3
Estado da Arte e Trabalhos Relacionados

Esse capítulo tem como objetivo disponibilizar um resumo do conteúdo que serviu de referência para realização deste trabalho. Na Seção 3.1 apresentamos o estado da arte em medição de código e na Seção 3.2 o estado da arte em estratégias de detecção. Na Seção 3.3 é feita uma revisão de ferramentas que dão suporte à detecção de anomalias através de estratégias de detecção. Na Seção 3.4 é apresentado um trabalho que propõe a utilização de linguagem específica de domínio para flexibilizar especificações de anomalias e regras de detecção. Na Seção 3.5 é apresentado o contexto de utilização de recursos de visualização de código para apoiar a detecção de anomalias. Ao longo dessas seções algumas limitações das abordagens e das ferramentas de detecção são discutidas.

3.1
Medição de Código

A medição captura informações quantitativas sobre atributos ou propriedades de entidades de software (Fenton e Pfleeger 1997). Entidades também são algumas vezes referenciadas como módulos ou ainda componentes, dentre os quais na orientação a objetos podemos citar classes, métodos ou pacotes. Atributos, por sua vez, são características ou propriedades de uma entidade. Por exemplo, para uma entidade classe, podemos citar como atributos: tamanho em termos de linhas de código, acoplamento com demais classes, dentre outros.

Métricas de software aplicáveis ao código (Chidamber e Kemerer 1994, Henderson-Sellers 1996, Lorenz e Kidd 1994) têm sido utilizadas para capturar e quantificar desvios de requisitos importantes no projeto orientado a objetos, como elevada coesão, baixo acoplamento, modularização de interesses e assim por diante. Elas habitualmente tornam-se necessárias para identificar onde é necessário efetuar melhorias (Fenton e Pfleeger 1997). Para apresentação do estado da arte em métricas de código, elas foram agrupadas em métricas de tamanho, de acoplamento, de coesão, de interesses e de evolução. Breves definições dessas métricas são disponibilizadas no Anexo A desta dissertação.

3.1.1 Métricas de Tamanho

Algumas métricas são utilizadas para identificar a concisão ou tamanho de um sistema. Essas tipicamente contam o número de ocorrência de um determinado módulo, como o número de pacotes ou o número de classes. Elas também podem contar o número de membros de um módulo (Kan 1994), como o Número de Atributos (NOA) (Lorenz e Kidd 1994) ou o Número de Operações (NOO) (Fenton e Pfleeger 1997).

O Número de Linhas de Código (LOC) (Fenton e Pfleeger 1997) (Lorenz e Kidd 1994) é uma das medidas mais simples que pode ser obtida de um software. Entretanto, uma métrica menos influenciada por fatores como linhas em branco, comentários ou estilo de programação seria o Número de Comandos (NOS) (Fenton e Pfleeger 1997). A relevância dessas métricas está no fato de que classes ou métodos muito grandes prejudicam a legibilidade, reusabilidade e manutenibilidade do código (Chidamber e Kemerer 1994, Fowler et al. 1999).

3.1.2 Métricas de Acoplamento

Acoplamento é uma medida do quão fortemente os componentes estão conectados, possuem conhecimento ou dependem uns dos outros (Chidamber e Kemerer 1994). Uma métrica bastante difundida para se medir acoplamento é a Acoplamento entre Objetos (CBO) (Chidamber e Kemerer 1994). Outras avaliam alguns tipos de acoplamento específicos, como os resultantes de heranças. Nesse caso, podemos citar as métricas Profundidade da Árvore de Herança (DIT) (Chidamber e Kemerer 1994) e Número de Filhos (NOC) (Chidamber e Kemerer 1994).

A relevância dessas métricas está no fato de que minimizar o acoplamento entre os componentes é um requisito desejável em sistemas (Selby e Basili 1991). Nos sistemas em que os módulos estão muito acoplados, os impactos possíveis são: (i) a classe é mais difícil de entender isoladamente; (ii) mudanças em quaisquer das classes relacionadas podem forçar mudanças locais à classe e (iii) dificuldade de reúso devido ao excesso de dependências.

3.1.3 Métricas de Coesão

Coesão é uma medida da diversidade das responsabilidades associadas a uma entidade. O fato de uma entidade possuir um alto grau de coesão implica que seus elementos internos, tais como métodos e atributos, estão

fortemente inter-relacionados. A maior parte das métricas propostas se baseia em relacionamentos explícitos entre os membros de uma classe, tais como acesso compartilhado de atributos por métodos internos.

Exemplos clássicos dessas métricas são Perda de Coesão em Métodos (LCOM) (Chidamber e Kemerer 1994) e Força de Coesão entre os Métodos (TCC) (Bieman e Kang 1995). O trabalho de Brian e colegas (1998) apresenta uma coletânea e estudo comparativo contendo apenas métricas de coesão. A relevância dessas métricas está no fato de que um nível baixo de coesão, indica que uma classe pode estar exercendo muitas responsabilidades ao mesmo tempo, o que pode repercurtir em dificuldade de compreensão, de reúso e de manutenção (Chidamber e Kemerer 1994).

3.1.4 Métricas de Separação de Interesses

Um interesse é alguma parte de um problema a ser resolvido por um sistema de software que se deseja tratar como uma unidade conceitual única. O princípio de separação de interesses define que para se dominar a complexidade do desenvolvimento de software, deve-se separar as responsabilidades de forma a se concentrar em apenas uma por vez (Dijkstra 1997). Na progamação orientada a objetos (POO), interesses são modularizados por meio das diferentes abstrações: classes, objetos, métodos e atributos.

Novas métricas têm sido propostas para indicar o nível de espalhamento e de entrelaçamento dos interesses no código (Sant'Anna et al. 2003, Figueiredo 2006). Alguns exemplos são a métrica de Difusão do Interesse por Componentes (CDC) (Sant'Anna et al. 2003) e a que contabiliza o Número de Interesses por Componente (NCC) (Figueiredo et al. 2009). Uma característica comum a essas métricas é que elas necessitam de uma atividade prévia de sombreamento dos interesses (Garcia 2004, Sant'Anna et al. 2003) para serem calculadas. Nessa atividade os elementos de código são associados aos interesses implementados.

3.1.5 Métricas Sensíveis à História

Em (Ratiu et al. 2004) algumas medições sensíveis à história foram propostas. A métrica Estabilidade de uma Medida é tida como o número médio de vezes que uma medida ou propriedade (por exemplo, número de métodos) é alterada ao longo das versões de um sistema. Nesse trabalho, é analisada a variação do número de métodos ao longo das evoluções de um sistema para avaliar a instabilidade de uma classe. Entretanto, sabemos que que duas ver-

sões distintas podem ter o mesmo número de métodos e, contuto, terem sofrido alterações internas a esses métodos.

Quando a métrica avalia a instabilidade com base no número de métodos tem-se a $\text{NOM}_{stability}$. Já quando é utilizado o número de linhas de código, tem-se a medida $\text{LOC}_{stability}$. Outra métrica mencionada neste trabalho é a Persistência de uma Anomalia que contabiliza o número médio de vezes que um mesmo módulo foi detectado com a mesma anomalia. Em sistemas de controle de versão é comum obtermos como medidas sensíveis à história: número de componentes adicionados, alterados, removidos, dentre outros.

3.1.6 Ferramentas de Suporte a Métricas

Sabemos que um dos requisitos indispensáveis no uso de métricas para avaliação de código é a disponibilização de ferramentas que automatizem seus cálculos. Por esse motivo optou-se por apresentar na Tabela 3.1 uma associação entre as métricas citadas e exemplos de ferramentas que dão suporte a tais cálculos. Não é escopo desse trabalho fazer um estudo comparativo entre ferramentas de métricas, pois nosso interesse principal está no âmbito de ferramentas de detecção (Seção 3.3). Por tal motivo, não apresentaremos características das ferramentas citadas.

Como podemos observar na Tabela 3.1, nenhuma das ferramentas apresentadas fornece suporte à métricas sensíveis à história. Além disso, destacamos que, dentre as apresentadas, apenas o Eclipse Metrics Plugin possui licensa gratuita. Tal fato dificulta bastante a utilização das demais ferramentas. Outra consideração é que dentre as ferramentas investigadas a Together foi a que se apresentou mais completa em relação ao número de métricas convencionais suportadas. Esse resultado fez com ela fosse a selecionada para gerar resultados utilizados pela ferramenta proposta (Capítulo 6).

3.2 Estratégias de Detecção

Apesar do uso extensivo de métricas (Chidamber e Kemerer 1994, Henderson-Sellers 1996), quando consideradas isoladamente, elas fornecem informações de granularidade muito fina e insuficientes para o reconhecimento de anomalias (Lanza e Marinescu 2006). Para contornar tal limitação, pesquisadores (Marinescu 2004)(Lanza e Marinescu 2006) propuseram as chamadas *estratégias de detecção*. Uma estratégia de detecção é uma condição lógica composta por métricas que detecta módulos do código em conformidade com tal condição. Por meio do uso dessas estratégias, o desenvolvedor pode localizar

Tabela 3.1: Ferramentas de suporte a métricas de código

Grupo	Métrica	Together [1]	Eclipse Metris Plugin [2]	Analyst4j [3]	JHawk [4]	Understand [5]	Ajato [6]
Tamanho	LOC (Fenton e Pfleeger 1997)	x	x	x		x	x
	NOS (Fenton e Pfleeger 1997)				x	x	x
	NOM (Li e Henry 1993)	x	x		x	x	x
	NOA (Lorenz e Kidd 1994)	x	x	x	x		x
Acoplamento e Hierarquia	CBO (Chidamber e Kemerer 1994)	x		x	x	x	
	DIT (Chidamber e Kemerer 1994)	x	x	x	x	x	x
	NOC (Chidamber e Kemerer 1994)	x	x	x			x
	MPC (Li e Henry 1993)						
Coesão	LCOM (Chidamber e Kemerer 1994)	x	x	x	x	x	
	TCC (Bieman e Kang 1995)	x			x		
Interesses	CDC (Sant'Anna et al. 2003)						x
	NCC (Sant'Anna et al. 2003)						
	LOCconcern (Figueiredo et al. 2009)						
	NOAconcern (Figueiredo et al. 2009)						x
	NOOconcern (Figueiredo et al. 2009)						x
História ou Evolução	LOCstability (Ratiu 2003)						
	NOMstability (Ratiu 2003)						

[1] http://www.borland.com/br/products/together/
[2] http://metrics.sourceforge.net/
[3] http://www.codeswat.com/
[4] http://www.virtualmachinery.com/jhawkprod.htm
[5] http://www.scitools.com/index.php
[6] http://www.teccomm.les.inf.puc-rio.br/emagno/ajato/

diretamente classes e métodos afetados por uma anomalia de modularidade particular. Isso é possível sem que ele tenha que inferir o problema a partir de um extenso conjunto de valores anormais de métricas.

As detecções são realizadas a partir de dois processos denominados *filtragem* e *composição* (Marinescu 2004). A filtragem reduz o conjunto inicial dos dados e dá suporte à avaliação de uma métrica, de forma isolada. Dessa forma, filtros capturam sintomas ou características de uma dada anomalia. Por exemplo, na detecção de anomalias presentes em classes, "$LOC > 250$" define um filtro do tipo *absoluto* (Marinescu 2004) para recuperar todas as classes que possuem mais de 250 linhas. Já "$LOC,\ TopValues(25\%)$" define um filtro do tipo *relativo* (Marinescu 2004) que recupera 25% das classes com os maiores

números de linhas de código. Como pudemos observar, filtros absolutos se diferem de relativos pois possibilitam a avaliação da métricas de uma entidade, independente dos resultados das métricas das demais entidades.

O processo de composição tem por objetivo apoiar a interpretação correlacionada dos resultados de diferentes filtros. Isso é feito através da utilização de operadores de composição: "E" e "OU". Por exemplo, "$(LOC,\ TopValues(25\%))\ E\ (LOC > 100)$", define a composição dos dois filtros anteriormente mencionados. Os dados numéricos (por exemplo, 25% e 100) considerados em cada filtragem e responsáveis por delimitar os valores de métricas são chamados de valores limites. Limites superiores e inferiores podem ser especificados aos filtros de uma estratégia de detecção. Um valor de métrica dentro dos limites estabelecidos por um filtro é que chamamos de um valor anômalo para aquela métrica. A Figura 3.1 apresenta, de forma geral, a relação entre os processos de filtragem e composição.

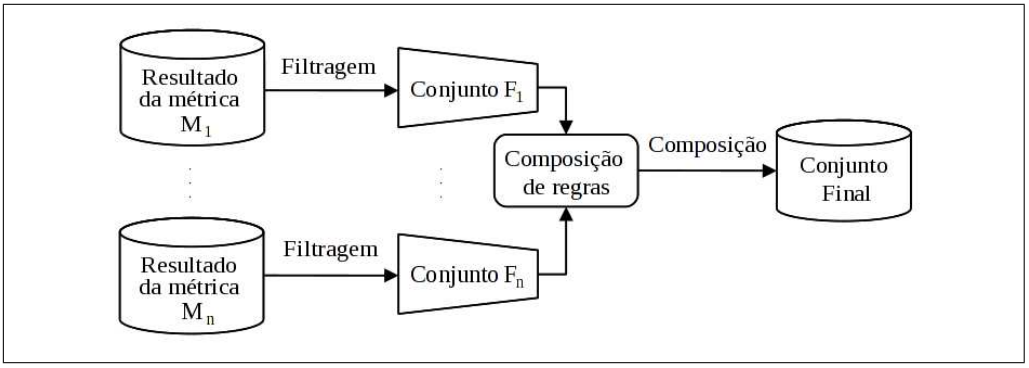


Figura 3.1: Processos de filtragem e composição pertencentes a estratégias de detecção (Figura adaptada de (Marinescu 2004)).

As métricas M_1 e M_n consideradas na Figura 3.1 visam capturar características ou sintomas da anomalia que se deseja detectar. Como podemos notar nesta figura, a primeira filtragem apresentada, ou filtragem superior, dá suporte à avaliação dos resultados da métrica M_1. De forma semelhante, a filtragem inferior dá suporte a avaliação da métrica isolada M_n. O conjunto resultante de cada filtragem, *Conjunto* F_1 *e Conjunto* F_n é composto apenas pelos módulos para os quais as métricas M_1 e M_n reportam valores anômalos. Por isso, a etapa de filtragem também é conhecida como etapa de redução de dados. Para concluir o mecanismo de detecção, a composição correlaciona os resultados dos filtros de acordo com o operador de composição considerado, resultando no *Conjunto Final*. O *Conjunto Final* é a representação dos módulos que foram detectados.

3.2.1 Algoritmo de Definição de Estratégias de Detecção

De acordo com (Marinescu 2004), são quatro os passos a serem seguidos na definição de uma estratégia de detecção:

Passo 1 - Identificar sintomas: devem ser identificados o conjunto de sintomas ou características do problema que se deseja detectar (e.g., classe grande, com alto acoplamento, etc).

Passo 2 - Selecionar métricas: devem ser selecionadas as métricas que quantifiquem os sintomas identificados para o problema em questão.

Passo 3 - Identificar filtros: são definidos os filtros que serão associados a cada métrica selecionada para apoiar a detecção do problema.

Passo 4 - Compor estratégia de detecção: enfim, é formalizada a expressão de detecção utilizando os operadores de composição para correlacionar as métricas e filtros selecionados.

A seguir é apresentado um exemplo de estratégia de detecção para a anomalia *God Class* proposta em (Marinescu 2004). Ilustramos como os passos apresentados são aplicados para este caso.

Exemplo de Estratégia Convencional de Detecção

Para detectar GCs, por exemplo, alguns pesquisadores (Marinescu 2004) formularam uma estratégia que visa capturar classes com os seguintes sintomas: (a) acessa vários dados de classes que possuem poucas funcionalidades, (b) possui elevada complexidade e, por fim, (c) apresenta baixa coesão. Para quantificar os sintomas (a), (b) e (c), Marinescu utiliza as seguintes métricas: ATFD (Acesso a Dados Estrangeiros) (Marinescu 2004), WMC (Soma das Complexidades dos Métodos) (Marinescu 2004) e TCC (Intensidade da Coesão da Classe) (Marinescu 2004), respectivamente. A partir dessas métricas, a seguinte estratégia convencional foi definida (Marinescu 2004).

$$GodClass := ((ATFD > 1)\ E\ (WMC,\ TopValues(25\%))\ E\ (TCC,\ BottomValues(25\%)) \quad (3\text{-}1)$$

De acordo com a estratégia da Equação 3-1, serão filtradas: classes que apresentam valores de ATFD maiores que 1; 25% das classes com os maiores valores de WMC; e 25% das classes com os menores valores de TCC. Como o operador de composição é do tipo "E", todas as classes em conformidade com os três filtros são recuperadas e apresentadas como possíveis *God Classes*.

3.2.2 Limitações das Estratégias Convencionais de Detecção

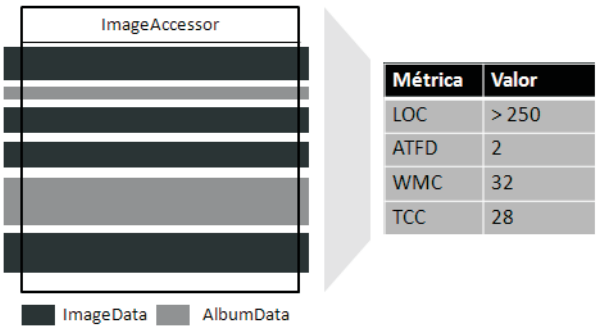
Não é difícil encontrar trabalhos que relatem sobre o considerável número de falsos positivos e negativos gerados por estratégias convencionais de detecção (Ducasse et al. 2004, Ratiu et al. 2004, Figueiredo et al. 2009). Quando há grande incidência de falhas nas detecções prejudica-se a obtenção dos benefícios propostos pela abordagem. Por exemplo, uma estratégia que notifica o usuário sobre a inexistência de problemas, mesmo quando esses existem, pode trazer ao desenvolvedor a falsa ilusão de que o seu código não precisa ser revisto.

Em outro extremo, quando não existem problemas no código, mas são apresentadas ao desenvolvedor listas gigantescas de detecção isso faz com o que o mecanismo perca credibilidade, além de tornar o processo de validação tedioso e cansativo. Ou seja, de acordo com a eficácia das estratégias de detecção, elas podem ajudar ou prejudicar muitos desenvolvedores ou inspetores de código.

Para ilustrar um exemplo de limitação de uma estratégia convencional, avaliaremos o resultado da estratégia de Marinescu, apresentada pela Equação 3-1, na página 33. Tal estratégia não considera a análise do histórico das classes para realizar as detecções. De fato, no estado da arte, estratégias de detecção (Marinescu 2004)(Lanza e Marinescu 2006) apresentam tal característica, agnósticas ao histórico, independentemente da anomalia a ser detectada.

A Figura 3.2 ilustra um caso de GC, a classe `ImageAccessor` presente em uma aplicação chamada Mobile Media (Figueiredo et al., 2008). Tal classe possui mais de 250 linhas de código e foi projetada para prover as funcionalidades de persistência de todos os objetos de domínio da aplicação. Isto viola a idéia que uma classe deveria capturar uma e somente uma abstração. Através de análise minuciosa do código também é possível observar o entrelaçamento de diferentes responsabilidades na classe. Na Figura 3.2, os dois diferentes tons de cinza ilustram o entrelaçamento de código relativo a operações de persistência dos objetos de negócio: `ImageData` e `AlbumData`.

`ImageAccessor` apresenta os seguintes valores de métricas utilizadas pela estratégia convencional: ATFD = 2, WMC = 32 e TCC = 28 (Figura 3.2). Aplicando a estratégia representada pela Expressão 3-1, ela não é detectada como GC pois não possui um dos menores valores de coesão do sistema. O uso de filtros relativos, como o utilizado para avaliar TCC, contribui frequentemente com a ocorrência de falsos positivos e negativos. Isso porque a avaliação de um determinado módulo fica muito dependente dos resultados das métricas dos demais módulos. Além disso, a inclusão ou remoção de outros módulos acaba interferindo diretamente no resultado do módulo avaliado.



Métrica	Valor
LOC	> 250
ATFD	2
WMC	32
TCC	28

Figura 3.2: Exemplo de *God Class*: classe `ImageAccessor`. Os valore de métricas apresentados à direita foram obtidos com base na utilização da ferramenta Together (Together 2009).

Uma outra estratégia convencional (Lanza e Marinescu 2006) foi testada para detectar essa mesma anomalia. Apesar dessa segunda estratégia apresentar apenas filtros do tipo absoluto, novamente o resultado foi contraproducente.

Diversos fatores poderiam ser investigados de forma a identificar os motivos pelos quais as estratégias convencionais têm falhado com tanta frequência. Uma possível limitação encontrada nessas estratégias é que as propriedades de evolução do sistema não são consideradas. Por exemplo, `ImageAccessor` de fato não possui um dos menores valores de coesão do sistema, mas talvez fosse importante observar que sua coesão veio decrescendo gradativamente ao longo de sua história. Informações sensíveis à história como essa poderiam fornecer algum suporte para auxiliar nas detecções. Entretanto, embora o desenvolvimento de software seja cada vez mais incremental, estratégias de detecção atuais não consideram a evolução das características dos módulos. Portanto elas não conseguem considerar o decaimento ou crescimento de propriedades importantes como coesão, acoplamento ou outras.

3.2.3 Notações

Uma análise das notações utilizadas para se representar estratégias de detecção nos ajuda a identificar uma notação conveniente para a representação das estratégias sensíveis à história. Para exemplificar os tipos de notações mais utilizadas tomamos como exemplo a estratégia de detecção definida recentemente em (Lanza e Marinescu 2006) para detectar *God Classes*. Tal estratégia faz uso das mesmas métricas de acoplamento, complexidade e coesão citadas na Equação 3-1. Ao final da seção, discutimos comparativamente tais notações e também justificamos a escolha daquela que servirá de referência para as estratégias sensíveis à história apresentadas em nosso trabalho.

Notação de Marinescu (2002)

(Marinescu 2002) apresenta estratégias de detecção sob a forma de expressões baseadas em uma linguagem descritiva, chamada SOD[1]. Dessa forma, é necessário que se conheça os elementos pertencentes à gramática da linguagem para representação das estratégias. Operadores de comparação são representados por nomes de funções e os valores limites como parâmetros. A Equação 3-2 exemplifica o uso da notação referida.

$$GodClass := (ATFD, HigherThan(4)) \text{ and } (WMC, HigherThan(20)) \text{ and } (TCC, LowerThan(0.33)) \quad (3\text{-}2)$$

Notação de Figueiredo e colegas (2009)

Figueiredo (Figueiredo et al. 2009) e Sant'Anna (Sant'Anna et al. 2007), que apresentam estratégias de detecção de problemas de modularização de interesses, expressam as regras como um comando condicional baseado em (Tekinerdogan e Aksit 1998). As regras seguem a seguinte estrutura:

$$\textbf{Se} < \text{condição} > \textbf{Então} < \text{consequência} > \quad (3\text{-}3)$$

A condição citada na expressão envolve uma combinação de métricas e valores limites, a qual representa os processos de filtragem e composição (Seção 3.2). Já a consequência se apresenta como o tipo de problema a ser detectado. Uma detecção de *God Class* de acordo com essa notação é apresentada pela Equação 3-4:

$$\textbf{Se } (ATFD > 4) \text{ e } (WMC \geq 47) \text{ e } (TCC < 0,33) \textbf{ Então } GOD\ CLASS \quad (3\text{-}4)$$

Notação de Lanza e Marinescu (2006)

Como pode ser visto na Figura 5.1, (Lanza e Marinescu 2006) ao invés de utilizarem uma expressão para representar uma estratégia de detecção, decidem criar uma notação gráfica baseada na representação de circuitos lógicos. Essa notação também é utilizada por (Macia et al 2008) na definição de estratégias aplicáveis à diagramas de classes.

Nessa representação, como mostra a Figura 5.1 os operadores de composição são representados pelas portas lógicas "AND" e "OR". As filtragens são expressas por retângulos que, além de conter as respectivas métricas, e

[1] **S**trategy of **D**etection

valores limites, também contêm, textualmente, as características ou sintomas considerados por cada filtro.

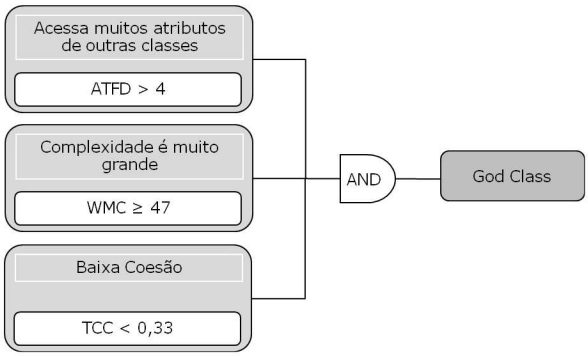


Figura 3.3: Notação para estratégia de detecção segundo (Lanza et al., 2006)

Discussões sobre as Notações da Literatura

A representação escolhida por (Lanza e Marinescu 2006) possui como vantagem a disponibilização tanto das métricas, limites e operadores de composição quanto do significado relacionado a cada filtragem. Já as expressões, das outras duas notações, possuem a limitação de apresentar apenas a combinação das métricas, sem deixar claro a semântica de cada filtragem.

Além disso, no caso da notação apresentada em (Marinescu 2002), apenas os elementos da linguagem SOD podem ser utilizados – o que faz com que seja necessário consultar a gramática da linguagem para conhecer seu poder de expressão. Por exemplo, operadores de comparação tão usuais da matemática, como "maior que" ou "menor que" não são reaproveitados através dos elementos de representação clássicos "$>$" e "$<$".

Na notação de (Lanza e Marinescu 2006), a explicação textual sobre o que captura cada filtro faz com que o leitor saiba exatamente o que está sendo considerado por cada métrica, mesmo que ele desconheça as siglas utilizadas. Contudo, a desvantagem dessa notação baseada em circuitos é que ela acaba demandando maior espaço e trabalho para representar cada estratégia em detrimento ao uso de expressões. Em nosso trabalho optamos pela utilização da notação de (Lanza e Marinescu 2006) pois consideramos importante a explicação dos sintomas capturados por cada filtro - o que aumenta o nível de abstração relacionado à compreensão das estratégias.

3.3 Ferramentas de Suporte a Estratégias de Detecção

Nessa seção, será feita uma breve apresentação e comparação entre ferramentas que dão suporte ao mecanismo de estratégias de detecção. As ferra-

mentas estudadas foram: Together (Together 2009), iPlasma (iPlasma 2009), inCode (inCode 2009) e inFusion (inFusion 2009). Os seguintes critérios sobre a flexibilidade de uso das ferramentas foram investigados:

1. Permite criar/alterar estratégias
2. Permite visualizar os elementos que integram as estratégias (métricas, operadores de composição e valores limites)
3. Permite avaliações sensíveis à história
4. Fornece uma linguagem declarativa para especificação de estratégias
5. Permite avaliar diferentes versões do sistema ao mesmo tempo
6. Permite detectar diferentes anomalias ao mesmo tempo
7. Permite salvar os resultados em relatórios
8. É gratuita

Na Seção 3.3.3, é disponibilizada uma tabela comparativa entre as ferramentas considerando esses critérios.

3.3.1 Together

O Together não é simplesmente um sistema de suporte à medições e estratégias de detecção. Tecnicamente, ele é um conjunto de *plugins* desenvolvidos sob a plataforma do Eclipse (Eclipse 2010) e que funciona como uma aplicação *standalone*. As anomalias a serem detectadas são selecionadas pelo usuário, através de uma interface de auditoria apresentada pela Figura 3.4. Além disso, a ferramenta dá suporte à detecção da grande maioria dos problemas de modularidade apresentados em (Fowler et al. 1999).

O Together também apresenta ao usuário, de forma transparente, as métricas e limites utilizados em cada estratégia suportada pela ferramenta. Um ponto negativo é que não é possível customizar tais estratégias, nem mesmo no que diz respeito aos valores limites. Além disso, é uma ferramenta comercial e sua versão de teste é disponibilizada por apenas 15 dias. Trata-se da única, dentre as ferramentas estudadas, que permite executar as estratégias de detecção em mais de uma versão ao mesmo tempo. Contudo, como mais uma de suas limitações, ela não fornece nenhum suporte à análise da evolução do sistema, nem através de métricas e estratégias sensíveis à história, nem através de gráficos.

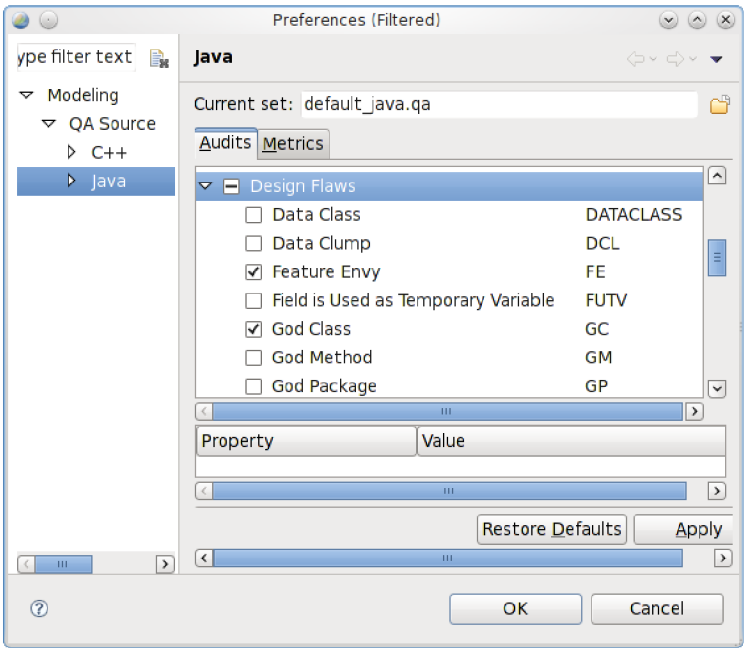


Figura 3.4: Ferramenta Together: suporte a estratégias de detecção

3.3.2 iPlasma, inCode e inFusion

Essas três ferramentas foram desenvolvidas pelo Grupo de Pesquisa LOOSE, liderado por Marinescu e, por isso, elas são discutidas em uma única seção. Tanto a iPlasma quanto a inFusion são ferramentas do tipo *standalone*. Já a inCode é um *plugin* para o Eclipse. Ao utilizar tais ferramentas, a impressão que se tem é que elas são baseadas em um mesmo *framework*, pois apresentam muitos aspectos comuns, como métricas disponíveis, telas, etc. A iPlasma e a inFusion suportam a detecção de um maior número de anomalias que a inCode, cobrindo quase todos os problemas apresentados por (Fowler et al. 1999). Entretanto elas não funcionam de forma integrada a nenhum ambiente de desenvolvimento. Além disso, cada anomalia é detectada separadamente, não havendo a possibilidade de selecionar diferentes anomalias para que sejam detectadas ao mesmo tempo

Já a inCode cobre a detecção de apenas quatro anomalias, a saber: *Data Class* (Fowler et al. 1999), *God Class* (Riel 1996), *Feature Envy* (Fowler et al. 1999) e *Dupplication Code* (Fowler et al. 1999). Uma nova versão do plugin inCode será disponibilizada com a detecção de outras anomalias, porém só poderá ser adquirida através de licença comercial (inCode 2009). Como ponto forte, ela sugere refatorações para os problemas detectados, como mostra a tela apresenta pela Figura 3.5.

Através de uma mesma interface de configuração, apresentada pela Figura 3.6, tanto a iPlasma quanto a inFusion permitem customização de estra-

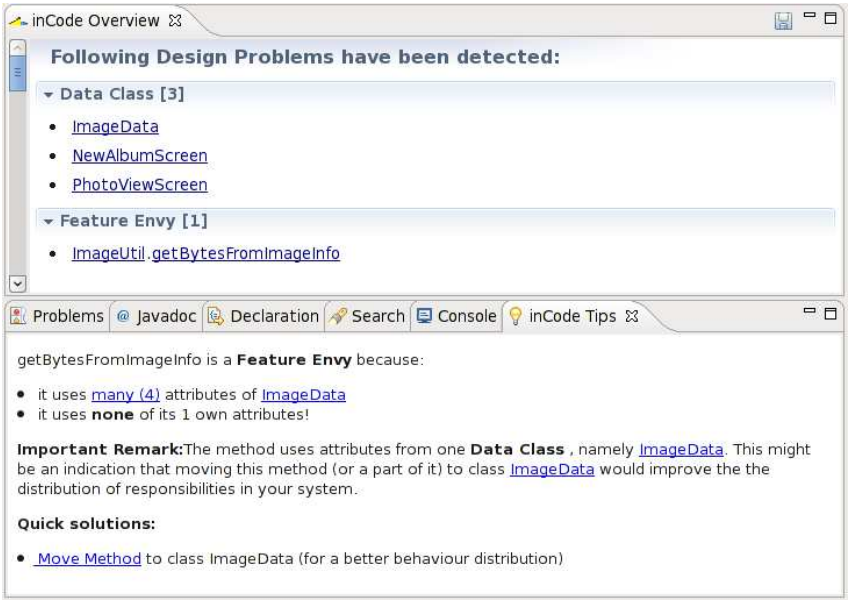


Figura 3.5: inCode: deteção de anomalias e sugestões de refatoração

tégias, porém a customização é limitada por algumas restrições da interface. Exemplos dessas restrições são apresentadas a seguir em discussões da Seção 3.3.3.

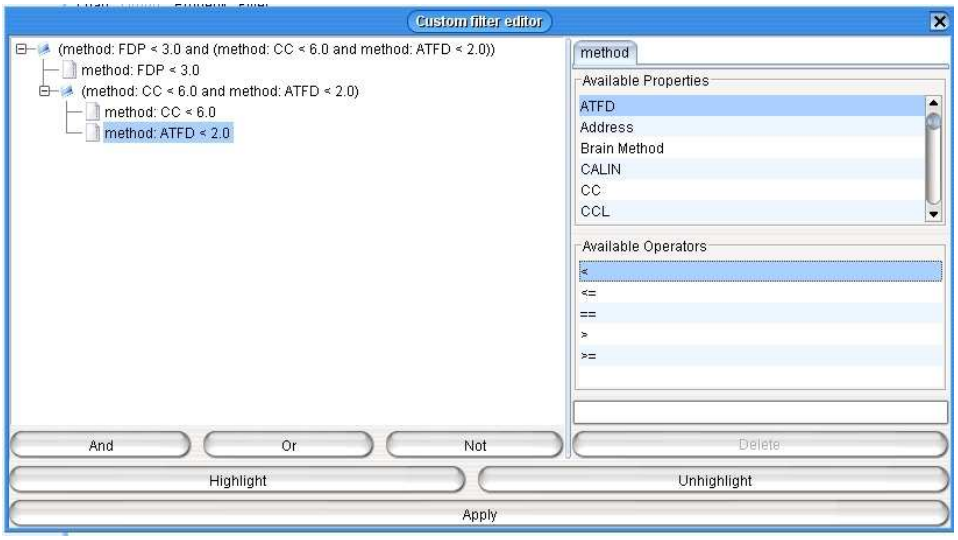


Figura 3.6: iPlasma e inFusion: tela de configuração de estratégias

3.3.3 Considerações sobre as Ferramentas Relacionadas

Como vimos, algumas ferramentas de suporte a estratégias de detecção já vem sendo propostas (Together 2009, inCode 2009, iPlasma 2009, inFusion 2009). Todas fornecem ao usuário informações sobre entidades com certas anomalias de modularidade. Porém, nenhuma delas apresenta flexibili-

dade quanto à criação e alteração de estratégias utilizadas. Elas também não dão suporte ao cálculo de métricas sensíveis à história e à aplicação de estratégias que levem em consideração a história de evolução dos módulos.

Por exemplo, em ferramentas clássicas como o Together (Together 2009) ou mesmo a inCode (inCode 2009), os algoritmos de detecção são definidos em âmbito de código por desenvolvedores, ao invés de poderem ser especificados em alto nível por especialistas no domínio de detecção e avaliação de código. A falta de possibilidade de incluir ou alterar estratégias torna complicado adaptá-las a diferentes contextos ou necessidades de usuários, ou ainda, comparar os resultados de diferentes estratégias. A rigidez dessas ferramentas limita as decisões sobre anomalias, métricas e valores limites considerados, ao grupo de desenvolvedores que as desenvolveu. Sendo assim, nenhuma customização é disponibilizada ao usuário e qualquer necessidade de alteração remete à necessidade de alterações a serem realizadas exclusivamente no código pelos desenvolvedores dessas aplicações.

Outras ferramentas como a iPlasma (iPlasma 2009) e a inFusion (inFusion 2009) até disponibilizam a configuração de estratégias em nível mais elevado, ou seja, à nível de usuário. Porém, (i) as customizações são limitadas pelo uso de elementos e operações disponíveis em uma interface de configuração e (ii) as novas estratégias não podem ser salvas para inspeções futuras. Nessas interfaces de configuração, não se consegue, por exemplo, especificar uma estratégia que contenha a expressão m_1/m_2 ou a expressão $m_1 > m_2$, onde m_1 e m_2 são métricas. A falta de suporte a avaliações que levem em consideração a evolução do código associada à inflexibilidade quanto à adição e alteração de estratégias de detecção, são algumas das limitações deixadas pelas ferramentas existentes.

Com base no estudo considerado, apresentamos a Tabela 3.2, um resumo comparativo entre as ferramentas de detecção mencionadas. Para a comparação, são destacados os critérios selecionados e listados no início da Seção 3.3. Com base nos dados da tabela podemos perceber que nenhuma das ferramentas atende os critérios 1, 3 e 4 cobertos pela ferramenta que propomos neste trabalho. Além disso, também é possível observar que, dentre as apresentadas, a que atende o maior número de critérios, a Together, trata-se de uma ferramenta paga.

3.4

Critério	Together	iPlasma	inCode	inFusion
1. Criação/alteração flexível de estratégias				
2. Permite visulizar estratégias utilizadas	x			
3. Permite avaliações sensíveis à história				
4. Fornece uma linguagem declarativa para especificação das regras				
5. Permite executar estratégias em diferentes versões simultaneamente	x			
6. Permite detectar diferentes anomalias simultaneamente	x			
7. Permite salvar os resultados em relatórios HTML	x	x	x	x
8. Gratuita		x	x	x

Tabela 3.2: Comparação entre ferramentas de suporte a estratégias de detecção

Uma técnica chamada Detecção Experta

DETEX ou Detecção Experta é apresentada em (Moha et al. 2010) como uma instância de uma metodologia chamada DECOR[2] que define passos para a especificação e detecção de anomalias. A principal contribuição desse trabalho, e que está diretamente relacionada ao nosso, é que a tal mecanismo aborda a utilização de uma linguagem específica de domínio para a especificação flexível de anomalias e regras de detecção associadas.

Tal técnica de detecção visa permitir a engenheiros de software especificações de regras em alto nível de forma que algoritmos de detecção possam ser gerados automaticamente a partir de tais especificações, sem a necessidade de codificar classes ou métodos. Tal característica é apresentada como uma das principais contribuições da DETEX. A linguagem utilizada foi chamada de SADSL, do inglês *Software Architectural Defects Specification Language* (Moha et al. 2006) (Linguagem de Especificação de defeitos Arquiteturais de Software).

No contexto da técnica DETEX, as especificações de regras associadas à detecção de anomalias são realizadas em arquivos chamados *Cartões de Regras* (*Rule Cards*), escritos em SADSL. Um Cartão de Regra contém um conjunto de regras que especificam uma determinada anomalia. Ou seja, baseado no cartão de regras que podemos ver na Figura 3.7, eles podem ser entendidos

[2] **DE**tection and **COR**rection

como uma única estratégia de detecção e cada regra que compõe o cartão pode ser equiparada aos filtros utilizados nas estratégias.

```
1  RULE_CARD:SpaghettiCode {
2    RULE:SpaghettiCode
           { INTER LongMethod NoParamete NoInheritance
           NoPolymorphism ProceduralName UseGlobalVariable };
3    RULE:LongMethod       { METRIC LOC_METHOD VERY_HIGH 10.0 };
4    RULE:NoParameter      { METRIC NMNOPARAM VERY_HIGH 5.0 };
5    RULE:NoInheritance    { METRIC DIT 1 0.0 };
6    RULE:NoPolymorphism   { STRUCT NO_POLYMORPHISM };
7    RULE:ProceduralName   { LEXIC CLASS_NAME
                                 (Make, Create, Exec...) };
8    RULE:UseGlobalVariable { STRUCT USE_GLOBAL_VARIABLE };
9  };
```

Figura 3.7: Cartão de regras da abordagem DETEX: especificação de anomalias em alto nível por meio de linguagem específica de domínio (DSL)

A Figura 3.7 apresenta a especificação das regras de detecção de uma anomalia conhecida como *Spaghetti Code* (?). O nome do cartão é o próprio nome da anomalia a ser detectada. No cartão podemos observar que foram especificadas regras de detecção relacionadas ao comprimento dos métodos (RULE:*LongMethod*), número de parâmetros (RULE:*NoParameter*), dentre outras propriedades.

Tal técnica, assim como a ferramenta desenvolvida nesse trabalho, possibilita o que os autores chamam de "Geração de Algoritmo", etapa em que os algoritmos de detecção são automaticamente gerados com base no *parser* das especificações realizadas em alto nível. A DETEX, diferentemente de nossa abordagem, atua sobre modelos obtidos por engenharia direta ou reversa. Já a ferramenta proposta por nós dispensa a necessidade de modelos realizando as detecções a partir do próprio código.

Além disso, nas referências relacionadas, ela é apresentada como uma técnica e não como um ferramenta em que a abordagem de detecção já esteja disponibilizada. Em nenhum dos artigos obeteve-se acesso a alguma tela ou arquivo de configuração que evidenciasse a implementação de uma ferramenta. Outro diferencial é que, mesmo permitindo especificações de alto nível de regras de detecção tal mecanismo não considera as possibilidades de avaliação sensível à história.

3.5 Detecção Baseada em Visualização de Software

O objetivo das ferramentas de visualização de código é fornecer meios fáceis para visualizar o relacionamento entre os componentes do programa, fa-

cilitando a compreensão do sistema. Essas ferramentas simplesmente auxiliam o entendimento do sistema através da abstração de detalhes da implementação. Alguns trabalhos têm proposto o uso de ferramentas de visualização para apoiar a identificação de anomalias de modularidade (Dhambri et al. 2008, Demeyer et al. 1999, Carneiro et al. 2010a). Nesse contexto, Demeyer e colegas (Demeyer et al. 1999) propuseram a ferramenta CodeCrawler e Carneiro e colegas (Carneiro et al. 2010a) apresentaram a ferramenta SourceMiner. Optamos por apresentar maiores informações sobre a SourceMiner, uma vez que tivemos a oportunidade de instalá-la, utilizá-la e de participar de estudos relacionados à sua utilização.

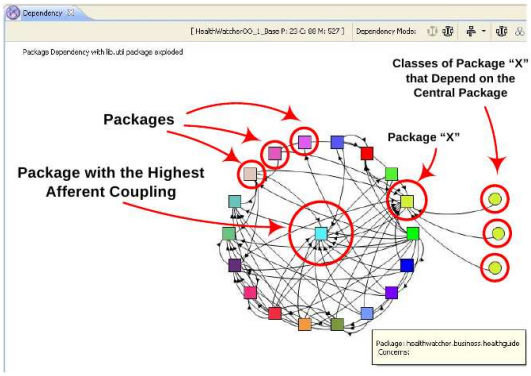
SourceMiner (Carneiro et al. 2009) é uma ferramenta de visualização gratuita, desenvolvida e mantida pelo grupo de pesquisa SoftVis[3] da Universidade Federal da Bahia (UFBA). Ela combina o uso de múltiplas visões (Carneiro et al. 2010a, Carneiro et al. 2010b) com mecanismos importantes de interação com usuário, como filtros de captura dos módulos que se deseja visualizar, dentre outros. Trata-se de um plugin para o Eclipse (Eclipse 2010) que tem como principal objetivo auxiliar programadores na análise e investigação do código fonte em um contexto conhecido como compreensão de sistemas. Cada visão ajuda na análise de propriedades de código relacionadas a conceitos importantes da orientação a objetos, como herança, acoplamento, tamanho, complexidade, e outros. A ferramenta com suas múltiplas visões permite a desenvolvedores o exame de diferentes representações de código-fonte.

Um dos grandes diferenciais da SourceMiner em relação a outras ferramentas de visualição de código é o suporte relacionado a informações sensíveis à interesses (Carneiro et al. 2009). Cores podem ser utilizadas em qualquer uma das visões de forma que se consiga ter uma visão panorâmica de quais módulos implementam os mesmos intereses, ou ainda dos diversos interesses implementados por um mesmo módulo. Apresentamos alguns exemplos de visões disponibilizadas pela SouceMiner. São elas: (1) a visão de dependências, (2) a visão de acoplamento ou *grid coupling*, (3) a visão polimétrica ou de hierarquias e (4) a visão *treemap* ou pacote-classe-método. Apresentamos nessa seção apenas uma visão geral dessas visões. Cada uma delas é explicada em maior grau de detalhes em (Carneiro et al. 2009, Carneiro et al. 2010a)

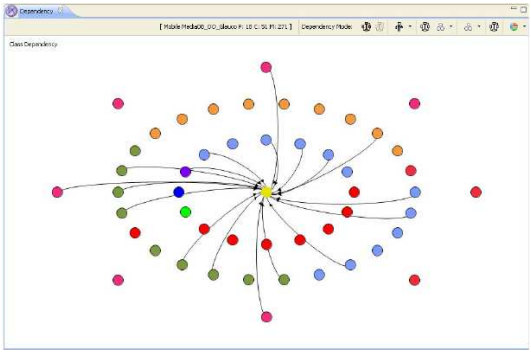
A visão de dependência é responsável por representar as dependências entre pacotes ou entre classes, através de grafos. Como apresenta a Figura 3.8(a), os pacotes são representadas como nós sob a forma de retângulos ao passo que as classes são representadas como nós em forma circular. As dependências são representadas como arestas. Os programadores podem configurar

[3]http://wiki.dcc.ufba.br/SoftVis/WebHome

as cores para representar (i) o pacote que contém uma classe ou (ii) o interesse que afeta uma entidade de software. Já a Figura 3.8(b) apresenta a relação de dependência entre classes de um sistema. A cor de cada nó representa o pacote que a contém.



3.8(a): Visão de Dependências: Pacotes



3.8(b): Visão de Dependências: Classes

Figura 3.8: Visão de dependências entre os módulos – ferramenta Source Miner

Nesta visão também é possível configurar o modo de depência a ser considerado: modo aferente ou eferente. No primeiro caso, as setas representam as depências dos demais módulos ao módulo selecionado. No segundo, as setas representam a dependência do módulo selecionado aos demais módulos. Um outro recurso é a seleção quanto ao significado das cores nos grafos: cores padrão ou cores representando interesses.

Uma outra visão, a visão de acoplamento, se apresenta como uma visão que complementa as informações apresentadas pela visão anterior. Esta visão de acoplamento apresenta a intensidade de acoplamento entre os módulos. Conforme mostra a Figura 3.9, quanto mais próximo está o módulo no centro da figura de qualquer outro módulo apresentado, maior é o nível de dependência entre eles. A Figura 3.9 apresenta um exemplo de tela dessa visão.

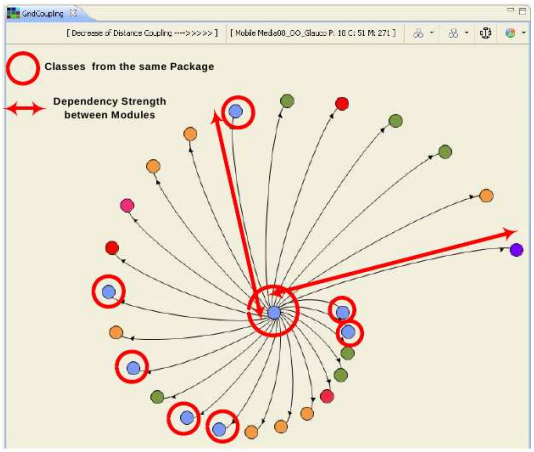
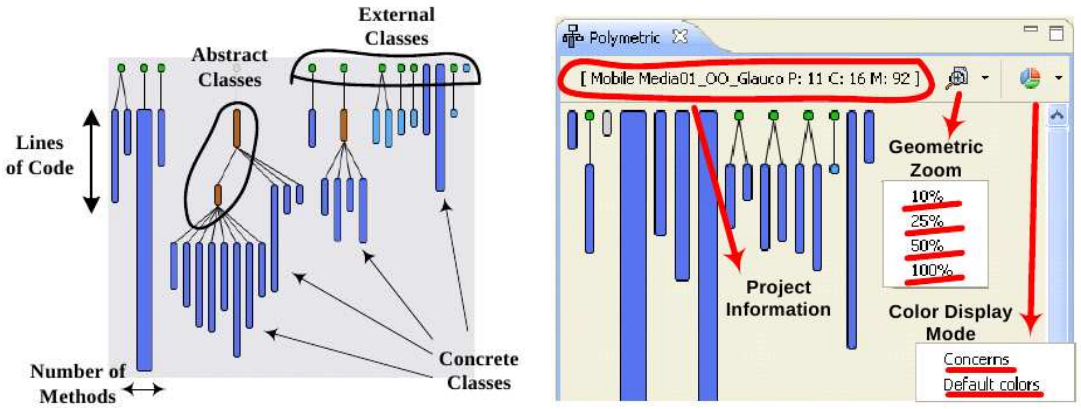


Figura 3.9: Força de acoplamento entre módulos – ferramenta Source Miner

Na visão polimétrica, representada pela Figura 3.10, o comprimento e a largura dos retângulos apresentados são baseados em duas métricas de tamanho. O comprimento representa o Número de Linhas de Código (LOC) de um módulo. Enquanto a largura está associada ao Número de Métodos (NOM). Ou seja, quanto maior a base do retângulo, maior a quantidade de métodos que integram aquele módulo. Quanto maior o comprimento, maior o tamanho em linhas de código.

Na visão polimétrica, cores podem representar informações sensíveis à interesses mas também indicar o tipo de classe considerada na hierarquia, por exemplo uma interface, uma classe abstrata, uma classe concreta, uma classe de API externa ou outras. Essas diferenças são apresentadas na Figura 3.10a.



3.10(a): Visão polimétrica

3.10(b): Visão Polimétrica

Figura 3.10: Visão Polimétrica – ferramenta Source Miner

Por fim, a visão Treemap (Carneiro et al. 2010a) apresenta a organização estrutural dos módulos, por exemplo métodos que integram uma classe, classes que integram um pacote e pacotes que integram o sistema. Por isso, ela também pode ser chamada de estrutura pacote-classe-método. Valores de métricas importantes como de tamanho e complexidade ciclomática também são disponibilizados. Tais medidas influenciam na largura e altura dos retângulos apresentados. Segundo os autores tal perspectiva costuma ser determinante em detecções de anomalias do tipo *God Class*.