação e empacotamento.

6.1. Avaliação das Estratégias de Detecção

Esta seção descreve o contexto e os resultados do primeiro estudo experimental. O objetivo desse estudo foi o de servir como uma primeira avaliação das estratégias de detecção propostas para modelo. Este estudo foi projetado para verificar se é possível detectar alguns dos problemas que afetam o *design* dos sistemas no código já desde as primeiras fases de desenvolvimento do software: a etapa de modelagem. Ou seja, avaliar a correlação entre as entidades problemáticas de *design* detectadas no modelo utilizando as estratégias de detecção propostas e as entidades problemáticas detectadas no código usando as estratégias definidas em (Lanza & Marinescu, 2006; Marinescu, 2002).

Para complementar a análise da correlação das estratégias de detecção, também foi avaliada a eficácia baseada em três medidas específicas: acurácia, precisão e *recall*. Estas medidas são normalmente utilizadas para avaliar a eficácia de um sistema de recuperação de informação e estão definidas no contexto de um sistema de informação da seguinte forma: a acurácia representa o grau de veracidade dos resultados; a precisão corresponde à relevância dos resultados obtidos de acordo à consulta realizada, e o *recall* representa a habilidade para a recuperação dos resultados relevantes de acordo à consulta realizada (Baeza & Ribeiro, 1999). Para a análise destas medidas foram utilizados dados coletados de nove sistemas diferentes que serão descritos a seguir.

6.1.1. Sistemas Envolvidos

O contexto deste estudo envolveu um conjunto de nove sistemas:

S1: Framework para a geração de relatórios estatísticos de gerenciamento de incidentes. Além da geração dos relatórios, o framework envia-os periodicamente a destinatários e emite alertas com relação a determinadas situações do sistema. Além disso, ele tem independência da modelagem e do banco de dados utilizados, da forma de envio de notificações e das situações em que os alertas devem ser emitidos.

S2: Framework para o gerenciamento de coleções virtuais. Este sistema permite armazenar, gerenciar e disponibilizar as coleções digitais dos usuários. O

gerenciamento do sistema inclui documentos, fotografias, arquivos de áudio e vídeo.

S3: Linha de produto de software de locadora de filmes. Este sistema consiste no gerenciamento, locação, e devolução de vídeos, que inclui gerenciamento de títulos e cópias dos títulos. Além disso, o sistema realiza o gerenciamento de clientes que fazem as locações e o gerenciamento de funcionários da locadora.

S4: Framework para a aplicação de métricas no código fonte. Este framework tem o objetivo de permitir a um desenvolvedor construir aplicações que pudessem extrair métricas de código fonte da linguagem de programação Java, com pouco esforço, inclusive permitindo criar suas próprias métricas.

S5: Ferramenta QCDTool apresentada no Capítulo 5.

S6, S7, S8 e S9 correspondem respectivamente às versões 5.2, 6.0, 7.0 e 7.1 do sistema *opensource* JHotdraw (2006). JHotdraw é um framework para a construção de editores de gráficos. Cada instância do framework corresponde a um domínio específico e reflete a semântica desse domínio por meio da definição de tipos de figuras a serem utilizadas e as relações específicas entre elas.

Os primeiros quatro sistemas foram desenvolvidos como trabalho da disciplina de Projeto de Sistemas de Software do programa de pós-graduação em informática da PUC-Rio. Esta disciplina avalia a correspondência dos diagramas UML com o código implementado e os modelos são criados pelos próprios desenvolvedores como parte do processo de desenvolvimento dos sistemas. Os diagramas de classes para o quinto sistema, também foram gerados pelo desenvolvedor como parte do processo de desenvolvimento. A principal motivação para incluir as diferentes versões do sistema JHotdraw neste estudo foi o fato deste sistema ter sido utilizado e referenciado em muitas outras pesquisas acadêmicas. Todos os sistemas foram desenvolvidos utilizando a linguagem de programação Java e suas características são apresentadas na Tabela 6.

Tabela 6 – Características dos sistemas envolvidos no estudo.

Sistema	Num. Pacotes	Num. Classes	Num. Métodos	Num. Atributos
S1	12	61	513	142
S2	38	102	529	218
S3	27	110	898	194
S4	18	96	459	248
S5	16	150	679	459
S6	17	172	1.438	363
S7	16	313	3.117	361
S8	21	332	3.251	735
S9	25	401	3.986	1.198

6.1.2. Estrutura do Estudo

A descrição da estrutura do estudo de acordo com o formato proposto por Wohlin et al (2000) está composta pelas seguintes tarefas: definição, planejamento e operação.

6.1.2.1. Definição do Estudo

De acordo a sugestão do Wohlin et al (2000), utilizou-se o padrão GQM (Vasili & Weiss, 1984; Basili & Rombach, 1988; Solinger & Berghout, 1999) para a definição do estudo. Sendo assim, os objetivos deste estudo são definidos a seguir:

Analisar: as estratégias de detecção para modelos

Com o propósito de: avaliar sua correlação, acurácia, precisão e *recall* na identificação das entidades portadoras de problemas de *design*

Com respeito a: identificação realizada pelas estratégias de detecção correspondentes para código

Do ponto de vista: do pesquisador

No contexto de: estudantes da pós-graduação do Laboratório de Engenharia de Software da PUC-Rio.

6.1.2.2. Planejamento do Estudo

Seleção do contexto e participantes. O estudo experimental foi desenvolvido no laboratório de engenharia de software (LES) da PUC-Rio e,

portanto. O estudo foi focado na análise da correspondência entre os resultados obtidos da aplicação das estratégias de detecção propostas para modelo e as existentes para código, utilizando para isto, nove sistemas (S1-S9) descritos na 6.1.1. Essa análise envolveu apenas um estudante de mestrado na área engenharia de software, o qual possui conhecimento sobre *design* OO, padrões de projeto e modelos UML. Esta escolha foi realizada por conveniência. Isso levou o estudo a ser classificado como quase-experimento (Wohlin et al, 2000) já que não possui randomização dos participantes. Conseqüentemente, a possibilidade de generalizar os resultados obtidos a partir deste contexto é baixa. Este ponto é abordado com maior detalhe na seção de ameaças à validade do estudo (Seção 6.3).

Seleção das hipóteses. Para a definição das hipóteses usamos o verbo "classificar" com o significado de "decidir se uma classe possui o problema de *design* ou não". Além disso, as hipóteses foram colocadas no singular já que elas foram testadas isoladamente para cada estratégia de detecção. As hipóteses utilizadas neste estudo foram definidas formalmente da seguinte forma:

- Hipótese nula, H_0: Não existe correlação entre as entidades de *design* classificadas como problemáticas pela estratégia de detecção proposta para modelo e sua homóloga para código fonte.
- Hipótese alternativa, H_1: Existe correlação entre as entidades de *design* classificadas como problemáticas pela estratégia de detecção proposta para modelo e sua homóloga para código fonte.

Seleção das variáveis. Esse estudo procura verificar se, quando o conjunto de classes detectadas pelas estratégias de detecção para modelo varia, o conjunto de classes detectadas pelas correspondentes estratégias para código acompanha essa variação. A variável dependente corresponde, portanto, às classes identificadas como suspeitas pelas estratégias de detecção para código. A variável independente corresponde às classes identificadas como suspeitas pelas estratégias de detecção correspondentes para modelos.

Desenho. Não houve uma randomização na associação entre os sistemas e participantes envolvidos no estudo. Um único estudante de mestrado foi responsável pela aplicação das estratégias de detecção no modelo e no código dos nove sistemas e pela análise dos resultados obtidos. Um principal fator que pode afetar os resultados do estudo é o grau de correspondência entre os diagramas e o código. Procurou-se evitar grandes discrepâncias entre os diagramas e o código.

Para isto, alguns dos diagramas de classes dos sistemas analisados foram gerados pelos próprios desenvolvedores como parte do processo de desenvolvimento e os diagramas de classes dos sistemas restantes foram gerados com o apoio da ferramenta Enterprise Architecture (EA, 2008).

6.1.2.3. Operação

A operação do estudo é composta por diferentes etapas: (i) seleção dos sistemas a serem utilizados; (ii) geração dos diagramas de classes correspondentes; (iii) aplicação das estratégias de detecção nos diagramas de classes usando a ferramenta QCDTool (Capítulo 5); (iv) aplicação das estratégias de detecção no código utilizando as ferramentas *InCode* (2008) e *Together* (2006); (v) análise dos resultados por meio da aplicação de teste estatístico.

As duas primeiras fases envolveram os processos de seleção dos sistemas a serem analisados e a geração dos modelos correspondentes. Procurou-se por sistemas com tamanhos e domínios distintos, desenvolvidos por pessoas e em ambientes diferentes, e cuja documentação e diagramas UML estivessem disponíveis. Chegou-se então aos nove sistemas (S1-S9) apresentados na Seção 6.1.1. Como o sistema JHotdraw foi desenvolvido de acordo com o princípio *opensource*, os diagramas de classes correspondentes às versões utilizadas tiveram que ser gerados mediante engenharia reversa. A geração destes diagramas foi apoiada pela ferramenta Enterprise Architecture (EA, 2008). Além disso, algumas das relações entre as classes foram geradas manualmente pelo estudante de mestrado. Visando evitar fadiga e desinteresse, o estudante dedicou a essa tarefa aproximadamente duas horas por dia durante o período de trinta dias.

Após todos os diagramas de classes estarem disponíveis passou-se a aplicar as estratégias de detecção definidas para modelo. Simultaneamente, foram aplicadas as estratégias de detecção correspondentes para código com as ferramentas *InCode* e *Together*. A primeira ferramenta foi escolhida pelo fato de ser desenvolvida pelo mesmo grupo que definiu as estratégias de detecção para código utilizadas neste trabalho. Portanto, deduzimos que ela seria mais confiável que qualquer outra ferramenta que automatizasse também estas estratégias de detecção. Porém, ela não oferece suporte para automatizar a aplicação de todas as

estratégias envolvidas neste estudo. Ela automatiza apenas as estratégias *Data Class* e *God Class*. Conseqüentemente, foi necessária a utilização de outra ferramenta para complementar a *InCode*. A ferramenta escolhida para esta tarefa foi o *Together*, que, além de permitir a aplicação de diferentes estratégias de detecção, apóia a aplicação de um conjunto grande de métricas. As estratégias de detecção automatizadas com apoio desta ferramenta foram: *Shotgun Surgery*, *Misplaced Class* e *God Package*. Como ainda não existe uma ferramenta que dê suporte a aplicação da estratégia de detecção *Long Parameter List* no código, todos os resultados desta estratégia no código foram colhidos manualmente.

Finalmente, com os resultados coletados da aplicação das estratégias de detecção no modelo e no código passou-se à fase de análise com base na correlação e nas medidas de acurácia, precisão e *recall*.

6.1.3. Resultados

Esta seção apresenta os resultados da aplicação das estratégias de detecção *Data Class*, *God Class*, *Shotgun Surgery*, *Misplaced Class*, *God Package e Long Parameter List*.

As medidas de acurácia, precisão e *recall* são definidas da seguinte forma: a acurácia representa o grau com o qual as estratégias de detecção para modelo classificaram as entidades no *design* de forma semelhante às suas homólogas para código fonte. A precisão corresponde à proporção entre a quantidade de entidades de *design* classificadas como problemáticas tanto pelas estratégias de detecção para modelo como pelas correspondentes para código e o total de entidades classificadas como problemáticas no modelo, e o *recall* representa a proporção entre a quantidade de entidades de *design* classificadas como problemáticas tanto pelas estratégias de detecção para modelo como pelas correspondentes para código e o total de entidades classificadas como problemáticas no código.

Para o cálculo destas medidas foi necessário categorizar as entidades classificadas pelas estratégias em cada sistema da seguinte forma: Falsos Positivos (FP), incluem todos os suspeitos revelados no modelo que não foram detectados no código; Falsos Negativos (FN), referem-se a todas as entidades que não foram classificadas como suspeitas no modelo, porém, foram identificadas no código;

Verdadeiros Positivos (VP), incluem todos os suspeitos no modelo que foram detectados também no código; e Verdadeiros Negativos (VN), referem-se às entidades não identificadas no modelo nem no código.

Com base nesta classificação, estas medidas foram calculadas para cada estratégia de detecção em cada sistema são calculadas, respectivamente, pelas seguintes fórmulas:

$$Acurácia = \frac{VP + VN}{VP + VN + FP + FN}$$

$$Precisão = \frac{VP}{VP + FP}$$

$$Recall = \frac{VP}{VP + FN}$$

Para as hipóteses (H_0 e H_1) relativas à correlação pretende-se determinar se as estratégias de detecção para o modelo possuem a habilidade de predizer os resultados obtidos da execução das estratégias de detecção para código. Para isto, foi utilizado o teste estatístico de coeficiente de correlação. Os dados utilizados no teste correspondem às entidades classificadas como verdadeiros positivos. Por meio da aplicação do teste Kolmogorov-Smirnov foi verificado que os dados obtidos não tinham distribuição normal. Portanto, utilizou-se o teste não parametrizado "Coeficiente de Spearman" para determinar o coeficiente de correlação entre os resultados dos dois tipos de estratégias. O teste foi aplicado com um nível de significância de α=0,05, o que significa que o resultado é tratado com um grau de confiança de 95%. Para uma amostra de tamanho nove e α=0,05, o coeficiente de correlação definido é de 0,68.

Nas próximas seções, são apresentados e discutidos os resultados obtidos para as estratégias tratadas. O Anexo B apresenta todos os dados obtidos com a aplicação destas estratégias de detecção.

6.1.3.1. Resultados para a Estratégia de Detecção *Data Class*

O valor do coeficiente de correlação, calculado utilizando o número total de classes detectadas no modelo e no código (colunas 2 e 3 da Tabela 7), foi de 0,76. Como ele é maior que o coeficiente de aceitação estimado (6,8- Seção 6.1.3), a hipótese nula H_0 referente à não existência de correlação entre a estratégia *Data*

Class proposta para modelo e sua homóloga para código fonte foi rejeitada. Pode-se concluir, então, que existe uma correlação positiva significativa entre a estratégia de detecção *Data Class* proposta para modelo e a definida para código.

A estratégia para modelo alcançou para todos os sistemas avaliados graus de acurácia superior a 95%. Além disso, apresentou para a maioria dos sistemas grau de precisão superior a 80%. Porém, existem sistemas para os que o grau foi inferior a 70%. Isto aconteceu em virtude desses sistemas apresentarem alta porcentagem de falsos positivos em relação ao total de classes classificadas como problemáticas ao aplicar a estratégia para código, Tabela 7. Finalmente, a estratégia de detecção *Data Class* para modelo alcançou, em todos os sistemas avaliados, 100% de *recall*, ou seja, todas as entidades classificadas como problemáticas pela estratégia para modelo, foram também classificadas como problemáticas pela estratégia homóloga para código.

Tabela 7 – Resultados da estratégia de detecção *Data Class*.

Sistemas	Total de suspeitos no modelo	Total de suspeitos no código	FP	FN	Acurácia	Precisão	Recall
S1	12	10	2	0	96%	83%	100%
S2	5	4	1	0	99%	80%	100%
S3	16	14	2	0	98%	88%	100%
S4	15	13	2	0	97%	88%	100%
S5	14	11	3	0	98%	78%	100%
S6	2	1	1	0	99%	50%	100%
S7	5	3	2	0	99%	60%	100%
S8	8	6	2	0	99%	75%	100%
S9	14	11	3	0	99%	78%	100%

De maneira geral, os resultados obtidos por esta estratégia apresentaram falsos positivos e nenhum falso negativo. Os casos de falsos positivos ocorreram porque as métricas da versão para modelos foram adaptadas de forma a usar menos informação do que as correspondentes da versão para código. Por exemplo, uma das cláusulas dessa estratégia diz que, para ser uma *Data Class*, uma classe tem que ter NOPA (*Number Of Public Attributes*)+NOAM (*Number Of Access Methods*) > 4 (Seção 4.2.1). A métrica NOAM conta o número de métodos de acesso de uma classe. Dentre outras coisas, essa métrica usa a complexidade do método para decidir se ele é de acesso ou não. Se a complexidade do método é alta, ele não é considerado como de acesso. No código, a complexidade do

método é sua complexidade ciclomática. No entanto, como no modelo não há informações para cálculo da complexidade ciclomática, a complexidade de todos os métodos foi fixada com o valor um ($NOAM_{adpt}$). Por causa disso, alguns métodos foram classificados no modelo como sendo métodos de acesso, mas não o foram no código. Isso fez com que algumas classes tivessem valor maior para NOAM no modelo, e conseqüentemente, fossem classificadas como *Data Class* no modelo e não no código. Situações semelhantes ocorreram também com relação à métrica WOC_{adpt} (Seção 4.2.1).

Essa forma de adaptação da estratégia *Data Class* também explica a ausência de falsos negativos. Além disso, a própria existência de menos informação nos diagramas de classes contribuiu para falsos positivos. Por exemplo, no código, a computação de NOAM para classes de negócio não leva em conta alguns métodos por eles serem herdados de classes de APIs. Porém, as classes de APIs não foram consideradas nos modelos. Por isso, a computação de NOAM nos modelos não sabe que tais métodos foram herdados. Isso contribuiu para diferenças nos valores dessa métrica, e, conseqüentemente, falsos positivos.

6.1.3.2. Resultados para a Estratégia de Detecção *God Class*

O valor do coeficiente de correlação, baseado nas classes detectadas no modelo e no código (colunas 2 e 3 da Tabela 8), foi de 0,84. Como o valor obtido é maior que o coeficiente de aceitação estimado, a hipótese nula H_0 de correlação entre a estratégia *God Class* proposta para modelo e sua homóloga para código fonte foi rejeitada. Pode-se concluir, então, que existe uma correlação positiva significativa entre a estratégia de detecção *God Class* proposta para modelo e a definida para código.

A estratégia de detecção *God Class* para modelo classificou as classes dos nove sistemas envolvidos no estudo com alto grau de acerto, alcançando para todos eles, graus de acurácia superior a 96% (Tabela 7). Por outro lado, a estratégia apresentou para a maioria dos sistemas avaliados baixos graus de precisão e *recall*. Isto aconteceu porque para os sistemas S2, S3, S4, S5, S6, S7 e S8 a estratégia de detecção para modelo não identificou as mesmas entidades que sua homóloga para código.

Tabela 8 – Resultados para a estratégia de detecção *God Class*.

Sistemas	Total de suspeitos no modelo	Total de suspeitos no código	FP	FN	Acurácia	Precisão	Recall
S1	0	0	0	0	100%	100%	100%
S2	0	1	0	1	99%	0%	0%
S3	3	0	3	0	97%	0%	0%
S4	0	0	0	0	100%	100%	100%
S5	1	2	0	1	99%	0%	0%
S6	3	1	2	0	98%	33%	100%
S7	3	1	2	0	99%	50%	100%
S8	5	8	1	3	98%	80%	57%
S9	5	3	3	1	99%	40%	66%

A seguir é apresentada uma análise sobre as causas dos baixos valores de precisão e *recall* para a estratégia de detecção *God Class* para modelo.

Falsos positivos. Os falsos positivos para a estratégia de *God Class* foram causados pelo uso, na estratégia para modelos, da métrica de coesão CAM no lugar da métrica TCC, usada na estratégia para código. Algumas classes do modelo tiveram valores baixos para a métrica CAM (*Cohesion Among Methods*), enquanto TCC (*Tight Class Cohesion*) apresentou valores altos para as mesmas classes no código (Seção 4.2.2). Isso levou algumas dessas classes a serem classificadas como *God Class* no modelo e não no código, uma vez que uma das cláusulas dessa estratégia diz que para ser *God Class* uma classe tem que ter coesão menor que 0,33. Note que a diferença na forma de calcular coesão dessas duas métricas poderia também levar a casos em que uma mesma classe tivesse valor alto de CAM e valor baixo de TCC. Isso poderia contribuir para a ocorrência de falsos negativos. Porém, não houve exemplos desse caso nos sistemas avaliados nesse estudo.

Falsos negativos. Os casos de falsos negativos para a estratégia de *God Class* foram causados pela adaptação das métricas WMC (*Weighted Method per Class*) e ATFD (*Access To Foreign Data*). Uma das cláusulas dessa estratégia diz que, para ser *God Class*, uma classe tem que ter `WMC > 47` (Seção 4.2.2). WMC mede a soma das complexidades de todos os métodos de uma classe. Novamente, as diferenças ocorreram porque no código a complexidade de um método corresponde a sua complexidade ciclomática, enquanto no modelo a

complexidade de cada método foi fixada em um. Houve casos de classes com poucos métodos com alta complexidade ciclomática. Essas classes apresentaram valor alto de WMC no código, mas valor baixo no modelo, gerando, então, falsos negativos. Em relação à métrica ATFD, a estratégia diz que para ser *God Class* uma classe deve ter `ATFD > 4` (Seção 4.2.2). No modelo, ATFD conta apenas o número de classes acessadas e não o número de atributos acessados, como é feito no código. Por isso, algumas classes apresentaram valores altos de ATFD no código e baixos no modelo, o que contribuiu para alguns casos de falsos negativos.

6.1.3.3. Resultados para a Estratégia de Detecção *Shotgun Surgery*

A Tabela 9 apresenta os resultados da estratégia de detecção *Shotgun Surgery* para cada um dos sistemas avaliados. O valor do fator de correlação calculado para esta estratégia, utilizando o número total de classes detectadas no modelo e no código, foi de 0,14. Como esse valor é bem perto de zero não se pode concluir que existe ou não correlação entre as duas versões tratadas desta estratégia.

Tabela 9 – Resultados para a estratégia de detecção *Shotgun Surgery*.

Sistemas	Total de suspeitos no modelo	Total de suspeitos no código	FP	FN	Acurácia	Precisão	Recall
S1	3	5	0	2	96%	100%	66%
S2	2	0	2	0	98%	0%	100%
S3	5	0	5	0	95%	0%	100%
S4	4	0	4	0	95%	0%	100%
S5	3	0	3	0	98%	0%	100%
S6	3	1	2	0	98%	33%	100%
S7	28	31	0	4	98%	100%	87%
S8	29	33	0	5	98%	100%	85%
S9	20	24	0	4	99%	100%	83%

Embora a estratégia não tenha correlação e apresentara para todos os sistemas casos de falsos positivos ou falsos negativos, o grau de acurácia para todos os sistemas foi superior a 95%. Isto ocorreu porque no teste da correlação são considerados, apenas, os verdadeiros positivos, ou seja, a quantidade de entidades classificadas como problemáticas tanto no modelo como no código.

Porém, no cálculo da acurácia é considerada a quantidade total de entidades classificadas no modelo de forma semelhante ao código, ou seja, tanto verdadeiros positivos como verdadeiros negativos. No caso da precisão, para quatro dos sistemas avaliados (S1, S7, S8 e S9) a estratégia alcançou índices de 100%. Porém, nos restantes sistemas, o índice da precisão foi menos que 35%. Finalmente para o *recall* a estratégia registrou para cinco sistemas (S2-S6) o grau de 100%. Porém, por causa de falsos negativos, nos restantes sistemas, os graus foram menores de 88%, sendo que para um destes sistemas o grau foi de 66%.

Falsos Positivos. Uma das cláusulas dessa estratégia diz que para ser uma *Shotgun Surgery* uma classe tem que ter CM (*Changing Methods*) > 10 (Seção 4.2.3). As diferenças ocorreram porque, no código, os métodos que poderiam ser afetados correspondem a aqueles que fazem chamadas aos métodos definidos na classe avaliada. No modelo, foram utilizadas as referências à classe avaliada feitas por meio de parâmetros e atributos. Porém, houve casos em que algumas das instâncias da classe avaliada foram utilizadas apenas na criação de outros objetos e não para realizar chamadas a alguns de seus métodos. Algumas classes no modelo tiveram valores altos da soma CM_{adpt}+CA (*Changing Attribute* – Seção 4.2.3), enquanto CM apresentou valores baixos para as mesmas classes no código. Isso levou algumas dessas classes a serem classificadas como *Shotgun Surgery* no modelo e não no código. A forma de determinar os métodos afetados influenciou na computação da métrica CC (*Changing Class* – Seção 4.2.3), e levou, portanto, a que classes no modelo tivessem valores altos da métrica CC_{adpt}, enquanto CC apresentou valores baixos para as mesmas classes no código.

Falsos Negativos. As mesmas métricas mencionadas anteriormente causaram a presença de falsos negativos para a estratégia *Shotgun Surgery*. No caso da métrica CM, no código, existem referências à classe avaliada feitas por meio da declaração de variáveis locais, chamadas a métodos estáticos. Já no modelo, estas informações não estão disponíveis. Algumas classes no código tiveram altos valores para a métrica CM, enquanto que no modelo a soma CM_{adpt}+CA apresentou baixos valores para as mesmas classes. Isso levou algumas dessas classes a serem classificadas como *Shotgun Surgery* no código e não no modelo. Em relação à métrica CC, o fato de no modelo não existir informações sobre detalhes de implementação fez com que algumas relações de dependência e/ou associação não fossem modeladas. Isso fez com que a que a métrica CC

apresentasse altos valores para código, enquanto no modelo a métrica CC_{adpt} tive baixos valores. Esta situação também foi causada devido à forma de computar as métricas CM e CM_{adpt}.

6.1.3.4. Resultados para a Estratégia de Detecção *God Package*

Os resultados da aplicação da estratégia de detecção *God Package* estão resumidos na Tabela 10. O valor do coeficiente de correlação, utilizando o total de pacotes detectados no modelo e no código, foi de 0,95. Como ele é maior que o coeficiente de aceitação estimado (6,8- Seção 6.1.3), a hipótese nula $H_{0,}$ que diz que não existe correlação entre a estratégia *God Package* proposta para modelo e sua homóloga para código fonte, foi rejeitada. Pode-se concluir, então, que existe uma correlação positiva significativa entre a estratégia de detecção *God Package* proposta para modelo e a definida para código.

A estratégia de detecção *God Package* para modelo alcançou graus de acurácia acima de 89% em todos os sistemas avaliados. Por tanto, pode-se dizer que a estratégia classificou os pacotes de maneira bem semelhante que sua estratégia correspondente para código. Em relação à precisão, a mesma estratégia apresentou graus de 100% de precisão para todos os sistemas avaliados. Ou seja, todos os pacotes detectados pela estratégia para modelo, foram também detectados pela estratégia homóloga para código. Embora a estratégia apresente para a maioria dos sistemas 100% de *recall*, dois dos nove sistemas avaliados apresentaram graus inferiores aos 55%. Para esse dois sistemas (S5 e S8), a estratégia de detecção para modelo apresentou altos valores de falsos negativos em relação ao total de pacotes identificados pela estratégia para código.

Tabela 10 – Resultados para a estratégia de detecção *God Package*.

Sistemas	Total de suspeitos no modelo	Total de suspeitos no código	FP	FN	Acurácia	Precisão	Recall
S1	0	0	0	0	100%	100%	100%
S2	0	0	0	0	100%	100%	100%
S3	1	1	0	0	100%	100%	100%
S4	1	1	0	0	100%	100%	100%
S5	0	1	0	1	93%	100%	50%
S6	1	1	0	0	100%	100%	100%
S7	2	2	0	0	100%	100%	100%
S8	1	3	0	2	90%	100%	33%
S9	0	0	0	0	100%	100%	100%

Os casos de falsos negativos na estratégia de detecção *God Package* ocorreram porque as métricas para modelos foram adaptadas de a forma a usar menos informações que as correspondentes da versão para código. Por exemplo, uma das cláusulas dessa estratégia diz que, para ser um *God Package*, um pacote tem que ter NOCC (*Number Of Client Class*) > 15 (Seção 4.3.1). A métrica NOCC conta o número de classes clientes em outros pacotes que uma classe possui. Dentre outras coisas, essa métrica utiliza os acessos a informações de outras classes para determinar a quantidade de clientes que uma classe possui. No código, estes acessos são determinados por chamadas a métodos e acessos a atributos de outras classes. No entanto, como no modelo não há informações para detectar estes acessos, as classes clientes foram detectadas utilizando apenas as referências a outras classes realizadas nas declarações de atributos e parâmetros, além de associações, herança e dependências entre classes ($NOCC_{adpt}$). Por causa disto, algumas classes foram identificadas como clientes no código, mas não o foram no modelo. Isso fez com que alguns pacotes tivessem maior valor para NOCC no código, e conseqüentemente, fossem classificados como *God Package* no código e não no modelo. Situações semelhantes ocorreram também com relação à métrica $NOCP_{adpt}$ (Seção 4.3.1). No caso da métrica PS como as informações utilizadas por ela estão disponíveis tanto no código como no modelo, ela apresentou sempre os mesmos valores para as duas versões e, portanto, não influenciou nos falsos negativos. Essa forma de adaptação da estratégia *God Package* também explica a ausência de falsos positivos.

6.1.3.5. Resultados para a Estratégia de Detecção *Misplaced Class*

Apenas três dos sistemas avaliados apresentaram este problema de *design*. O coeficiente de correlação calculado, utilizando o total de classes identificadas pelas estratégias (colunas 2 e 3 da Tabela 11), foi de 0,75. Com este resultado a hipótese nula correspondente à correlação é rejeitada e, portanto, pode-se afirmar que os resultados obtidos por meio da aplicação da estratégia de detecção proposta para modelo possuem correlação com os resultados gerados pela estratégia correspondente para código. A estratégia apresentou grau de acurácia acima de 95% para todos os sistemas avaliados. Porém, houve sistemas em que os graus de precisão e *recall* estiveram baixos de 55%. Estes resultados aconteceram porque a estratégia apresentou para alguns dos sistemas casos de falsos positivos e falsos negativos (Tabela 11).

Tabela 11 – Resultados para a estratégia de detecção *Misplaced Class*.

Sistemas	Total de suspeitos no modelo	Total de suspeitos no código	FP	FN	Acurácia	Precisão	Recall
S1	1	3	0	2	96%	33%	100%
S2	0	0	0	0	100%	100%	100%
S3	0	0	0	0	100%	100%	100%
S4	0	0	0	0	100%	100%	100%
S5	0	0	0	0	100%	100%	100%
S6	0	0	0	0	100%	100%	100%
S7	5	10	1	5	97%	50%	50%
S8	1	1	0	0	100%	100%	100%
S9	0	0	0	0	100%	100%	100%

Falsos Positivos. O caso de falso positivo no sistema S7 aconteceu porque a métrica CL (*Class Locality* – Seção 4.3.2) foi adaptada de forma a usar menos informação que a correspondente para código. Uma das cláusulas desta estratégia diz que, para ser uma *Misplaced Class* uma classe tem que ter CL < 0.33. Esta métrica representa o grau de dependência que uma classe tem em seu próprio pacote em relação ao total de dependências que ela possui. No código, de maneira semelhante às outras métricas que realizam contagem de dependências entre classes, a métrica CL utiliza informações disponíveis no código interno dos métodos. Por causa disto, algumas classes tivessem menor valor para CL no

modelo, e conseqüentemente fossem classificadas como *Misplaced Class* no modelo e não no código.

Falsos Negativos. Duas das cláusulas da estratégia dizem que, para ser uma *Misplaced Class* uma classe tem que ter NOED (*Number Of External Dependencies*) > 6 e DD (*Dependency Dispersion*) > 3 (Seção 4.3.2). De maneira semelhante ao caso de falso positivo, as métricas NOED e DD, por realizarem contagem de dependências, apresentaram maiores valores no código que no modelo. Isso fez com que algumas classes fossem classificadas como *Misplaced Class* no código e não modelo, o que contribuiu para alguns casos de falsos negativos.

6.1.3.6. Resultados para a Estratégia de Detecção *Long Parameter List*

O valor do coeficiente de correlação, calculado utilizando os resultados apresentados na Tabela 12, foi de 0,91. Este valor é maior que o coeficiente de aceitação estimado (0,68 – Seção 6.1.3). Pode-se, portanto, rejeitar a hipótese nula correspondente à correlação e concluir que existe uma correlação positiva significativa entre a estratégia *Long Parameter List* proposta para modelo e a estratégia para código aplicada manualmente.

A estratégia para as restantes medidas avaliadas (acurácia, precisão e *recall*) apresentou graus acima de 90%. Estes resultados foram motivados pelo fato de que este problema não ocorreu em cinco dos sistemas. Além disso, nos sistemas restantes, ocorreram poucos falsos positivos e nenhum falso negativo.

Tabela 12 – Resultados para a estratégia de detecção de *Long Parameter List*.

Sistemas	Total de suspeitos no modelo	Total de suspeitos no código	FP	FN	Acurácia	Precisão	Recall
S1	0	0	0	0	100%	100%	100%
S2	0	0	0	0	100%	100%	100%
S3	0	0	0	0	100%	100%	100%
S4	0	0	0	0	100%	100%	100%
S5	0	0	0	0	100%	100%	100%
S6	4	4	0	0	100%	100%	100%
S7	11	10	1	0	99%	90%	100%
S8	16	14	2	0	99%	87%	100%
S9	29	27	2	0	99%	93%	100%

Os casos de falsos positivos aconteceram porque a estratégia de detecção para modelo usa menos informação que o mecanismo de inspeção manual utilizado para identificar este problema de *design* no código. Por exemplo, no modelo, o uso da métrica NOP (*Number Of Parameters*) assume que os parâmetros dos métodos possuem relação entre si (Seção 4.4.1). Porém, no código existem informações para determinar se os parâmetros dos métodos estão ou não inter-relacionados. Houve casos em que parâmetros foram identificados como inter-relacionados no modelo e que, portanto, poderiam ser encapsulados, enquanto no código esses parâmetros não possuíam relação. O fato da informação representada por esses parâmetros não ser útil para outros métodos fez com que os desenvolvedores não encapsulassem em uma estrutura a fim de reutilizá-la. Além disso, os métodos herdados de classes das APIs não foram considerados no código. Porém, estas classes não foram representadas no modelo e, portanto, os métodos foram considerados no modelo. Conseqüentemente, essas diferenças contribuíram para casos de falsos positivos e motivaram a ausência de falsos negativos.

6.1.4. Discussões sobre os Resultados

Os resultados do estudo mostraram que, a princípio, as estratégias de detecção *Data Class*, *God Class* e *God Package* estão correlacionadas significativamente com suas correspondentes versões para código. Isto permite que técnicas de reestruturação de *design* possam ser aplicadas desde a etapa de modelagem eliminando retrabalho em outras etapas futuras.

As estratégias de detecção *Long Parameter List* e *Data Class* apresentaram os melhores graus de *recall* em relação a todas as estratégias utilizadas (100% para todos os sistemas avaliados). Isto significa que todas as entidades que tiveram este problema de *design* no código foram detectadas pela estratégia proposta para o modelo. Porém, algumas das entidades identificadas como problemáticas no modelo não foram identificadas como tal no código, o que pode acarretar que elas sejam tratadas como instâncias destes problemas no modelo sem serem realmente problemáticas.

Por outro lado, a estratégia de detecção *God Package* apresentou a melhor precisão para todas as estratégias analisadas (100% para todos os sistemas avaliados). Ou seja, todos os pacotes classificados como instâncias *God Package* no modelo foram também detectados no código. Isto permite aos usuários melhorar diretamente o problema no código já que existe uma grande probabilidade que todos os pacotes detectados como problemáticos no modelo sejam posteriormente detectados no código.

O alto grau de acurácia da estratégia *God Class* mostrou que as classes grandes no modelo possuem alta probabilidade de utilizar muita informação referente a outras classes e ter pouca interdependência entre seus métodos no código. Isto é apoiado pelo fato de que as classes com maiores valores da métrica WMC no modelo, apresentaram baixo valor na métrica TCC e alto em ATFD no código, acarretando que fossem classificadas como *God Class* pela estratégia para código.

A estratégia de detecção *Shotgun Surgery* apresentou melhores resultados de precisão nos sistemas com maior quantidade de classes. Nestes sistemas evidenciou-se um aumento dos valores das métricas CC, CM em relação aos sistemas restantes. Isso fez com que a ocorrência deste problema de *design* no código fosse maior e, portanto, a quantidade de falsos positivos menor.

Finalmente, embora poucos dos sistemas avaliados tenham apresentado problemas de *Misplaced Class* no código, a estratégia apresentou apenas um caso de falso positivo. Isto faz com que a probabilidade das classes identificadas como suspeitas no modelo serem problemáticas no código é alta, oferecendo uma alta confiabilidade aos usuários.

6.2. Avaliação do modelo da qualidade QMOOD

Esta seção descreve o segundo estudo experimental. Este estudo tem como objetivo avaliar a qualidade do *design* de um conjunto de sistemas com base na análise de atributos da qualidade. Para isto, foi utilizado o modelo da qualidade QMOOD (Seção 2.3.1.1), automatizado pela ferramenta QCDTool (Capítulo 5). Nesse estudo, foi avaliado o *design* das quatro versões do framework JHotdraw: 5.2, 6.0, 7.0 e 7.1. Procurou-se verificar se a aplicação do QMOOD nos diagramas

de classes dos sistemas era capaz de identificar alterações nos níveis de qualidade do *design* devido às mudanças realizadas ao longo das quatro versões.

Os atributos da qualidade escolhidos para serem avaliados são: facilidade de compreensão, flexibilidade e reusabilidade. Estes atributos formam parte do conjunto de atributos da qualidade quantificados pelo modelo QMOOD.

6.2.1. O Formato do Estudo

Seguindo a estrutura do padrão GQM (Vasili e Weiss, 1984; Basili e Rombach, 1988; Van Solingen e Berghout, 1999), os objetivos deste estudo são:

Analisar: o *design* das versões 5.2, 6.0, 7.0 e 7.1 do framework JHotdraw

Com o propósito de: avaliar os atributos da qualidade de reusabilidade, flexibilidade e facilidade de compreensão definidos pelo modelo QMOOD

Com respeito a: identificação de alterações nos valores desses atributos ao longo das versões

Do ponto de vista: do pesquisador

No contexto de: estudantes da pós-graduação do Laboratório de Engenharia de Software da PUC-Rio.

Diferentemente do primeiro estudo experimental, neste estudo não foi necessária uma fase para a geração dos diagramas UML mediante engenharia reversa, porque os diagramas utilizados foram reutilizados do primeiro estudo. Os diagramas utilizados neste estudo correspondem aos diagramas das versões 5.2, 6.0, 7.0 e 7.1 do framework JHotdraw usados no primeiro estudo experimental. Tendo os diagramas gerados, foi aplicado o modelo da qualidade QMOOD para quantificar os atributos facilidade de compreensão, flexibilidade e reusabilidade em cada *design* a ser avaliado. A aplicação do modelo QMOOD nos diagramas foi feita com o uso da ferramenta QCDTool (Capítulo 5). Finalmente foram analisadas, utilizando os diagramas de classes e o código fonte associado, as causas das variações nos valores dos atributos da qualidade identificadas por QMOOD ao longo das versões. A quantificação dos atributos foi realizada com base nas equações propostas em (Bansiya & Davis, 2002).

*Reusabilidade := -0.25*acoplamento + 2.5*coesão+0.5*trocademensagens +0.25*tamanho*

*Flexibilidade := 0.25*encapsulamento - 0.25*coesão + 0.5*composição + 0.5* polimorfismo*

*Facilidade de Compreensão := -0,33*Abstração + 0,33*Encapsulamento-0,33*Acoplamento + 0,33*Coesão - 0,33*Polimorfismo - 0,33*Complexidade-0,33*Tamanho*

Os coeficientes das equações são baseados em estudos experimentais realizados pelos pesquisadores que definiram o QMOOD como parte do processo de validação desse modelo. Porém, elas podem ser modificadas procurando melhores resultados destes atributos em correspondência ao domínio do sistema que está sendo analisado.

6.2.2. Resultados das Propriedades do *Design* e Atributos da Qualidade Avaliados

Esta seção analisa os resultados obtidos para as propriedades que compõem as equações do modelo QMOOD e atributos da qualidade avaliados no estudo. Os valores para essas propriedades são apresentados na Tabela 13.

Tabela 13 – Valores das propriedades de *design*.

Propriedade	Hotdraw 5.2	JHotdraw 6.0	JHotdraw 7.0	JHotdraw 7.1
Abstração	1,8	1,8	1,73	1,8
Acoplamento	3,4	3,8	2,97	2,9
Coesão	0,1	0,1	0,1	0,1
Complexidade	7,8	8,4	9,3	8,8
Composição	0,4	0,49	0,51	0,56
Encapsulamento	0,95	0,96	0,93	0,90
Polimorfismo	0,15	0,15	0,17	0,17
Tamanho	172	313	332	401
Troca de Mensagens	7,8	8,48	9,5	9,27

Como diferentes tipos de métricas (percentuais, absolutas) são combinados na computação dos atributos da qualidade, esses valores foram normalizados com relação aos valores das métricas da versão 5.2. A Tabela 14 apresenta os valores normalizados destas propriedades (calculados dividindo o valor de cada métrica pelo valor da métrica correspondente na primeira versão).

Tabela 14 – Valores normalizados das propriedades de *design*.

Propriedade	Hotdraw 5.2	JHotdraw 6.0	JHotdraw 7.0	JHotdraw 7.1
Abstração	1	1	0,96	0,96
Acoplamento	1	1,11	0,87	0,85
Coesão	1	1	1	1
Complexidade	1	1,07	1,19	1,13
Composição	1	1,22	1,25	1,4
Encapsulamento	1	1,01	0,97	0,94
Polimorfismo	1	1	1,13	1,13
Tamanho	1	1,8	1,93	2,33
Troca de mensagens	1	1,08	1,2	1,18

Esta normalização é valida, pois ela é baseada em diferentes versões de um mesmo sistema. Contudo, esse mecanismo não deve ser utilizado em casos nos quais sistemas diferentes influenciem na normalização.

Acoplamento. Embora a quantidade de funcionalidades oferecidas pelo sistema tenha aumentado entre as diferentes versões, o grau de acoplamento entre as classes diminuiu de uma versão para a seguinte. Isto aconteceu pelos seguintes motivos: (i) as classes existentes das versões anteriores não foram muito acopladas às novas classes, e (ii) de maneira geral as classes criadas apresentaram grau de acoplamento relativamente baixo.

Troca de Mensagens e Tamanho. A quantidade de mensagens que os objetos podem trocar aumentou de uma versão à seguinte. Isso ocorreu porque muitos métodos e classes foram criados com o objetivo de oferecer mais funcionalidades. Por exemplo, a versão 7.0, diferentemente da versão 6.0, define um framework para estruturar os editores gráficos. Este framework pode ser utilizado para criar editores gráficos animados e interativos. Além disso, atributos das classes foram encapsulados e passaram a ser acessados por meio de métodos.

Coesão. Ao longo das versões foi mantido o mesmo valor de coesão. Este valor é baixo devido à métrica de coesão utilizada ser baseada na sintaxe dos métodos de uma classe. Sendo assim, apenas foram analisados os tipos dos parâmetros dos métodos. Muitas classes apresentaram valores quase nulos de coesão. Nestas classes, a maioria dos parâmetros faz referência a classes definidas no escopo da linguagem Java. Por causa disto, estes parâmetros não participam no cálculo da coesão, acarretando baixos valores para esta propriedade do *design*.

Encapsulamento. Não houve diferenças significativas entre os valores de encapsulamento nas versões analisadas. Além disso, observou-se que a maioria das classes apresentou altos índices de encapsulamento. A principal razão para esses resultados foi o fato dos desenvolvedores ao longo destas versões manterem a estratégia de reduzir o uso de atributos públicos nas classes e passaram a utilizar atributos privados, de modo que a maioria destes atributos foi acessada por meio de métodos.

Abstração. Os valores da abstração ao longo das versões foram mantidos quase constantes. Isto aconteceu pelas seguintes razões. Algumas das classes criadas foram inseridas em hierarquias já existentes. As outras classes fizeram parte das novas hierarquias criadas. Essas hierarquias apresentaram características semelhantes às hierarquias já existentes. Por exemplo, nenhuma das hierarquias nas versões do framework excede 5 níveis de profundidade. Além disso, não existe diferença substancial entre o número total de interfaces utilizadas em uma versão e na seguinte.

Composição. O grau de composição aumentou entre as diferentes versões. Isto ocorreu porque a maioria das classes que representam figuras gráficas possui atributos que referenciam outras classes do sistema e a cada nova versão novas entidades deste tipo foram definidas. Além disso, essas classes foram modeladas utilizando o padrão de projeto *Composite* (Gamma et al., 1995), sendo que, muitas delas são figuras compostas. Também, muitas das classes definidas no framework manipularam as figuras, e, por causa disto, elas estão compostas por outras classes.

Polimorfismo. Não houve diferenças no grau de polimorfismo nas diferentes versões. Isso aconteceu porque, de maneira geral, a maioria das classes incorporadas ao framework foram sempre baseadas em outras classes. Por exemplo, no desenvolvimento de novas figuras e editores gráficos foram reutilizadas muitas classes já existentes.

Complexidade. A quantidade de métodos por classes foi aumentando de uma versão para a outra. Por exemplo, o *design* da versão 5.2 possui 12 classes com mais de trinta métodos, entanto a versão 6.0 tem 20 classes com esta característica. Isto aconteceu porque algumas classes tais passaram a definir novas funcionalidades. Além disso, algumas das classes criadas provocaram o aumento

do nível de complexidade na versão 6.0, pois elas definiram uma alta quantidade de métodos.

A Tabela 15 mostra os resultados da quantificação dos três atributos da qualidade baseado nos valores normalizados das métricas apresentados na Tabela 14.

Tabela 15 – Resultados da quantificação dos atributos da qualidade.

Propriedade	Hotdraw 5.2	JHotdraw 6.0	JHotdraw 7.0	JHotdraw 7.1
Reusabilidade	1	1,10	1,2	1,28
Facilidade de Compreensão	-0,99	-1,31	-1,62	-1,75
Flexibilidade	1	1,11	1,18	1,25

Os resultados da aplicação do modelo QMOOD nos diagramas de classe mostraram que os valores dos atributos da qualidade reusabilidade e flexibilidade aumentaram ao longo das quatro versões da JHotdraw. Isto aconteceu pelo aumento ao longo das versões das propriedades de *design*: troca de mensagens, tamanho, composição e polimorfismo. Além disso, o atributo da qualidade facilidade de compreensão diminuiu. Isto foi motivado pelo fato que novas funcionalidades e características foram inseridas para aumentar a capacidade do sistema por meio da implementação de novas classes e métodos a classes já existentes.

6.2.3. Conclusões do Estudo

Os resultados desse estudo mostraram que o modelo QMOOD pode ser aplicado em diagramas de classes. Além disso, QMOOD foi capaz de identificar as diferenças das propriedades de *design*, e, conseqüentemente, atributos da qualidade entre as diferentes versões analisadas. Os resultados relativos às propriedades tamanho, troca de mensagens e complexidade foram as que apresentaram as maiores diferenças entre uma versão e a seguinte, influenciando os atributos reusabilidade e facilidade de compreensão. A métrica de coesão gerou sempre valores iguais para todas as versões, portanto, ela não contribuiu na identificação de diferenças. As classes criadas ao longo das versões mantiveram propriedades semelhantes às classes já existentes. Isso fez com que não existissem diferenças muito altas entre uma versão e a seguinte.

Por outro lado, o estudo mostrou que a quantificação dos atributos da qualidade pode ser usado de forma complementar à detecção de problemas de *design*, pois um mostra a variação geral da qualidade e o outro identifica os fragmentos de *design* problemáticos. Espera-se que a remoção desses pontos problemáticos melhore a qualidade do *design*.

6.3. Ameaças à Validade dos Estudos Realizados

Uma questão fundamental a respeito dos resultados dos estudos é quão válidos eles são. É importante, portanto, considerar a validade dos resultados já desde a etapa de planejamento para desta maneira garantir que os resultados possam ser considerados válidos para a população analisada. Nos estudos realizados, têm-se diferentes tipos de validade a serem considerados.

Validade de construção é o grau com o qual as variáveis independentes e dependentes medem realmente o que elas se propõem a medir (Briand et al, 1997). No primeiro estudo, a variável independente corresponde às entidades identificadas como suspeitas pelas estratégias de detecção para modelo, as que foram definidas tomando como base as estratégias para código apresentadas em (Marinescu 2002 e Lanza & Marinescu, 2006). A aplicação destas estratégias nos diagramas de classes utilizados foi automatizada pela ferramenta apresentada no Capítulo 5. Por outro lado, a variável dependente corresponde às entidades identificadas como suspeitas pelas estratégias de detecção para código. A identificação destas entidades nos sistemas analisados foi automatizada com apoio das ferramentas *InCode* e *Together*, a primeira desenvolvida pelo mesmo grupo que definiu as estratégias de detecção que ela automatiza e a segunda desenvolvida pela Borland. A validade dos resultados destas ferramentas está fora do escopo desta dissertação, porém pode-se destacar que ambas têm sido utilizadas e referenciadas em outros estudos acadêmicos. Sendo assim, podemos considerar que as ameaças à validade de construção são mínimas, uma vez que a computação das estratégias de detecção tanto para modelos como para código foi totalmente automatizada.

Validade interna é o grau com o qual se pode confiar nas conclusões sobre o efeito das variáveis independentes nas variáveis dependentes (Briand et al,

1997). A principal ameaça à validade interna dos estudos está relacionada ao grau de correspondência entre os diagramas e código. Para minimizar essa ameaça, nos preocupamos com os seguintes pontos: (i) os diagramas dos sistemas S1 a S5 foram gerados pelos próprios desenvolvedores como parte do processo de desenvolvimento e dentro do contexto de disciplinas que avaliavam a correspondência dos modelos com o código, os alunos tiveram bom desempenho nessas disciplinas, (ii) os diagramas de classes dos sistemas S6 a S9 (versões do JHotdraw) foram gerados com o apoio da ferramenta Enterprise Architecture (EA, 2008). Além disso, algumas relações nos diagramas de classes das versões do JHotdraw foram geradas manualmente pelo estudante de mestrado. O estudante possui bons conhecimentos e experiência relativa a modelagem orientada a objetos. Além disso, visando evitar sua fadiga e desinteresse, o estudante dedicou a essa tarefa aproximadamente duas horas por dia durante trinta dias.

Validade externa é o grau com o qual os resultados do estudo podem ser generalizados para a população em estudo e para outros contextos (Briand et al, 1997). O uso de apenas um estudante no estudo, e o tamanho, natureza e complexidade dos sistemas analisados, podem restringir a extrapolação dos resultados obtidos. Todavia, apesar dos resultados não poderem ser diretamente generalizados para desenvolvedores profissionais e sistemas do mundo real, o primeiro estudo permitiu a realização de avaliações iniciais das estratégias de detecção propostas para modelo, que indicaram que estudos mais profundos sobre elas merecem ser realizados.