2
Avaliação Quantitativa do *Design* de Software

Este trabalho discute técnicas automatizadas baseadas em métricas para a detecção de problemas de *design* OO utilizando os diagramas UML. O trabalho está relacionado com os seguintes conceitos: (i) *design* de software OO, (ii) medição de software e (iii) mecanismos para o controle da qualidade do *design* OO. O objetivo deste capítulo é familiarizar o leitor com estes conceitos.

Este capítulo está estruturado da seguinte forma; primeiramente são apresentadas algumas das principais características de *design* de software e seus impactos nos atributos da qualidade. A segunda parte aborda os conceitos fundamentais da medição de software. No âmbito deste trabalho foi adotada a definição de medição apresentada em (Fenton & Pfleeger, 1997). Na terceira e última parte deste capítulo, são introduzidos os mecanismos de análises da qualidade de *design* que são utilizados no resto deste trabalho.

2.1.
Design de Software Orientado a Objetos

“O *Design* Orientado a Objeto (OO) é uma estratégia de *design* na qual os engenheiros de sistemas pensam em termos de coisas ao invés de operações ou funções. A execução do sistema é feita por meio da interação de objetos que provêem operações e informações” (Sommerville, 1995).

A qualidade do *design* OO tem um forte impacto no processo de desenvolvimento dos sistemas de software. Considerando o ciclo de desenvolvimento de sistemas de software, na literatura podem ser encontradas estimativas que revelam que a etapa de *design* é responsável por não mais do que 10% até 15% do esforço total do desenvolvimento. Além disso, estudos revelam que boa parte dos custos investidos está relacionada com a correção de *design* incorreto gerados nessa etapa (Bell et al, 1987).

Fenton e Pfleeger (1997) expressaram as características de um bom *design* de software nos seguintes termos:

PUC-Rio - Certificação Digital Nº 0711290/CA

"*Design* de alta qualidade deve ter características que levam à qualidade dos produtos: facilidade de compreensão, facilidade de implementação, facilidade para a aplicação dos testes, facilidade de modificação, e o mapeamento correto dos requisitos especificados".

Um bom *design* pode produzir uma baixa manutenção devido ao mapeamento simplificado entre as entidades do domínio do problema e os objetos envolvidos no *design*. Este mapeamento simplificado provê menos esforço da análise e uma baixa complexidade no *design*. Além disso, com a facilidade de uso dos artefatos do *design* é possível economizar tempo e recursos.

Pesquisadores têm se preocupado em definir técnicas para apoiar a concepção de *design* de boa qualidade, tais como, padrões de projeto (Gamma et al., 1995) e refatorações (Fowler et al, 1999). Além disso, mecanismos, tais como, *bad smells* (Fowler et al, 1999) e estratégias de detecção (Marinescu, 2002) têm sido propostos para avaliar a qualidade de *design* OO. Yourdon & Constantine (1979) assumem que um *design* no qual os módulos têm baixo acoplamento entre si e alta coesão dá origem a código mais confiável e fácil de manter. A seguir são apresentadas algumas das características mais relevantes de um bom *design* OO e seus impactos na qualidade.

PUC-Rio - Certificação Digital Nº 0711290/CA

Baixo Acoplamento

No *design* OO, o conceito de acoplamento está relacionado com o "grau de interdependência que existe entre duas peças" (Coad & Yourdon, 1991) e depende do volume de elementos que constituem a interface e da forma com que é estabelecida a interface por meio desses elementos (Staa, 2000). Um módulo, por exemplo, com elevado (ou forte) acoplamento interage com muitos outros módulos e/ou possui uma interface extensa. Nos sistemas em que os módulos estão fortemente acoplados, uma pequena mudança em algum deles terá um impacto em todos os outros módulos que com que interage o módulo no qual foi feita a mudança.

O nível de acoplamento entre as entidades de um sistema tem sido utilizado na literatura como um critério importante para avaliar a qualidade do *design*. A *facilidade de uso* de um sistema pode ser afetada em sistemas que apresentam forte acoplamento entre seus módulos já que a utilização de um módulo precisará de todos os outros módulos dos quais o módulo utilizado

depende. A *facilidade de compreensão* de um módulo também é afetada, pois será necessária a compreensão transitiva de todos os outros módulos com os quais o módulo a ser compreendido interage. Finalmente, o módulo é *difícil de manter*, pois alterações precisam ser propagadas para muitos outros módulos.

Alta Coesão

A coesão é considerada como um aspecto complementar do acoplamento. Ela descreve "o grau de inter-relacionamento entre os elementos envolvidos em uma parte do *design* a fim de oferecer uma funcionalidade". O fato de uma classe possuir um alto grau de coesão implica que seus métodos e atributos estão fortemente inter-relacionados.

O nível de coesão das classes tem sido utilizado na literatura como um indicador importante para avaliar a qualidade do *design*. Um nível baixo de coesão afeta negativamente a *facilidade de compreensão* de uma classe já que seus métodos não estão focados em um mesmo conceito. Geralmente, essas classes possuem muita funcionalidade e conseqüentemente apresentam uma lista comprida de métodos. Às vezes é necessário agrupar os métodos com uma funcionalidade comum a fim de compreender os serviços oferecidos pela classe. Estas classes são *difíceis de reusar*, pois quando se quer reusar alguma(s) funcionalidade(s) oferecida(s), normalmente, as outras funcionalidades não fazem sentido serem utilizadas no mesmo contexto.

Complexidade Tratável

Níveis elevados de complexidade não são desejados, pois influenciam negativamente diferentes características do *design* especialmente a *facilidade de compreensão* e a *manutenibilidade*. Módulos com muita funcionalidade são *difíceis de compreender*, especialmente se eles possuem níveis baixos de coesão já que oferecem diferentes serviços que não estão relacionados entre si. A dificuldade de compreensão de classes complexas implica que elas também são *difíceis de manter, testar*, além de serem *pouco confiáveis*. O fato de que as classes complexas sejam difíceis de *testar* implica que elas sejam mais propensas a erros e por tanto, sua *confiabilidade* seja reduzida. "Reduzir a complexidade e o tamanho de um sistema deve ser o objetivo em cada um dos passos do desenvolvimento: especificação, *design* e implementação" (Wirth, 1995).

PUC-Rio - Certificação Digital Nº 0711290/CA

Boa Abstração de Dados

Uns dos princípios comumente utilizados para facilitar o entendimento de alguma coisa complexa é a abstração (Marinescu, 2002). No paradigma de programação orientada a objetos, a abstração está preocupada com o fato de destacar apenas aquelas características relevantes na perspectiva que o objeto vai ser analisado e ao mesmo tempo suprime todas as outras características do objeto. Por exemplo, nas classes, os elementos são visíveis para o restante do sistema por meio das interfaces (tipicamente, os atributos e métodos públicos). A abstração dos dados, tal como as outras características expostas, influencia na qualidade do *design*. Um *design* com uma abstração de dados bem delimitada expõe um bom nível de modularidade, o que faz com que seja mais *fácil de compreender* (Meyer, 1988).

Métricas de software têm sido definidas para capturar e quantificar desvios das características apresentadas (McCabe, 1976; Chidamber & Kemerer, 1994). Estas métricas são muito utilizadas pelos mecanismos de análise abordados no contexto deste trabalho para a detecção e localização de entidades causadoras dos desvios que elas quantificam.

PUC-Rio - Certificação Digital Nº 0711290/CA

2.2.
Medição de Software

À medida que a demanda por software complexo, confiável, fácil de utilizar aumenta, a medição de software passa a desempenhar um papel cada vez mais importante no entendimento e controle das práticas e produtos do desenvolvimento de software (Kitchenham et al, 1995). Alguns desenvolvedores medem características do software para verificar se os requisitos estão consistentes e completos, se o projeto tem boa qualidade ou se o código está pronto para ser testado. Alguns gerentes de projetos medem características do processo e do produto para serem capazes de dizer quando o software estará pronto para entrega ou se o orçamento será ultrapassado. Pessoas responsáveis pela manutenção podem usar a medição para avaliar se o produto precisa ser melhorado.

A medição pode ser definida em termos gerais como: o processo de atribuir números ou símbolos aos atributos de entidades do mundo real de tal forma que se possam descrever as entidades de acordo com regras claramente definidas (Fenton

& Pfleeger, 1997). Conseqüentemente, a medição captura informações sobre os atributos das entidades. Uma entidade é o sujeito sobre o qual é feita a medição. Pode ser um objeto físico ou evento que acontece num determinado momento de tempo. No caso da engenharia de software, exemplos de entidades podem ser: um objeto, uma classe, a especificação de um software ou uma etapa do processo de desenvolvimento do software, entre outras. Um atributo é uma característica ou propriedade de uma entidade. No caso da engenharia de software, um exemplo de atributo é o tamanho de uma classe em termos de linhas de código. Nesse caso, a entidade é a classe.

A primeira tarefa de qualquer atividade de medição é a identificação das entidades e atributos que se deseja e possa medir. Na engenharia de software são identificadas três classes de entidades (Fenton & Pfleeger, 1997): (i) processos, que são coleções de atividades relacionadas ao desenvolvimento de software, (ii) produtos, que são artefatos entregues ao cliente ou usuário, (iii) recursos, que são entidades requeridas para realizar uma atividade do processo.

Para cada uma destas classes de entidades podem-se distinguir atributos internos e atributos externos. Atributos internos são aqueles que podem ser medidos diretamente, ou seja, em termos dos processos, produtos e recursos separadamente de seu comportamento. Atributos externos são aqueles que podem ser medidos somente de acordo a como os processos, produtos e recursos são relacionados com seu ambiente. Assim, o comportamento é importante, em vez da própria entidade.

Para que a diferença entre estas classes de atributos possa ser entendida, considere um conjunto de módulos de software. Embora não se tenha executado o código correspondente aos módulos, pode ser examinado um conjunto importante de atributos internos: o tamanho (em termos de linhas de código ou número de operações), a complexidade (em termos de número de pontos de decisão no código – complexidade ciclomática) e as dependências entre os módulos. Já os atributos externos são aqueles que podem ser medidos somente quando o código for executado. Exemplos de atributos externos são: quantidade de falhas observadas pelos usuários, a dificuldade de navegação entre as telas ou a quantidade de tempo necessária para procurar uma informação no banco de dados.

Embora os produtos sejam artefatos entregues ao cliente, é importante destacar que os artefatos não são restritos apenas a estes itens. Qualquer artefato

PUC-Rio - Certificação Digital Nº 0711290/CA

produzido durante as etapas do ciclo de desenvolvimento do software (modelagem, desenvolvimento, teste, manutenção) pode ser medido e conseqüentemente avaliado. Existem muitos exemplos de atributos externos de produtos de software. Confiabilidade, facilidade de compreensão, manutenibilidade, usabilidade, integridade, eficiência, reusabilidade, portabilidade e interoperabilidade são exemplos de atributos externos de produtos. Esses atributos não estão relacionados apenas ao código, mas também a outros documentos e artefatos que dão apoio ao desenvolvimento de software. Por exemplo, a manutenibilidade e a reusabilidade das especificações de requisitos e do projeto de um software são tão importantes quanto a manutenibilidade e a reusabilidade do código.

Existem também muitos atributos internos de produtos. Especificações podem ser avaliadas em termos de seu tamanho, grau de reutilização, redundância e corretude sintática. O projeto detalhado e o código de um software podem ser avaliados por esses mesmos atributos e mais alguns outros como, por exemplo, acoplamento, coesão e estrutura de fluxo de controle e de dados. Projetistas e desenvolvedores estão interessados nos atributos internos (facilidade de compreensão, tamanho, grau de reutilização). Gerentes de projetos estão preocupados com a estimativa de tempo, esforço e produtividade. Entretanto os usuários finais estão preocupados com atributos externos como fidedignidade, facilidade de uso, eficiência e portabilidade.

Às vezes atributos externos são mais difíceis de medir que os atributos internos, pois só podem ser medidos tardiamente no processo de desenvolvimento. Atributos internos podem ser muito úteis para sugerir o que provavelmente será encontrado ao se avaliar os atributos externos (Fenton & Pflegeer, 1997). Por isso, é necessária a criação de modelos que definam as relações de dependências entre eles. Por exemplo, Yourdon & Constantine (1979) assumem que projetos de software que possuem módulos com baixo acoplamento entre si e alta coesão dão origem a códigos mais confiáveis e fáceis de manter. Neste trabalho, nos concentramos no estudo de estratégias de detecção e modelos da qualidade que possam ser úteis para capturam desvios dos princípios de bom *design* OO.

PUC-Rio - Certificação Digital Nº 0711290/CA

2.3. Modelos de Controle da Qualidade de *Design* OO

Um dos principais objetivos da engenharia de software é assegurar qualidade satisfatória dos produtos de software. Numa perspectiva de medição, qualidade de software deve ser definida em termos de atributos de produtos de software que são de interesse do usuário. Usuários de sistemas de software geralmente querem ser capazes de medir e predizer atributos externos, uma vez que o comportamento do sistema os afeta diretamente. Por outra parte, desenvolvedores de software precisam, nas diferentes etapas do desenvolvimento, analisar e monitorar a evolução dos sistemas. Para isto eles podem utilizar os atributos internos a fim de predizer os atributos externos. Este trabalho se concentra no estudo de dois tipos de mecanismos de controle da qualidade de *design* OO: modelos da qualidade (Bansiya & Davis, 2000) e estratégias de detecção. Estes dois tipos de mecanismos se baseiam na medição de atributos internos.

2.3.1. Modelos de Qualidade

Como a qualidade é composta de muitas características, ela pode ser interpretada de modo diferente de acordo com os interesses e o domínio do problema. Embora todas as pessoas envolvidas no desenvolvimento dos sistemas concordam que o produto final tem que ter um nível alto de qualidade, às vezes nem todas elas concordam com a forma de estimar a qualidade em termos de atributos externos. Por causa disto, engenheiros de software definiram modelos de qualidade para a estimativa dos atributos externos em termos dos atributos internos.

A necessidade da definição de bons modelos de qualidade é relevante, pois tais modelos descrevem os atributos que influenciam a qualidade e as relações entre eles. Além disso, eles podem ser utilizados como guias para o monitoramento dos sistemas já que podem oferecer as especificações e os níveis da qualidade que devem ser alcançados nas distintas etapas do desenvolvimento dos sistemas.

PUC-Rio - Certificação Digital Nº 0711290/CA

Os primeiros modelos definidos com o objetivo de descrever a qualidade do produto de software foram o modelo de McCall (1977) e o modelo de Boehm (1978). Nestes modelos os autores se concentram na qualidade do produto final (especialmente no código executável), e identificam os atributos principais da qualidade em termos da perspectiva do usuário. De acordo com (Fenton & Pflegeer, 1997) modelos de qualidade podem ser usados de duas maneiras:

- Um modelo já existente é escolhido e os relacionamentos entre os atributos externos, os atributos internos e as métricas são aceitos exatamente como propostos pelo autor do modelo.
- A filosofia geral de que um atributo externo é influenciado por vários atributos internos é aceita, mas não é adotado um modelo de qualidade já desenvolvido. Neste caso, um modelo de qualidade próprio é definido, baseado em modelos de qualidade e teorias já existentes.

No presente trabalho, utilizou-se o modelo de qualidade QMOOD (*Quality Model Object Oriented Design*) (Bansiya & Davis, 2000), pois ele se baseia nas características do *design* OO para estimar os atributos da qualidade que são de interesse para os usuários.

2.3.1.1. Modelo de Qualidade QMOOD

O modelo QMOOD (Bansiya & Davis, 2000) é uma abordagem para a estimativa e avaliação dos atributos da qualidade em um *design* OO. A estrutura deste modelo, igual à maioria dos modelos da qualidade já desenvolvidos, tem forma hierárquica ou de árvore. Níveis superiores representam os atributos externos da qualidade do *design* que se deseja avaliar, como a flexibilidade e facilidade de uso. Estes atributos externos estão compostos por um ou mais atributos internos, os quais, no contexto do modelo QMOOD representam as propriedades do *design* orientado a objeto.

Os modelos de qualidade representados em forma hierárquica descrevem os relacionamentos pertinentes entre os atributos externos e os atributos internos que os influenciam. Portanto, os atributos externos poderão ser estimados a partir dos atributos internos pelos quais eles são influenciados (Fenton & Pflegeer, 1997). Sendo assim, o QMOOD é decomposto em quatro níveis, os quais são

PUC-Rio - Certificação Digital Nº 0711290/CA

apresentados na Figura 1**Error! Reference source not found.**. A seguir é apresentada uma descrição dos diferentes níveis.

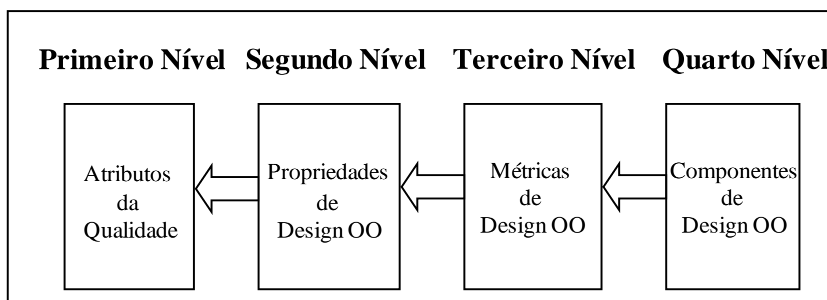

Figura 1 – Níveis e relações no QMOOD.

Atributos da Qualidade

No primeiro nível estão os atributos externos da qualidade considerado no modelo QMOOD. Estes atributos se baseiam no conjunto de atributos da norma ISO 9126. Como esta é uma norma definida para avaliar a qualidade de um produto de software, alguns de seus atributos, pela sua natureza, não podem ser quantificados utilizando apenas as informações disponíveis no *design*. Os atributos de qualidade escolhidos por Bansiya & Davis são: funcionalidade, eficácia, facilidade de compreensão, facilidade de extensão, usabilidade e flexibilidade.

Propriedades de *Design* OO

As propriedades de *design* são conceitos tangíveis que podem ser diretamente avaliados examinando a estrutura interna e externa e as relações dos componentes do *design* tais como: classes, métodos e atributos. Ou seja, são os atributos internos.

No QMOOD, foram utilizadas algumas propriedades da abordagem estruturada tais como: acoplamento, coesão, complexidade, tamanho para estimar os atributos da qualidade da abordagem orientada a objetos. Também foram utilizadas propriedades exclusivas de *design* OO tais como: herança, polimorfismo, composição. Sendo assim, as propriedades do *design* OO utilizadas no QMOOD são: tamanho, hierarquia, coesão, abstração, acoplamento, herança, polimorfismo, composição, troca de mensagens entre classes e complexidade.

PUC-Rio - Certificação Digital Nº 0711290/CA

Métricas de *Design* OO

Cada uma das propriedades de *design* utilizadas no modelo QMOOD representa características do *design* que podem ser objetivamente quantificadas mediante o uso das métricas.

Tabela 1 – Métricas do modelo QMOOD.

Métrica	Nome	Descrição
DSC	Número de classes no *design*	Conta o número total de classes no *design*
NOH	Numero de Hierarquias	Conta a quantidade de hierarquias de classes no *design*
ANA	Média de Ancestrais	Representa a média de classes das quais a classe avaliada herda informação. É calculado utilizando o número total de classes na estrutura de herança.
DAM	Métrica de acesso a dados	Calcula a proporção que representam os atributos privados (protegidos) do total de atributos da classe (faixa 0-1)
DCC	Acoplamento direto entre classes	Conta a quantidade de classes com as quais a classe avaliada está direitamente relacionada. Na contagem, são incluídas as declarações de atributos e parâmetros.
CAM	Coesão entre os métodos da classe	Representa o grau de relação entre os métodos de uma classe baseada nos parâmetros dos métodos da classe. Primeiramente é calculado o conjunto máximo independente dos tipos dos parâmetros de todos os métodos. Depois é computada a soma da interseção dos tipos dos parâmetros de cada um dos métodos com o conjunto independente. Finalmente, a soma é divida pelo número total de parâmetros multiplicado pela quantidade de métodos.
MOA	Medida de agregação	Representa a composição de uma classe utilizando os atributos declarados. É computada contando o número de atributos cujo tipo é definido pelo usuário.
MFA	Abstração funcional	Calcula a proporção que representam os métodos herdados do total de métodos da classe (faixa 0-1)
NOP	Numero de métodos polimórficos	Conta a quantidade de métodos que podem ter comportamento polimórfico (e.x. em Java, os métodos não declarados como final).
CIS	Tamanho da interface de uma classe	Conta a quantidade de métodos públicos que a classe possui. Não são considerados nesta métrica os atributos públicos.
NOM	Quantidade de métodos	Conta a quantidade de métodos declarados na classe.

O modelo utiliza um conjunto de métricas já disponibilizadas na literatura (Bansiya, 1997) e algumas métricas novas. Bansiya & Davis (2001) propuseram

PUC-Rio - Certificação Digital Nº 0711290/CA

métricas a fim de quantificar propriedades de *design* estruturado tais como: complexidade, acoplamento e coesão utilizando exclusivamente informações de *design* OO. Alguns pesquisadores (Briand et al, 1998) criticaram a medição da coesão baseada na contagem de propriedades sintáticas. De acordo com eles, existem informações relevantes, relacionadas à semântica, que influenciam na coesão das entidades. Contudo, entendemos esta questão como uma limitação que deve ser considerada nas medições da coesão baseadas na sintaxe. O conjunto de métricas utilizadas no modelo é apresentado na Tabela 1.

Componentes de *Design* OO

No modelo QMOOD são consideradas como componentes do *design* OO os seguintes elementos: classes, atributos, métodos, relações entre classes como a composição, a herança, as hierarquias de classes.

O modelo reduz algumas das limitações que foram apontadas em modelos já desenvolvidos como o *Factor Criteria Metrics* (FCM) (McCall, 1977). Uma dessas limitações é o pouco detalhamento na relação entre as métricas e atributos. Para reduzir tal limitação, QMOOD torna explícitas as relações entre cada um dos níveis, oferecendo tabelas de mapeamento e equações (no caso da correspondência entre os atributos da qualidade e as propriedades do *design*). Por exemplo, cada propriedade do *design* no segundo nível é relacionada com a métrica do terceiro nível pela qual ela é quantificada (Tabela 2).

A correspondência entre o primeiro e segundo níveis é ainda mais explícita, pois cada atributo da qualidade é o resultado da soma do valor de cada uma das propriedades do *design* associadas a ele. Por exemplo, Bansiya & Davis estimam o atributo “Facilidade de extensão” mediante a associação das seguintes propriedades do *design*: acoplamento, coesão, comunicação e tamanho:

*Facilidade de extensão = -0.25*acoplamento + 0.25*coesão + 0.25*trocademensagens + 0.5*tamanho.*

em que os pesos podem ser negativos ou positivos dependendo da relevância das propriedades do *design* em relação ao atributo da qualidade. Os autores informam que estes pesos têm sido calculados de maneira intuitiva por eles.

PUC-Rio - Certificação Digital Nº 0711290/CA

Tabela 2 – Relações entre as métricas e as propriedades do *design*.

Propriedade do *design*	**Métrica do *design***
Tamanho do *design*	Número de classes no *design* (DSC)
Hierarquias	Número de Hierarquias de classes
Abstração	Média de Ancestrais ou super-classes (ANA)
Encapsulamento	Métrica de acesso a dados (DAM)
Acoplamento	Acoplamento direto entre classes (DCC)
Coesão	Coesão entre os métodos da classe (CAM)
Composição	Medida de Agregação (MOA)
Herança	Medida de Abstração funcional (MFA)
Polimorfismo	Numero de métodos polimórficos (NOP)
Troca de mensagens	Tamanho da interface de uma classe (CIS)
Complexidade	Numero de métodos (NOM)

O modelo QMOOD oferece flexibilidade para cada um de seus níveis em relação ao domínio do sistema a ser avaliado. Mediante o uso dos pesos se permite aumentar ou diminuir a importância das propriedades do *design* nos atributos de qualidade, com o objetivo de refletir de uma melhor maneira a qualidade desejada. No segundo nível, novas propriedades do *design* podem ser incluídas para avaliar atributos da qualidade. Finalmente, no terceiro nível as métricas podem ser modificadas.

Embora o modelo tenha reduzido pontos negativos apontados nos modelos FCM, QMOOD tem algumas limitações. Uma delas é que o modelo apenas se preocupa em fornecer valores para os atributos de qualidade, mas não se preocupa em apontar os elementos do *design* responsáveis por valores ruins eventualmente obtidos para os atributos. Por causa disso, se torna mais difícil realizar uma transformação eficiente no *design* para melhorar a sua qualidade. O desenvolvedor é informado apenas do problema, devendo diagnosticar a verdadeira causa por conta própria, além de procurar um modo para eliminar o problema e, desta forma, melhorar o *design*. Por exemplo, uma classe com nível elevado de complexidade e acoplamento, e baixa coesão entre seus métodos é uma classe que pode influenciar negativamente a facilidade de extensão do sistema (Meyer, 1997).

PUC-Rio - Certificação Digital Nº 0711290/CA

2.3.2. Estratégias para a Detecção de Problemas de *Design*

Apesar de ser um mecanismo poderoso para avaliar e controlar a qualidade do *design* de software, métricas de software apresentam uma limitação intrínseca: é difícil obter interpretações adequadas a partir de medições de *design* (Marinescu, 2004). Esta limitação está presente também no modelo da qualidade QMOOD já que ele oferece como resultado valores para os atributos externos de qualidade, ignorando os elementos do *design* causadores das anomalias reportadas. A limitação anterior gera um impacto negativo na relevância dos resultados de medições (Marinescu, 2004; Lanza & Marinescu, 2006).

Para tratar essa limitação das métricas, alguns pesquisadores propuseram mecanismos para formular regras baseadas em métricas que capturam desvios dos princípios e heurísticas do bom *design* OO (Sahraoui et al, 2001; Martin, 2002, Emden & Moonen, 2002; Marinescu, 2004; Munro, 2005; Lanza & Marinescu, 2006). Marinescu (2004) chama esse mecanismo de *estratégias de detecção*. Estratégias de detecção ajudam os desenvolvedores a detectar e localizar problemas de *design* de um software.

No contexto deste trabalho, esse mecanismo de análise foi incluído visando alcançar conjuntamente com o modelo QMOOD uma avaliação mais completa da qualidade do *design*. Utilizando os dois mecanismos, se terá a estimativa dos atributos externos da qualidade e a detecção/localização de possíveis entidades do *design* causadoras dos valores não desejados dos atributos estimados.

Uma estratégia de detecção é uma condição lógica composta por métricas que detecta elementos do *design* com problemas específicos. Por meio do uso de estratégias de detecção, o desenvolvedor pode localizar diretamente classes e métodos afetados por um problema de *design* particular, em vez de ter que inferir o problema a partir de um extenso conjunto de valores anormais de métricas. As estratégias de detecção são baseadas também em outros dois mecanismos: filtragem e composição.

O objetivo dos filtros é reduzir consideravelmente o conjunto de dados a ser analisado. O conjunto resultante é composto apenas por aqueles elementos do

PUC-Rio - Certificação Digital Nº 0711290/CA

design para os quais as métricas reportam valores anômalos. Os filtros podem ser classificados em:

- **Marginais**: são aqueles filtros em que um dos limites (inferior ou superior) do conjunto resultante é implicitamente identificado com o valor limite especificado na definição.
- **Intervalos**: são aqueles filtros em que os limites (inferior e superior) do conjunto resultante são explicitamente especificados na definição.

A Tabela 3 apresenta a estrutura destes filtros contendo alguns exemplos.

Tabela 3 – Classificação dos filtros para dados.

Filtro de Dado	Limites		Exemplo
Marginal	Semântico	Relativo	TopValues(20%) BottomValues(5%)
		Absoluto	HigherThan(3) LowerThan(10)
	Estatísticos		Box-Plot[1]
Filtro de Dado	**Especificação**		**Exemplo**
Intervalo	Composição de dois filtros marginais com polaridades opostas.		Between(5,9):=HigherThan(5)^ LowerThan(9)

Visando uma boa utilização dos filtros na especificação de uma estratégia de detecção, Marinescu oferece um conjunto de regras que ajudam ao engenheiro a decidir qual filtro deve ser usado. Porém, o mecanismo de filtragem ainda possui a limitação de apoiar a interpretação isolada dos resultados das métricas. Portanto, Marinescu apresenta o mecanismo de composição com o objetivo de apoiar a interpretação correlacionada dos resultados da aplicação de diferentes filtros. Este mecanismo se baseia na utilização de operadores lógicos para combinar os conjuntos resultantes da aplicação dos filtros. A Figura 2 apresenta a relação entre estes mecanismos.

[1] Método estatístico para representar graficamente grupos de dados numéricos. Mediante estes gráficos podem ser detectadas diferenças entre populações sem fazer suposições da distribuição estatística dos dados (Fenton & Pfleeger, 1999).

PUC-Rio - Certificação Digital Nº 0711290/CA

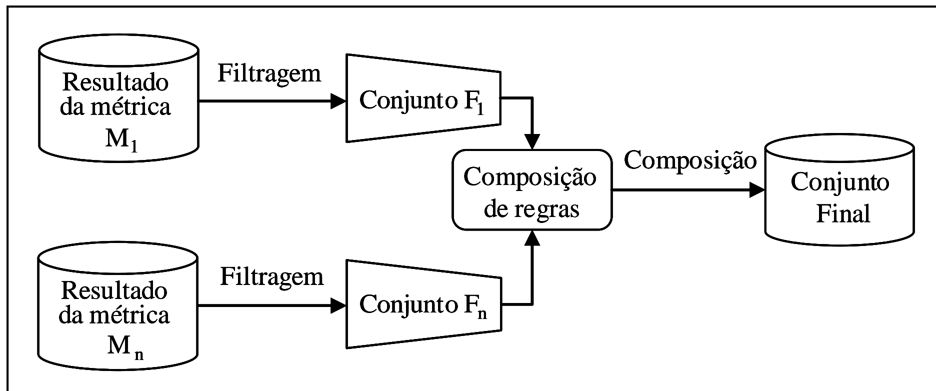

Figura 2 – Mecanismo de filtragem e composição.

O mecanismo de estratégias de detecção pode ser resumido numa seqüencia de cinco passos; (i) identificar as características ou conjunto de sintomas do problema a detectar de modo que eles possam ser capturados por métricas, (ii) seleção das métricas e filtros para quantificar os sintomas identificados, (iii) expressão da estratégia de detecção utilizando os operadores de composição para correlacionar as métricas e filtros (iv) aplicação da estratégia de detecção e (v) exame dos candidatos detectados.

Com o objetivo de mostrar uma estratégia de detecção, a seguir é apresentada uma das estratégias propostas por Marinescu (2002). Esta estratégia tem como objetivo a detecção do problema de modularidade, chamado de *God Class* (Riel, 1996). Este problema corresponde a classes que tentam centralizar a maioria do trabalho de um sistema.

A estratégia de detecção *God Class* é baseada em métricas de acoplamento, coesão e complexidade. Sendo assim, a estratégia é definida por Marinescu (2002) da seguinte forma:

God Class := (WMC, HigherThan(47)) and (ATFD, HigherThan(4)) and (TCC, LowerThan(0,33))

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 definida em (McCabe, 1976) para calcular a complexidade dos métodos). ATFD (*Access To Foreign Data*) conta o número de atributos de outras classes acessados, diretamente ou utilizando métodos de acesso, pela classe avaliada (Marinescu, 2002, 2004; Lanza & Marinescu, 2006). TCC (*Tight Class Cohesion*) conta o número relativo de métodos de uma classe que acessam pelo menos um atributo comum (Marinescu, 2002, 2004; Lanza & Marinescu, 2006).

PUC-Rio - Certificação Digital Nº 0711290/CA

Esta estratégia de detecção diz que uma classe não deve ter WMC maior que 47, ATFD maior que 4 e TCC menor que 3; se não, ela será classificada como uma instância *God Class*. Esta estratégia de detecção é abordada em maior detalhe na Seção 4.2.2.

A aplicação das estratégias de detecção alerta o projetista sobre possíveis problemas causados por *design* incorreto. No entanto, não garante que problemas realmente existem, e que o *design* precisa ser alterado. Portanto, mesmo que as estratégias gerem alerta, o projetista deve analisar o projeto e o código para verificar quais são as razões do alerta. Na verdade, os alertas compreendem informações que ajudam o projetista a se concentrar em certas partes do projeto e do código que são possivelmente problemáticas.

InCode (2008) é um *plugin* para o Eclipse que automatiza algumas das estratégias de detecção propostas em (Lanza & Marinescu, 2006). As estratégias de detecção atualmente automatizadas por *InCode* são aquelas definidas para detectar os seguintes problemas de *design*: *Data Class* (Fowler et al, 1999), *God Class* (Riel, 1996), *Feature Envy* (Fowler et al, 1999) e *Dupplication Code* (Fowler et al, 1999). *InCode* aponta as classes e métodos que possuem tais problemas de *design*. Porém, tem a limitação de permitir apenas a aplicação das estratégias de detecção no código fonte. Sendo assim, os problemas de *design* somente podem ser detectados em estágios avançados do desenvolvimento dos sistemas.

PUC-Rio - Certificação Digital Nº 0711290/CA