4
Detecção de Problemas de *Design*

Com o objetivo de antecipar a detecção de problemas de design desde os modelos do sistema, nós adaptamos um conjunto de estratégias de detecção a fim de permitir que elas sejam aplicadas em diagramas UML, mais especificamente a diagramas de classe. A idéia é que estratégias de detecção originalmente definidas com base em informações disponíveis no código fonte, possam ser usadas apenas com informações disponíveis nos diagramas de classes.

Este capítulo descreve as estratégias para código definidas em (Lanza & Marinescu, 2006) e (Marinescu, 2002) e as adaptações a elas realizadas para poder aplicá-las em diagramas de classes. Os problemas detectados por estas estratégias são os seguintes: *bad smells*: *Data Class*, *God Class*, *Shotgun Surgery*, *God Package* e *Misplaced Class*, definidos em (Fowler et al, 1999). Além disso, criamos a estratégia de detecção para o *bad smells Long Parameter List*, definido em (Fowler et al, 1999). As adaptações foram de dois tipos: (i) substituição de métricas baseadas em informações existentes apenas no código fonte por métricas semelhantes baseadas em informações disponíveis em diagramas de classes, e (ii) mudança na forma de computar algumas métricas. Os valores limites foram usados nas estratégias para modelos com base nos valores limites das estratégias para código correspondentes: (i) mesmos valores para as métricas cuja informação necessária para seu cálculo está disponível no modelo em nível de detalhe semelhante ao do código, e (ii) valores limites menores para as métricas cuja informação necessária para seu cálculo está disponível no modelo em menor detalhe que no código. Os valores limites precisam ser mais testados e podem ser modificados no futuro.

Nas próximas seções estes problemas são apresentados (utilizando os critérios de Fowler et al (1999), Marinescu (2002) e Lanza & Marinescu (2006), exceto para o problema de *design Long Parameter List*) e agrupados de acordo com as entidades do *design* a que eles estão relacionados: classes, pacotes e métodos. Depois são descritos os sintomas destes problemas, e finalmente, são

apresentadas as estratégias de detecção para código e modelo correspondentes a cada um deles.

4.1. Características dos Valores Limites

Antes de começar a apresentação das estratégias de detecção, é importante destacar um aspecto que tem influência decisiva na classificação das entidades de *design* feita pelas estratégias de detecção: os valores limites utilizados na parametrização de qualquer estratégia de detecção. Este problema, longe de ser novo, caracteriza intrinsecamente qualquer aproximação baseada em métricas. Na maioria dos casos, a escolha dos valores limites é um processo empírico baseado em experiências prévias (Lorenz & Kidd, 1994).

Os valores limites utilizados nas estratégias de detecção propostas para o modelo são baseados nos valores limites usados nas estratégias correspondentes para código. Esses limites foram definidos em (Lanza & Marinescu, 2006; Marinescu, 2004, 2002) com base em estudos empíricos por eles realizados. Além disso, em alguns casos utilizaram-se valores limites permissivos visando obter um maior número de falsos positivos em vez de não detectar algum problema devido ao uso de um valor limite estrito. Extrapola os objetivos desta dissertação realizar os estudos empíricos necessários para ajustar os valores limite.

4.2. Problemas de *Design* para Classes

As classes são entidades chaves no paradigma de programação OO, já que são elas as responsáveis por descrever a estrutura dos objetos. Por causa disto, muitos dos problemas de *design* OO definidos na literatura estão relacionados com classes. Nesta seção, são apresentadas estratégias de detecção definidas a fim de identificar e localizar alguns dos problemas de *design* para classes.

4.2.1. *Data Class*

Este problema de *design* corresponde a classes que não definem funcionalidade própria (Fowler et al, 1999). Segundo Lanza & Marinescu (2006),

a falta de métodos funcionais nessas classes pode indicar que o comportamento associado aos seus dados não está localizado em um único lugar. Uma instância de uma *Data Class* normalmente possui atributos e métodos de acesso que são utilizados por muitas outras classes e poucos serviços. Porém, são pouco complexas. Além disso, normalmente, estas classes manifestam ausência de abstração dos dados. As classes ou estruturas definidas no escopo de outras classes não são consideradas instâncias de uma *Data Class*. Essas classes ou estruturas são consideradas *Data Class* quando permitem a outras classes manipular seus dados.

Como foi apresentado na Seção 2.1, a boa abstração dos dados é uma das características necessárias para um bom *design* OO. A ausência de abstração tem impacto negativo em diferentes atributos da qualidade. Sendo assim, instâncias de *Data Class* reduzem a manutenibilidade, facilidade de teste e compreensão dos sistemas.

4.2.1.1. Estratégia de Detecção para Código Fonte

Lanza & Marinescu (2006) definiram a estratégia para a detecção de *Data Class* para código com base nas características prováveis dessas classes mencionadas anteriormente. Primeiramente, eles detectam as classes do sistema que oferecem pouca funcionalidade. Depois, o conjunto destas classes é reduzido, selecionando apenas as classes que definem muitos métodos de acesso ou possuem muitos dados na sua interface. A estratégia detecção *Data Class* para código é composta das seguintes cláusulas (Figura 3):

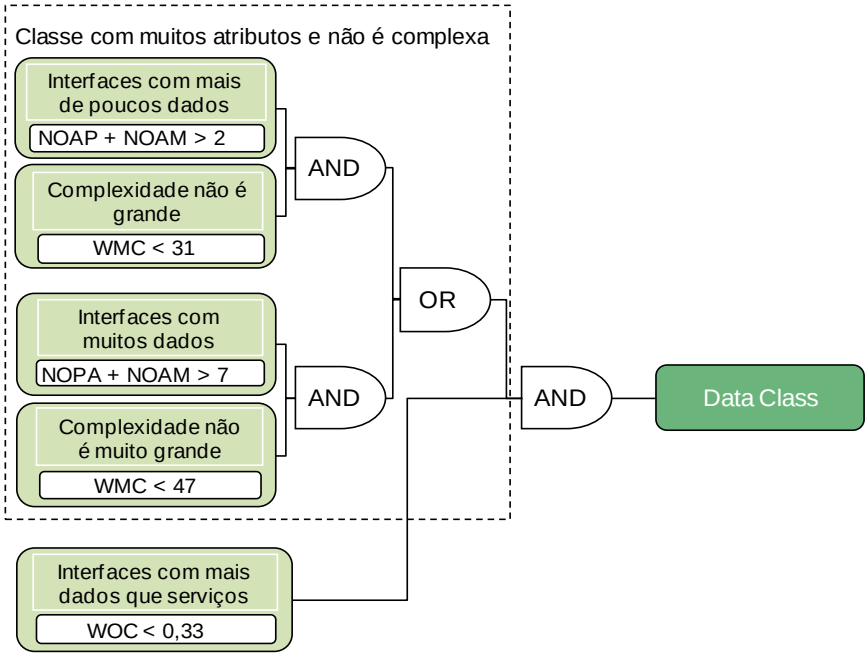

Figura 3 – Estratégia de detecção de *Data Class* para código.

Na Figura 3, a métrica WOC (*Weight Of Class*) conta a porcentagem de métodos correspondentes aos serviços públicos oferecidos pela classe em relação ao total de membros públicos que ela possui. Métodos de serviços são aqueles que realizam algum cálculo ou transformação. A métrica WMC (*Weighted Method per Class*) é a soma das complexidades de todos os métodos de uma classe. Lanza & Marinescu usam a métrica de complexidade ciclomática de McCabe (1976) para calcular a complexidade dos métodos. A métrica NOPA (*Number Of Public Attributes*) conta o número de atributos públicos de uma classe, e a métrica NOAM (*Number Of Access Methods*) conta o número de métodos de acesso não herdados que uma classe possui. Marinescu identifica estes métodos como aqueles cujo nome tem como prefixo '*get*' ou '*set*' e cuja complexidade ciclomática tem valor um. Os detalhes de implementação e as referências originais dessas métricas podem ser encontrados em (Lanza & Marinescu, 2006). A seguir as cláusulas da estratégia de detecção são explicadas.

1. **Interfaces oferecem muitos dados e poucos serviços.** Usando a métrica WOC pode-se decidir se a classe analisada oferece muitos dados e poucos serviços.
2. **Classes com muitos atributos e pouco complexas.** Mediante o uso da métrica WOC pode-se determinar se a interface está formada principalmente por dados ou métodos de acesso. Porém, com o objetivo

de conferir se a quantidade de dados que a classe possui corresponde à complexidade, Lanza & Marinescu (2006) definiram dois possíveis casos:

a) Normalmente as instâncias de *Data Class* clássicas não são classes complexas, mas apresentam muitos dados. Este caso é identificado na estratégia mediante o uso da cláusula que diz que, para ser uma *Data Class*, uma classe tem que ter `(NOPA+NOAM >2)` e `(WMC < 31)`.

b) O outro caso identificado é quando as classes apresentam índices consideráveis de complexidade, mas possuem uma quantidade grande de dados. Estes casos são identificados utilizando a cláusula da estratégia que diz que para que uma classe seja *Data Class* tem que ter `(NOPA+NOAM > 7)` e `(WMC < 47)`.

4.2.1.2. Estratégia de Detecção para Modelo

Na estratégia de detecção *Data Class* para código, três das quatro métricas utilizadas – WOC, NOAM, WMC – dependem de informações disponíveis no código interno dos métodos para: (i) determinar quais são os métodos de acesso, e (ii) calcular a complexidade ciclomática dos métodos. No entanto, cada uma destas métricas pode ser adaptada de modo que utilizem apenas informações disponíveis nos diagramas de classes. Primeiro, ao realizar as adaptações, simplificou-se a identificação dos métodos de acesso para a computação das métricas WOC e NOAM: a complexidade ciclomática não é mais levada em conta. Agora os métodos de acesso são aqueles que têm prefixo "*get*" ou "*set*". Segundo, na computação da métrica WMC, o valor da complexidade de um método é o número de seus parâmetros, e na contagem o valor que o método devolve também é considerado como um parâmetro. Essa última adaptação tem como base o fato de que instâncias de *Data Class* oferecem muitos dados e poucos serviços, e, portanto, seus métodos normalmente possuem poucos parâmetros. Finalmente, no caso da métrica NOPA não foi necessário efetuar nenhum tipo de adaptação já que toda a informação utilizada por essa métrica está também disponível nos diagramas de classes. Sendo assim, a estratégia de detecção de *Data Class* para modelos ficou da seguinte forma:

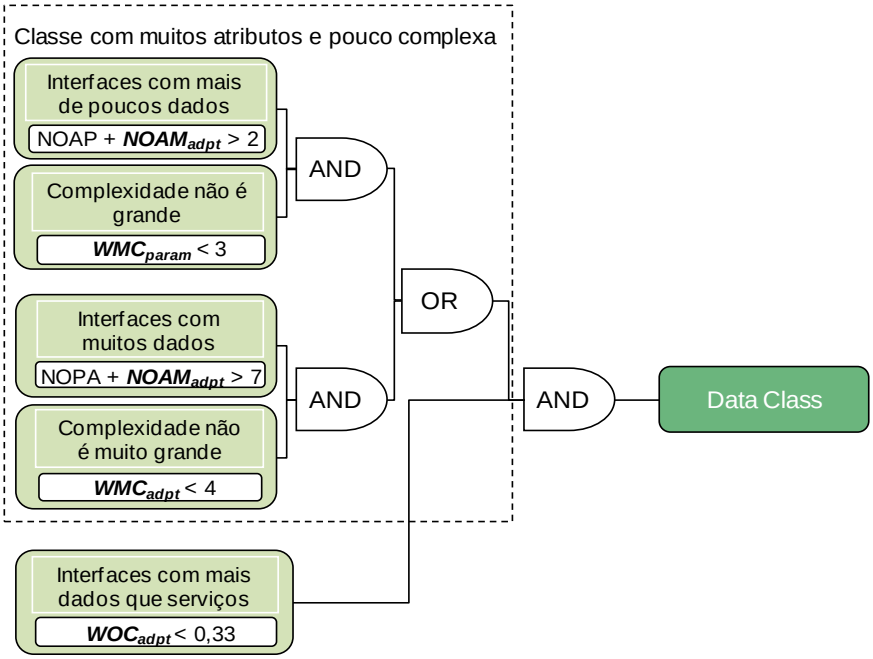

Figura 4 – Estratégia de detecção de *Data Class* para modelo.

Na Figura 4, WOC_{adpt}, e $NOAM_{adpt}$ referem-se às métricas WOC e NOAM adaptadas, e WMC_{param} refere-se a WMC com valor de complexidade dos métodos igual ao número de parâmetros. Classes internas não são consideradas pela estratégia.

A maioria dos valores limites utilizados na estratégia de detecção *Data Class* para modelo correspondem aos valores utilizados na estratégia homóloga para código. Isto é motivado pelo fato de que, na maioria das adaptações, procurou-se com que a forma de computar as métricas no modelo correspondesse com suas homólogas no código. No caso particular da métrica WMC_{adpt}, como no diagrama de classe não é possível computar a complexidade ciclomática dos métodos, passou-se a considerar como valor de complexidade a quantidade de parâmetros dos métodos. Portanto, foi necessária a adaptação do seu valor limite de acordo à nova forma de cálculo.

4.2.2. *God Class*

Em um bom *design* OO, o trabalho do sistema deve tender a ser uniformemente distribuído entre as classes que o compõe. O problema de *design God Class* se refere às classes que tendem a centralizar a maioria do trabalho de um sistema (Riel, 1996). Ao fazer isso, uma instância de uma *God Class* usa

dados de várias outras classes e delega a elas apenas detalhes de implementação (Riel, 1996). Sendo assim, normalmente, estas classes apresentam as seguintes características: (i) elas acessam muitos dados de outras classes, (ii) são grandes e complexas, e (iii) têm muitos métodos que não interdependem entre si, ou seja, existe baixa coesão entre seus métodos. Sendo assim, instâncias de *God Class* têm impacto negativo nos atributos da qualidade facilidade de reuso e facilidade de compreensão, já que, tanto para reusá-las como para compreender seu funcionamento, é necessário reusar e compreender todas as classes das quais as instâncias de *God Class* dependem (Seção 2.1).

4.2.2.1. Estratégia para Código Fonte

Lanza & Marinescu (2006) definem a estratégia para a detecção de classes com essa anomalia com base em três características prováveis dessas classes mencionadas anteriormente. Primeiramente, são detectadas as classes que dependem fortemente dos dados provenientes de outras classes. Depois, o conjunto de classes detectadas é reduzido, eliminando as classes pequenas e com alta coesão. Sendo assim, a estratégia de detecção para código é definida da seguinte forma:

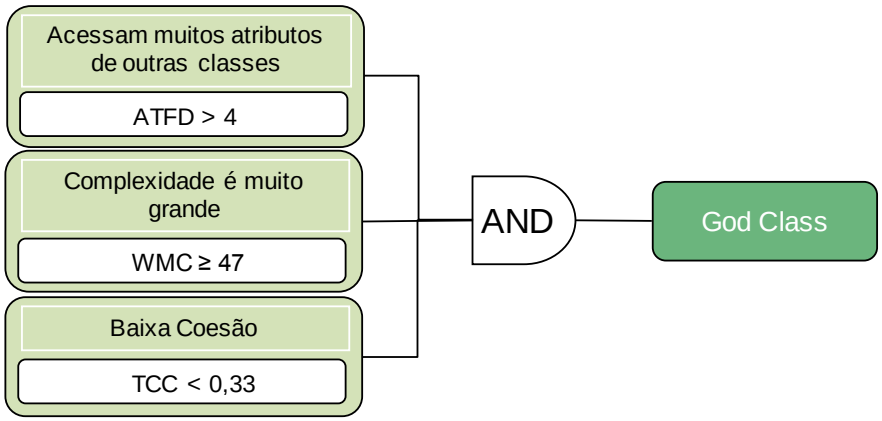

Figura 5 – Estratégia de detecção de *God Class* para código.

Na Figura 5, a métrica ATFD (*Access To Foreign Data*) conta o número de atributos de outras classes acessados, diretamente ou por meio de métodos de acesso, pela classe avaliada, a métrica WMC (*Weighted Method per Class*) é computada da mesma forma que para a estratégia *Data Class* (Seção 4.2.1), e a métrica TCC (*Tight Class Cohesion*) conta o número relativo de métodos de uma

classe que acessam pelo menos um atributo em comum. Os detalhes de implementação e as referências originais dessas métricas podem ser encontrados em (Lanza & Marinescu, 2006). A seguir são explicadas as cláusulas da estratégia de detecção.

1. **God Classes acessam muitos dados de outras classes.** Mediante à métrica ATFD se pode estimar a quantidade de dados acessados de outras classes. Quanto maior for esse número, maior é também a probabilidade que a classe seja uma instância de uma *God Class*.
2. **God Classes são grandes e complexas.** Esta característica é avaliada mediante a cláusula que diz que `WMC ≥ 47`. O valor 47 representa a soma das complexidades ciclomáticas dos métodos da classe.
3. **God Classes possuem baixa coesão entre seus métodos.** Instâncias de *God Class* utilizam conjuntos disjuntos de atributos para apoiar as funcionalidades que elas oferecem. Esta característica é avaliada pela cláusula que diz que `TCC < 0,33` em que o valor limite especificado indica que menos de um terço do total de pares de métodos da classe acessam um mesmo atributo.

4.2.2.2. Estratégia de Detecção para Modelo

As três métricas usadas na estratégia de detecção de *God Class* para código – ATFD, WMC e TCC – dependem de informações disponíveis exclusivamente no código interno dos métodos, respectivamente, para: (i) determinar os atributos de outras classes acessados pelos métodos, (ii) calcular a complexidade ciclomática (McCabe, 1976) dos métodos, e (iii) calcular a coesão de uma classe com base nos atributos acessados por cada método. Como essas informações não estão disponíveis em diagramas de classes, foi necessário realizar adaptações para definir a estratégia de detecção de *God Class* para modelos.

Primeiro, foi substituída a métrica TCC pela métrica CAM (*Cohesion Among Methods*) (Bansiya & Davis, 1999). CAM calcula a coesão de uma classe com base na quantidade de pares de métodos que possuem parâmetros do mesmo tipo. Porém, adaptou-se a implementação dessa métrica de forma que agora passou-se a considerar apenas aqueles parâmetros cujo tipo é definido pelo usuário. Segundo, no cálculo de WMC considerou-se que todos os métodos têm

valor de complexidade igual a um. Esta adaptação é baseada no fato de que uma das características das *God Class* é que elas normalmente são grandes e oferecem muita funcionalidade. Finalmente, na computação de ATFD, foram consideradas apenas referências a outras classes realizadas nas declarações de atributos e parâmetros, além das relações de associação e dependência entre elas. Além disso, ATFD no código conta o número de atributos acessados, enquanto, no modelo, conta apenas o número de classes referenciadas. Dessa forma, a estratégia de detecção de *God Class* para modelos ficou da seguinte forma:

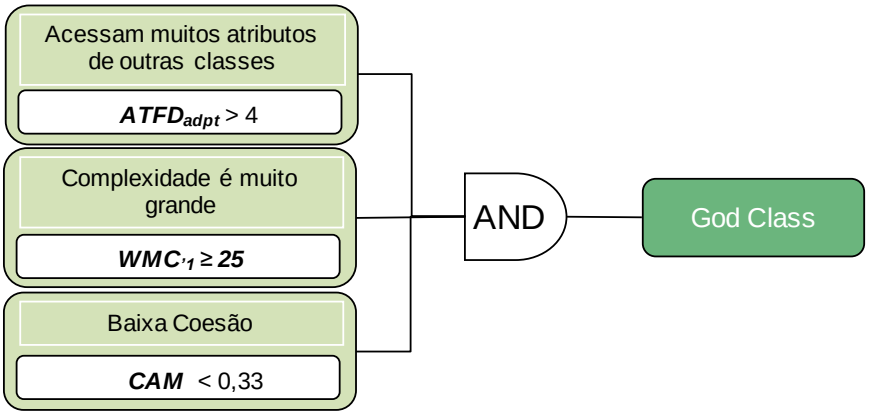

Figura 6 – Estratégia de detecção de *God Class* para modelo.

Na Figura 6, $ATFD_{adpt}$ refere-se à métrica ATFD adaptada e WMC_1 refere-se a WMC com valor de complexidade dos métodos igual a um, ou seja, o número de métodos públicos.

Na estratégia de detecção *God Class* para modelo apenas o valor limite associado à métrica WMC não corresponde ao mesmo valor limite utilizado na estratégia homologa para código fonte. Isso é motivado pelos seguintes fatos: (i) no caso da métrica ATFD, as informações referentes às dependências de uma classe com outras estão também disponíveis nos diagramas de classes, só que com nível menor de detalhe, (ii) como as complexidades dos métodos não podem ser computadas no modelo, foram consideradas como classes grandes aquelas cuja quantidade de métodos é superior a 25, e (iii) para a métrica $CAM_{adpt,}$ a princípio decidimos manter o mesmo valor limite, já que um terço é um valor padrão e não foram realizados estudos a fim de respaldar melhores aproximações para essa métricano futuro, esse valor poderá ser refinado.

4.2.3.
Shotgun Surgery

O problema de *design Shotgun Surgery* corresponde a classes cuja modificação implica muitas pequenas modificações em muitas outras classes (Fowler et al, 1999). Quando as modificações estão espalhadas, elas são difíceis de encontrar e, por isso, é muito provável esquecer uma modificação importante. Por exemplo, quando são feitas modificações em algoritmos, deve-se verificar se as modificações realizadas não afetaram todos os clientes do algoritmo. Nesses casos, ferramentas de análise estática não ajudam na identificação dos possíveis conflitos. Isto acarreta, portanto, um impacto negativo na manutenibilidade dos sistemas já que as mudanças não estão agrupadas (Seção 2.1). As classes que possuem este problema de *design* normalmente apresentam as seguintes características: (i) seus métodos são muito utilizados por outras classes, e (ii) elas têm muitas outras classes como clientes.

4.2.3.1.
Estratégia de Detecção para Código Fonte

Marinescu (2002) define a estratégia detecção de *Shotgun Surgery* para código fonte a partir de: (i) quantidade de métodos que poderão sofrer modificações devido a mudanças em uma determinada classe, e (ii) quantidade de classes que deverão ser inspecionadas devido a mudanças feitas em uma classe avaliada. Baseada nestas características, a estratégia para a detecção de instâncias *Shotgun Surgery* é definida da seguinte forma seguinte forma (Marinescu, 2002):

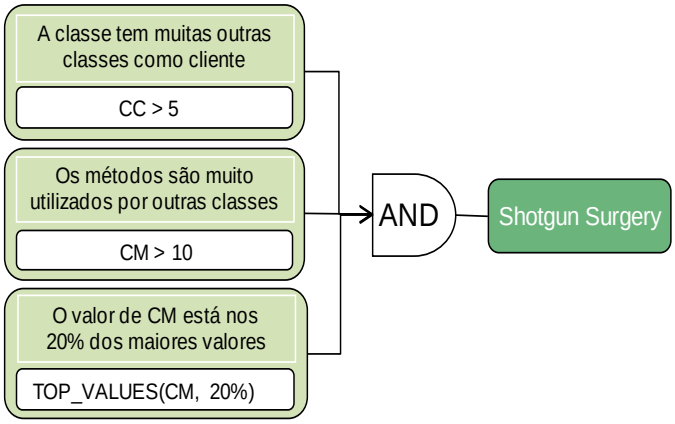

Figura 7 – Estratégia de detecção de *Shotgun Surgery* para código.

Figura 7, a métrica CM (*Changing Methods*) conta o número de métodos que podem ser potencialmente afetados por mudanças na classe avaliada, e CC (*Changing Class*) conta o número de classes que deverão ser inspecionadas como resultado às mudanças na classe avaliada. Os detalhes de implementação e as referências originais dessas métricas podem ser encontrados em (Marinescu, 2002). A seguir são explicadas as cláusulas da estratégia de detecção.

1. **Os métodos são muito utilizados por outras classes.** Quando os métodos de uma classe são modificados, devem ser inspecionadas todas as dependências da classe. Se a quantidade de dependências for alta, o risco de esquecer alguma delas é alto também.
2. **A classe tem muitas outras classes como cliente.** Se todos os métodos que utilizam informação de uma determinada classe estiverem agrupados em poucas classes, potenciais mudanças estariam melhor localizadas, reduzindo o risco de esquecer alguma delas.
3. **O valor da métrica CM está nos 20% dos maiores valores.** Esta filtragem é utilizada com o objetivo de reduzir o conjunto de classes detectadas identificando apenas aquelas que possuem os maiores valores da métrica CM.

4.2.3.2. Estratégia de Detecção para Modelo

As duas métricas utilizadas na estratégia para código – CM e CC – dependem de informações disponíveis no código interno dos métodos para determinar: (i) os métodos que podem ser afetados por modificações na classe avaliada, e (ii) a quantidade de classes em que são definidos os métodos possivelmente afetados pelas mudanças na classe avaliada. Conseqüentemente, foi necessária sua adaptação para serem utilizadas na estratégia para modelos.

Primeiramente, na computação da métrica CM, passou-se a considerar os métodos de outras classes: (i) com parâmetros cujos tipos de fazem referência à classe avaliada, e (ii) que redefinem algum método da classe avaliada. Porém, ainda com esta adaptação, os métodos detectados no modelo representam uma porção baixa dos detectados utilizando informações disponíveis no código fonte. Visando diminuir esta limitação, foi incluída, para a detecção dos métodos

potencialmente afetados, a métrica CA (*Changing Attributes*). Esta métrica representa a quantidade de atributos cujo tipo faz referência à classe avaliada. Esta adaptação está apoiada pelo fato de que os atributos de uma classe normalmente são acessados pelos métodos por ela definidos.

Da mesma forma que na versão para código fonte, no cálculo da métrica CC, foram consideradas as classes cujos métodos utilizam informações da classe avaliada. Porém, existem dependências entre classes que são feitas por meio de declarações de variáveis locais, chamadas a métodos estáticos, entre outros. Como essas informações não se encontram disponíveis nos diagramas de classes, foram consideradas também no cálculo da métrica CC, as classes que apresentam com a classe avaliada, relações de associação e dependência. Dessa forma, a estratégia de detecção de *Shotgun Surgery* para modelos é definida da seguinte maneira:

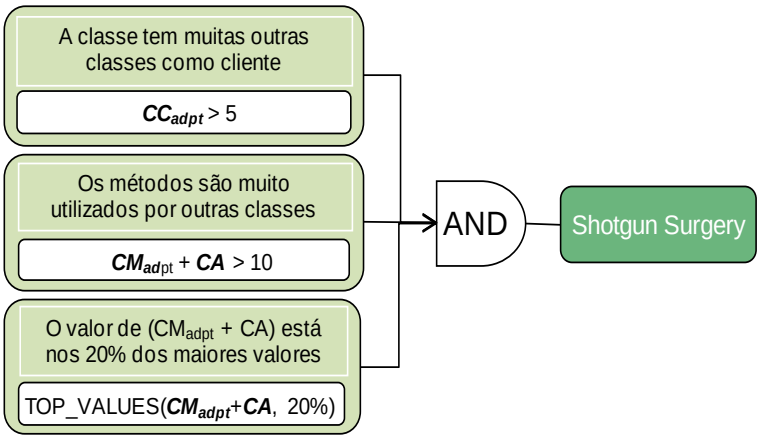

Figura 8 – Estratégia de detecção de *Shotgun Surgery* para modelo.

Na Figura 8, as métricas CM_{adpt} e CC_{adpt} referem-se respectivamente as adaptações realizadas às métricas CM e CC.

Na estratégia de detecção *shotgun surgery* para modelo foram utilizados os mesmos valores limites usados na estratégia correspondente para código fonte (Figura 8). Isto é motivado pelo fato de que se passou a considerar no cálculo da métrica CC_{adpt} as relações de dependências e associações entre as classes disponíveis nos diagramas de classes. Além disso, para compensar a ausência de informação no cálculo da métrica CM_{adpt} utilizou-se a métrica CA, permitindo desta forma, alcançar maiores valores na estimativa do acoplamento e portanto, manter o valor limite utilizado na cláusula correspondente na estratégia de código fonte. No caso da cláusula TOP_VALUES(CM_{adpt}+CA, 20%) o valor limite foi

mantido com o objetivo de detectar as classes com a mesma relevância que na estratégia para código.

4.3. Problemas de *Design* em Pacotes

Pequenos sistemas de software podem ser organizados por meio das classes. Estas entidades agrupam as definições das funcionalidades do sistema. Porém, em sistemas grandes torna-se necessário ter uma representação, em nível mais alto, da organização dos sistemas, o que pode ser feito por intermédio de pacotes.

Muitos pesquisadores têm se dedicado à tarefa de definir regras e/ou heurísticas que capturem desvios de uma boa organização de pacotes (Martin, 1997; Lakos, 1996). Baseado nestas pesquisas, Marinescu (2002) definiu um conjunto de estratégias de detecção a fim de oferecer continuidade à detecção de problemas de *design* no nível de pacotes.

Neste trabalho, utilizaram-se duas das estratégias de detecção definidas por Marinescu para serem adaptadas e aplicadas sobre os modelos. Estas estratégias foram escolhidas porque visam identificar problemas de coesão, propriedade do *design* OO que tem sido abordada neste trabalho (Seção 2.1). A principal conseqüência dos problemas de coesão no nível de pacotes é que eles tendem a agrupar classes de maneira inadequada. Isto pode conduzir a ter poucos pacotes muito grandes, ou a muitos pacotes pequenos. As estratégias de detecção estão relacionadas com um dos três princípios abordados em (Martin, 2002): o *Common Closure Principle* (CCP). Este princípio está relacionado à manutenibilidade e diz que as classes definidas em um pacote devem estar fortemente ligadas e sujeitas aos mesmos tipos de mudanças. Sendo assim, uma mudança que afeta um pacote, deve afetar as classes definidas nele e não a classes localizadas em outros pacotes. Portanto, o CCP minimiza o esforço relacionado à execução de teste, e redistribuição do software.

4.3.1. *God Package*

O problema de *design God Package* refere-se a pacotes que tendem a ser muito grandes e possuir classes que não interdependem, ou seja, classes com

baixa coesão entre elas. Além disso, segundo Marinescu (2002), outro sintoma deste problema são pacotes que possuem muitos clientes (classes definidas em outros pacotes) que dependem muito dele. Conseqüentemente, quando for realizada uma mudança nesse pacote deve-se verificar se a mudança realizada não afetou os clientes (Marinescu, 2002).

4.3.1.1. Estratégia de Detecção para Código Fonte

Marinescu (2002) define a estratégia para a detecção de pacotes com essa anomalia com base em três características comuns desses pacotes. Primeiramente, são detectados os pacotes que contêm muitas classes e que são muito utilizados por classes definidas em outros pacotes. O fato de que as classes que utilizam o pacote avaliado estejam definidas em diferentes pacotes do sistema aumenta a probabilidade de o pacote ser uma instância de *God Package*. Finalmente, o conjunto de pacotes identificados é reduzido, eliminando aqueles que apresentam alto grau de coesão, ou seja, as classes dependem entre si. Sendo assim, a estratégia de detecção para código é definida da seguinte forma:

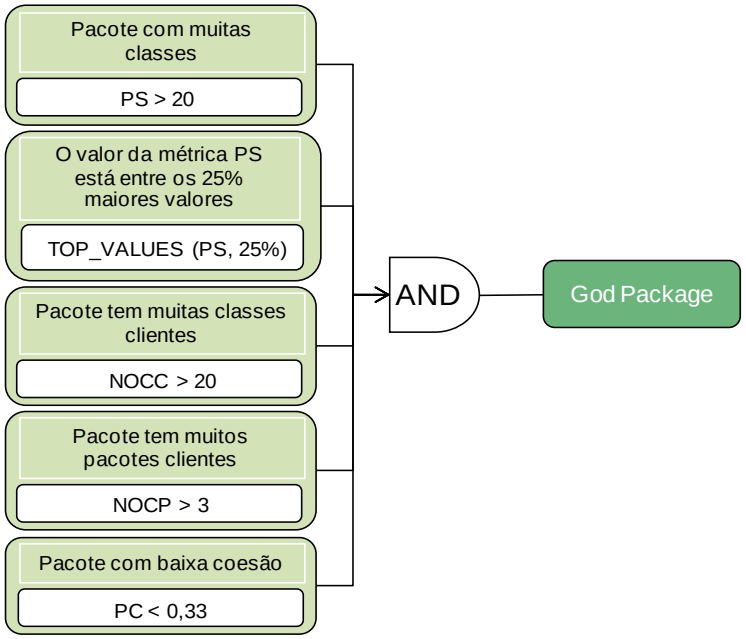

Figura 9 – Estratégia de detecção de *God Package* para código.

Na Figura 9, a métrica PS (*Package Size*) conta o número de classes que estão definidas no pacote avaliado. A métrica NOCC (*Number Of Client Classes*) representa o número de classes definidas em outros pacotes e que usam o pacote

avaliado. Uma classe utiliza um pacote se ela realiza chamadas a métodos, acessa a variáveis ou herda de alguma classe definida nesse pacote. A métrica NOCP (*Number Of Client Packages*) conta o número de pacotes que utilizam o pacote avaliado. Um pacote A utiliza outro pacote B se ao menos uma das classes definidas em A usa alguma classe do pacote B. A métrica PC (*Package Cohesion*) é definida como o número relativo de pares de classes que dependem entre si. A seguir são explicadas as cláusulas da estratégia de detecção.

1. **Pacotes com muitas classes.** Segundo Marinescu & Lanza (2006), normalmente, nos pacotes cuja interface não é realizada através de uma fachada, quanto maior for a quantidade de classes definidas em um pacote, maior será a quantidade de funcionalidades oferecidas por esse pacote. Conseqüentemente, é maior a probabilidade que o pacote esteja centralizando grande parte da funcionalidade do sistema e seja uma instância de uma *God Package*. Esta característica é avaliada pela cláusula que diz que `PS > 20`.
2. **O valor da métrica PS está entre os 20% maiores valores.** Esta filtragem é utilizada com o objetivo de identificar apenas as classes que possuem os maiores valores da métrica PS.
3. **Pacote tem muitas classes clientes.** Esta característica é avaliada mediante a cláusula que diz que `NOCC ≥ 20`. Isto indica que o pacote tem mais de 19 classes clientes em outros pacotes e que, portanto, modificações nesse pacote provocarão que classes definidas em outros pacotes tenham que ser inspecionadas.
4. **Pacote tem muitos pacotes clientes.** Esta característica traz como conseqüência que modificações no pacote avaliado provoquem inspeções em muitos outros pacotes diferentes. Conseqüentemente, o risco de esquecer alguma inspeção e/ou mudança e a probabilidade de que o pacote esteja centralizando a funcionalidade do sistema serão maiores. Esta característica é avaliada mediante a cláusula que diz que `NOCP > 3`.
5. **Pacote tem pouca coesão entre suas classes.** Uma característica comum nos *God Packages* é que suas classes se comunicam pouco entre si. Esta característica é avaliada mediante a cláusula que diz `PC < 0,33`

em que o valor limite especificado indica que menos de um terço do total de pares de classes relacionam-se entre si.

4.3.1.2. Estratégia de Detecção para Modelos

Na estratégia de detecção para a localização de *God Package* para código se pode observar que duas das três métricas usadas – NOCC, NOCP e PC – dependem de informações disponíveis no código dos métodos, respectivamente, para determinar: (i) as chamadas a métodos de outras classes realizadas por uma determinada classe, e (ii) a quantidade de variáveis definidas em outras classes que uma determinada classe acessa. Por causa disto, adaptamos as métricas NOCC, NOCP e PC de forma que elas passassem a considerar apenas que uma classe depende de outra classe quando: (i) existem relações de dependência, herança ou associação entre elas, e (ii) ou utiliza informações por meio de declarações de atributos e/ou parâmetros. Dessa forma, a estratégia de detecção de *God Package* para modelos foi definida da seguinte forma:

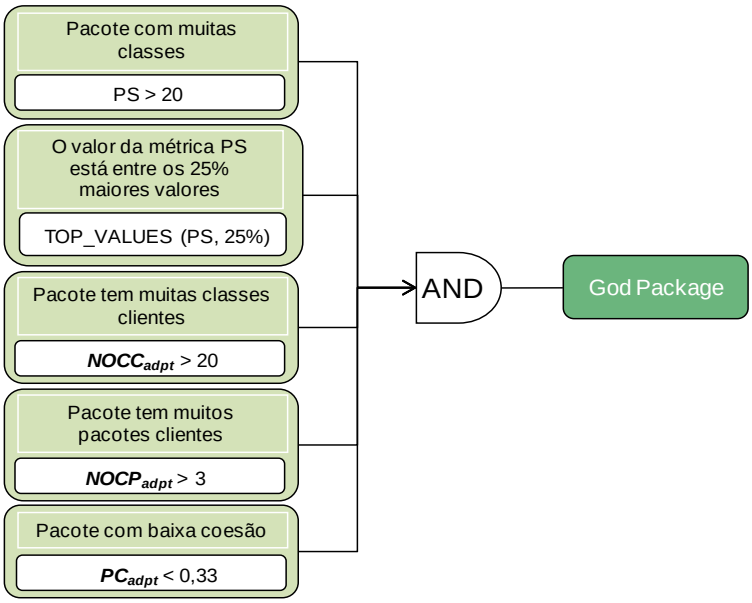

Figura 10 – Estratégia de detecção de *God Package* para modelo.

Na Figura 10, $NOCC_{adpt}$, $NOCP_{adpt}$ e PC_{adpt} referem-se, respectivamente, às métricas NOCC, NOCP e PC adaptadas.

Embora o volume de informação disponível nos diagramas de classes para o cômputo da métrica $NOCC_{adpt}$ seja menor que no código, o valor limite utilizado na estratégia para código foi mantido na versão para modelo. Isto é motivado

porque na computação dessa métrica na versão para modelo, as classes clientes sempre são consideradas, mesmo que elas sejam apenas utilizadas como parâmetros para a instanciação de outros objetos. Já na versão para código, essa métrica somente considera aquelas classes cuja informação seja utilizada (e.x. atributos, métodos e redefinições de métodos). Isto faz com que em alguns casos a seguinte relação seja válida $NOCC_{adpt} > NOCC$. No caso da métrica $NOCP_{adpt}$, embora seu valor esteja condicionado às classes detectadas pela métrica $NOCC_{adpt}$, não foi possível diminuir seu valor limite por ser um valor absoluto muito pequeno. Finalmente, como a métrica PC_{adpt} é relativa, existe uma correspondência entre os valores obtidos tanto no código quanto no modelo. Por causa disso, o valor limite utilizado na estratégia para código pode ser mantido.

4.3.2. *Misplaced Class*

O problema de *design Misplaced Class* faz referência às classes em que o grau de interdependência com as demais classes definidas no pacote é baixo, além de dependerem muito de classes definidas em outros pacotes. Os pacotes que são identificados como possíveis instâncias de *God Package* geralmente possuem classes que dependem mais das classes definidas em outros pacotes do que as definidas nele próprio. Uma classe que utiliza principalmente as funcionalidades definidas em um pacote diferente do seu, deveria provavelmente ser movida para esse outro pacote. Desta maneira, refatorando os possíveis *God Packages* reduz-se a quantidade de instâncias de *Misplaced Class* que ele possui. Porém, um pacote pode possuir classes com este problema de *design* sem ser considerado *God Package*. Daí ser necessária também a definição de estratégias para a detecção das instâncias de *Misplaced Class*.

4.3.2.1. Estratégia de Detecção para Código Fonte

Marinescu (2002) define a estratégia para a detecção de classes com essa anomalia com base nas três características comuns dessas classes: (i) classes que dependem muito de classes definidas em pacotes diferente ao seu, (ii) classes que dependem mais das classes definidas em outros pacotes que das definidas no seu

pacote, e (iii) classes cujas dependências com outras classes estão espalhadas em diferentes pacotes. Sendo assim, a estratégia de detecção para código é definida da seguinte forma:

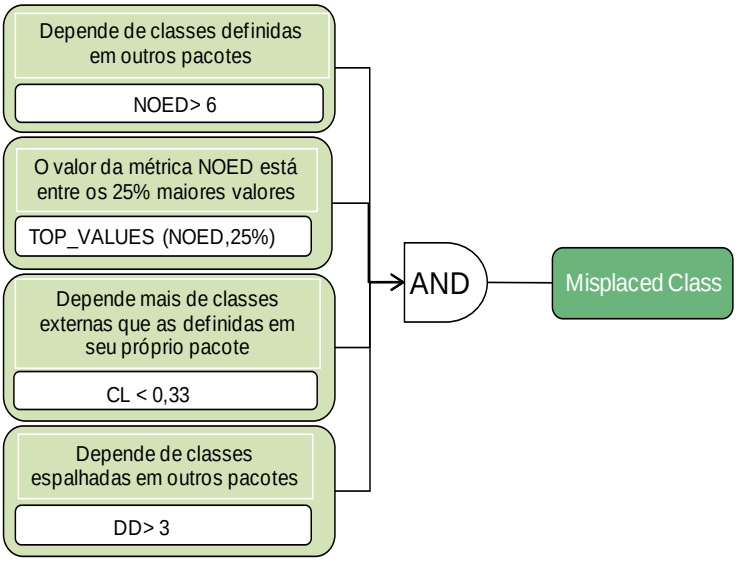

Figura 11 – Estratégia de detecção de *Misplaced Class* para código.

Na Figura 11, (i) a métrica NOED (*Number Of External Dependencies*) conta o número de classes de outros pacotes das quais a classe avaliada depende, (ii) a métrica CL (*Class Locality*) conta a porcentagem de dependências que uma classe tem em seu próprio pacote em relação ao total de dependências que ela possui (uma classe A depende de outra classe B se a classe A chama algum método, e/ou acessa algum atributo e/ou herda da classe B), e (iii) a métrica DD (*Dependency Dispersion*) representa o número de outros pacotes dos quais a classe avaliada depende. Uma classe depende de um pacote se ela depende de alguma classe definida nesse pacote. A seguir são explicadas as cláusulas da estratégia de detecção.

1. **Depende de classes definidas em outros pacotes.** Esta característica é identificada por meio da cláusula que diz que `NOED > 6`. Isto significa que a classe para seu funcionamento depende de mais de `6` classes definidas em outros pacotes.
2. **Depende mais de classes externas que das definidas em seu próprio pacote.** Quando uma classe depende mais das classes definidas em outros pacotes do que das definidas em seu pacote, a localização dessa classe tem que ser analisada. Esta característica é representada pela cláusula que diz que `CL < 0,33` em que o valor limite especificado

indica que menos de um terço do total de classes das que a classe avaliada depende se encontram definidas no seu mesmo pacote.

3. **Depende de classes espalhadas em outros pacotes.** Esta característica é representada pela cláusula que diz que `DD > 3`. Isto significa que a classe depende de mais de `3` pacotes diferentes. Quanto maior for a quantidade de pacotes diferentes dos quais a classe depende, maior será a probabilidade de ela ser instância de uma *Misplaced Class*.

4.3.2.2. Estratégia de Detecção para Modelo

Analisando a estratégia de detecção *Misplaced Class* para código pode-se observar que as três métricas utilizadas – NOED, CL, DD – dependem de informações disponíveis apenas no código interno dos métodos para determinar: (i) quando uma classe depende de outra classe, e (ii) quando um pacote depende de outro pacote. Igualmente à estratégia de detecção God Package considerou-se que uma classe depende de outra classe apenas quando: (i) existem relações de dependência, herança ou associação entre elas, (ii) referências são realizadas nas declarações de atributos e parâmetros. De maneira semelhante na estratégia de detecção *God Package*, uma classe depende de um pacote quando utiliza alguma das classes definidas em ele. Mantendo as mesmas métricas utilizadas no código, porém, com as adaptações realizadas a serem utilizadas no modelo, a estratégia de detecção de *Misplaced Class* para modelos é definida da seguinte forma:

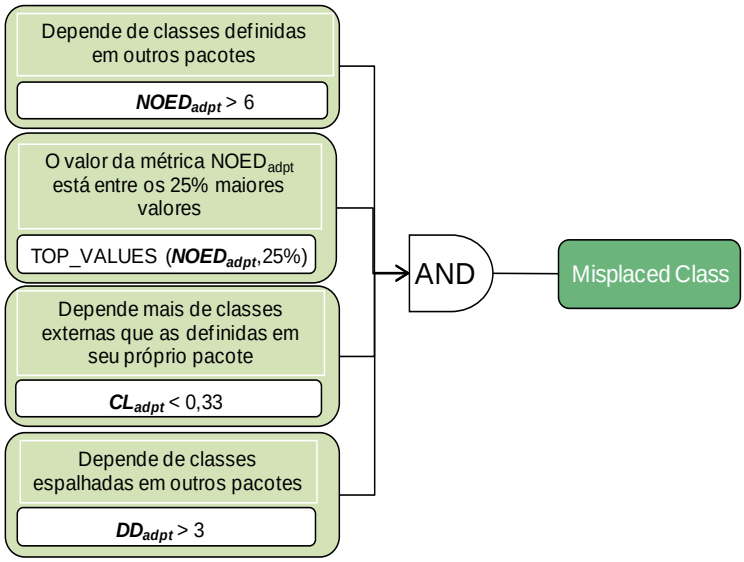

Figura 12 – Estratégia de detecção de *Misplaced Class* para modelo.

Na Figura 12, $NOED_{adpt}$, CL_{adpt} e DD_{adpt} referem-se às métrica NOED, CL e DD adaptadas. Para o cálculo da métrica CL_{adpt} dividiu-se o valor da métrica $NOED_{adpt}$ pelo número total de dependências que a classe avaliada possui.

Na estratégia de detecção *Misplaced Class* para modelo, os valores limites associados às métricas, correspondem aos mesmos valores limites utilizados na estratégia homóloga para código fonte. Embora o nível de detalhe da informação para determinar se uma classe depende de outra seja menor que no código fonte, os valores limites utilizados para as métricas NOED e DD no código são absolutos e pequenos, o que impossibilita diminuí-los para serem adaptados às novas condições. No caso da métrica CL_{adpt}, como ela é uma métrica relativa, existe uma correspondência entre os valores obtidos tanto no código quanto no modelo. Por causa disso, o valor limite utilizado na estratégia para código pode ser mantido.

4.4. Problemas de *Design* para Métodos

Os métodos possuem um papel fundamental na programação OO já que são eles que definem as funcionalidades de uma classe. Os métodos contêm informação detalhada sobre o comportamento do sistema. Utilizando este ponto de vista, a maioria dos princípios e regras heurísticas da programação estruturada é aplicada a níveis de métodos. Como o escopo deste trabalho é a detecção de problemas em *design* OO utilizando diagramas de classes, sendo que estes diagramas não oferecem detalhes relativos a métodos além de suas características estruturais, considerou-se apenas uma estratégia de detecção para métodos. Esta estratégia, portanto, é baseada nas características estruturais dos métodos, que são também relevantes no contexto OO.

4.4.1. *Long Parameter List*

O problema de *design Long Parameter List* corresponde a métodos que possuem uma lista longa de parâmetros (Fowler et al, 1999). Na programação OO muita da informação que os métodos precisa está armazenada na própria classe. Porém, quando um objeto não possui todas as informações necessárias para seu

funcionamento, por meio da troca de mensagens, ele pode perguntar a outro objeto e desta forma tem acesso ao que precisa. Sendo assim, os métodos na programação OO não precisam receber toda a informação necessária através de seus parâmetros. Isto faz com que as listas de parâmetros sejam menores que nos paradigmas de programação tradicionais. Listas de parâmetros muito extensas afetam a facilidade de compreensão dos métodos, aumentam a chance de erros nas chamadas dos métodos e reduzem a capacidade dos compiladores verificarem a corretude dos argumentos em tempo de compilação.

Considerando que a informação necessária para a identificação de métodos com este problema de *design* está disponível tanto nos modelos como no código, não foi necessária a definição de duas versões. A estratégia de detecção de *Long Parameter List* é definida da seguinte forma:

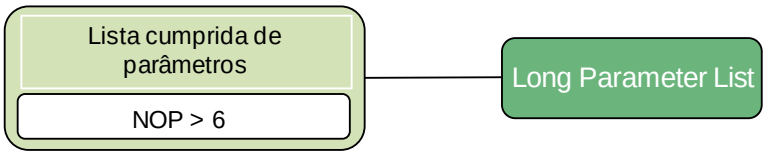

Figura 13 – Estratégia de detecção de *Long Parameter List*.

Na Figura 13 a métrica NOP (*Number Of Parameters*) conta o número dos parâmetros que um método possui. Considerando que não existe uma estratégia definida previamente para código fonte, o valor limite utilizado é um valor inicial, mas que deve ser ajustado com a experiência de uso da estratégia.