UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

ESCOLA DE INFORMÁTICA APLICADA

# Padronização e Auditoria de Projetos

Diogo Avancini Moreno

João Carlos Correia Pena

**Orientador**

Márcio de Oliveira Barros

RIO DE JANEIRO, RJ – BRASIL

AGOSTO DE 2010

Padronização e Auditoria de Projetos.

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação.

Diogo Avancini Moreno
João Carlos Correia Pena

**Orientador**
Márcio de Oliveira Barros

Padronização e Auditoria de Projetos.

Aprovado em ____/_____ /_____

BANCA EXAMINADORA

_____

Prof. Márcio de Oliveira Barros, DSc. (UNIRIO)

_____

Prof. Gleison dos Santos Souza. DSc. (UNIRIO)

_____

Prof. Flávia Maria Santoro DSc. (UNIRIO)

Os autores deste Projeto autorizam a ESCOLA DE INFORMÁTICA APLICADA da UNIRIO a divulgá-lo, no todo ou em parte, resguardando os direitos autorais conforme legislação vigente.

Rio de Janeiro, ____ de _____ de _____

_____

Diogo Avancini Moreno

_____

João Carlos Correia Pena

# Agradecimentos

Agradecemos primeiramente as nossas respectivas famílias pela educação que nos foi proporcionada. Agradecemos também ao professor Márcio Barros, orientador deste projeto, pela paciência e atenção, e à banca examinadora, composta pelos professores Gleison Souza e Flávia Santoro, por terem aceitado participar da avaliação deste projeto.

## RESUMO

Neste trabalho, apresentamos o problema de falta de padrão de código-fonte e arquitetura de projetos de desenvolvimento de software, ocasionado por possíveis falhas ou omissões no caminho que a informação pode ter até chegar ao codificador. Junto a este problema demonstramos o conceito de auditoria em projetos e em código-fonte apresentando como uma possível solução a utilização de ferramentas de auditoria automatizada de código.

A incorporação deste tipo de ferramenta em um projeto pode ser benéfica, pois o trabalho de verificação do código-fonte pode ser feito de maneira automatizada. A ferramenta escolhida foi o FXCop, que já possui diversas regras nativas de verificação de padrões e boas-práticas comuns em *.Net*. No caso de uma empresa possuir padrões próprios de codificação, como, por exemplo, nomenclatura de variáveis ou exigência de documentação em métodos, regras de auditoria para o código produzido podem ser construídas.



**Palavras-chave**: Auditoria, Arquitetura, FXCOP.

## ABSTRACT

In this monograph, it will be presented the problem of lack of source code and software development project architecture pattern, prompted by possible failures or omissions on the path the information takes until it reaches the developer. Along with this problem, it is presented the concept of auditing in projects and source code bringing forth as a possible solution the usage of automated code auditing tools.

The incorporation of this type of tool in a project may be beneficent, for the task of source code verification may be performed in an automated manner. FXCop was the chosen tool, which already possesses several native auditing rules of common patterns and good practices in *.Net*. In case a company has its own encoding patterns, as, for instance, variable nomenclatures or requirement for documentation in methods, auditing rules for the code produced may be developed.



**Keywords**: Auditing, Architecture, FXCOP.

## Índice

## Índice de Figuras

## Índice de Tabelas

# 1 Introdução

Antes da popularização do uso de softwares nas empresas, as aplicações eram projetadas a fim de resolver problemas que se mostravam menores e mais pontuais. Um exemplo comum desse tipo de aplicação é a funcionalidade de cadastro de clientes, que não necessitava de grande capacidade de processamento das máquinas. Outro fator relevante na construção desses softwares era a escassa disponibilidade de desenvolvedores capacitados, considerando a dificuldade de acesso à informação e o estágio prematuro das IDEs disponíveis.

Portanto, as equipes dessa época apresentavam-se menores e mais homogêneas em relação ao seu nível de conhecimento, visto que desenvolver um sistema de informação era uma tarefa que requeria alta qualificação. Com o advento da Internet e, conseqüentemente, um maior alcance à informação, foi possível uma drástica evolução das ferramentas de desenvolvimento, que trouxeram com elas um grande número de facilidades para o desenvolvedor. As equipes passaram a contar com cada vez mais recursos humanos, muitas vezes geográfica e culturalmente distantes, criando um ambiente mais heterogêneo. Esse ambiente é capaz de proporcionar vasta troca de conhecimento, beneficiando os recursos envolvidos, que estão em contato com novas idéias. Por outro lado, com as equipes cada vez maiores, o controle organizado da disseminação de informação sobre o trabalho realizado se tornou mais difícil, fato que facilita a criação de código-fonte de má qualidade, sem padronização e mais passível de defeitos na aplicação final.

Em sistemas de informação, a tendência é que empresas e instituições necessitem de aplicações cada vez mais complexas e integradas, manipulando maior variedade de tipos de dados e necessitando, assim, de equipes de desenvolvimento maiores e mais heterogêneas, neste caso a necessidade de auditoria em projetos é evidente para garantir que o software é construído de forma organizada e seguindo os padrões da empresa. Em vista disso, a motivação deste projeto é fornecer customizações dentro de ferramentas

preexistentes de auditoria de código para facilitar o controle de qualidade e padronização do código-fonte manipulado por uma equipe de desenvolvimento.

## 1.1 Motivação

Hoje um dos maiores desafios das equipes de desenvolvimento é encontrar um processo de construção de software que se adapte à necessidade de compartilhamento de informações de forma organizada e rápida. Vários problemas recorrentes são citados por fábricas de software, como a falta de compartilhamento de informações de maneira dinâmica, dificuldade de entendimento de código alheio e recursos humanos com dificuldade de trabalho em grupo e/ou de comunicação.

Uma das possíveis formas de manter a qualidade de um trabalho de desenvolvimento de software é haver uma auditoria da qualidade do código escrito pelos desenvolvedores. Existem alguns programas de auditoria de código que podem automatizar este trabalho, como o FXCop (1), NDepend (2) e a própria IDE Visual Studio, que em suas novas versões provê a análise do código-fonte criado.

## 1.2 Objetivos

A proposta deste trabalho é aumentar a padronização de projetos a partir de auditoria de código-fonte. Um grande problema de empresas é a padronização de código e arquitetura. Este trabalho tem o intuito de demonstrar o potencial que ferramentas de auditoria de código-fonte automatizadas têm na solução deste problema, contando inclusive, com a possibilidade de criação de regras personalizadas com a utilização da ferramenta FXCOP.

Com esta ferramenta é demonstrada a criação de regras de auditoria que têm o objetivo de identificar problemas comuns no processo de construção de software, demonstrando ser possível realizar varreduras automatizadas para cenários que podem ser estabelecidos por diferentes metodologias e processos de desenvolvimento de sistemas.

## 1.3 Organização do texto

O presente trabalho está estruturado em cinco capítulos e será desenvolvido da seguinte forma:

- Capítulo II: Auditoria de Projeto – Descreve os conceitos relacionados com a auditoria em projetos de desenvolvimento de sistemas de informação, apresentando as principais ferramentas de auditoria automatizada de código-fonte de sistemas que utilizam o *.Net Framework*;

- Capítulo III: Solução Proposta – Apresenta a solução proposta para o problema de auditoria de código-fonte automatizada, explicando como é o processo de compilação da linguagem .NET transformando seu código-fonte em MSIL e metadados, além de apresentar as regras personalizadas que foram construídas utilizando a biblioteca do FXCOP;

- Capítulo IV: Exemplo de Uso - Apresenta casos reais para a utilização das regras que foram construídas com o FXCOP;

- Capítulo V: Conclusão **–** Reúne as considerações finais, assinala as contribuições da pesquisa e sugere possibilidades de aprofundamento posterior.

# 2 Auditoria de Projeto

Auditoria é uma atividade que engloba o exame das operações, processos, sistemas e responsabilidades gerenciais de uma determinada entidade, com o intuito de verificar sua conformidade com certos objetivos e políticas institucionais, orçamento, regras, normas e padrões, como define Cláudia Dias (3). Partindo deste conceito, a fim de que o processo de auditoria seja executado de forma eficaz, é necessário definir os objetivos a serem atingidos, delineando seu foco através de planejamento.

A atividade de planejamento em auditoria de sistemas de informação é definida por Joshua Onome (4) como imprescindível para melhor orientar o desenvolvimento dos trabalhos. O trabalho de auditoria representa um processo contínuo de avaliação de risco, ao qual se adicionam as experiências individuais dos profissionais e a evolução da prática e metodologias. Em projetos de desenvolvimento de software, alguns processos e metodologias de trabalho podem ser aplicados, facilitando o planejamento que deve ser feito para a auditoria.

A próxima seção apontará informações necessárias para que os desenvolvedores realizem a codificação. A partir do detalhamento destas informações, podem ser geradas regras a ser revisadas através de auditoria. Em seguida, será apresentado um conjunto de ferramentas que pode facilitar o trabalho de verificação do código-fonte e que forma o objeto de estudo deste trabalho.

## 2.1 Informações Necessárias para Construção de Software

Durante o processo de construção de software, os desenvolvedores introduzem aspectos computacionais em modelos gerados a partir dos requisitos. Estas informações são necessárias para iniciar a codificação (5). Neste trabalho, estas informações são demonstradas de maneira sucinta como parte do Projeto Arquitetônico e do Projeto Detalhado, divisão clássica da fase de projeto de software.

Quanto mais detalhada a informação disponível sobre o projeto, mais restrições são impostas sobre a forma com que o software deve ser construído e menor é o poder de decisão atribuído aos codificadores (6), facilitando a padronização na construção do sistema e, posteriormente, sua auditoria. A Figura 2.1 apresenta os caminhos que a informação percorre até chegar ao codificador. As próximas subseções discutem estas diversas informações.

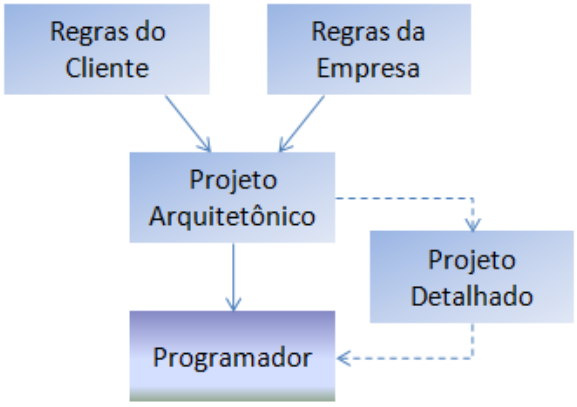


Figura 2.1 - Caminho percorrido pela informação até chegar ao desenvolvedor.

### 2.1.1 Projeto Arquitetônico

Um projeto de software de larga escala e complexo necessita de uma arquitetura para que todos os desenvolvedores possam realizar seu trabalho de codificação de acordo com uma visão comum do sistema (7). A arquitetura visa reduzir o retrabalho e a adoção de diferentes abordagens nas camadas de lógica de negócio, acesso a dados e visualização que implementam as diversas funcionalidades que compõem um sistema. A dependência de uma arquitetura bem definida e conhecida pela equipe responsável pelo desenvolvimento de um software torna-se ainda mais clara em equipes distribuídas ou geograficamente distantes, onde a comunicação entre os integrantes é limitada.

A Arquitetura de Software, segundo Pressman (7), é a estrutura ou organização dos componentes do programa (módulos), o modo pelo qual esses componentes interagem e as estruturas de dados que são utilizadas por eles. Nesta fase são tomadas importantes decisões sobre:

- Os elementos estruturais mais relevantes, juntamente com o seu comportamento, detalhando as colaborações entre eles. Estes elementos são aqueles que descrevem a fundação do sistema, que será utilizada como base para entender, desenvolver e evoluir o projeto de maneira eficiente. Eles podem ser subsistemas (partes do sistema como um todo), interfaces, colaborações e classes;

- Um ou mais padrões de projeto que serão utilizados na codificação. Appleton (8) define um padrão de projeto da seguinte maneira: "O padrão é uma porção identificada de conhecimento profundo, que transmite a essência de uma solução comprovada, para um problema recorrente em certo contexto, em meio a preocupações concorrentes". Os padrões adotados em um projeto devem favorecer o desacoplamento entre as camadas do sistema, remover dependências entre classes fornecedoras/consumidoras, reduzir o retrabalho e facilitar o reaproveitamento do código-fonte já escrito. Com estas características, a equipe de desenvolvimento passa a ganhar em produtividade e cumprir com maior facilidade o prazo definido para a conclusão dos trabalhos.

- A organização do sistema: os componentes do sistema devem ser organizados para facilitar a localização de qualquer classe ou função de maneira intuitiva, levando em consideração os padrões de projeto utilizados.

A arquitetura não se preocupa somente com estruturas e comportamentos, mas também leva em consideração desempenho, usabilidade e tecnologia a ser utilizada no sistema. Os elementos são estruturados de modo a promover uma boa performance durante o processamento dos dados do sistema, principalmente nas operações de cálculo e acesso a dados.

O projeto arquitetônico também deve levar em consideração algumas regras de codificação, que são definidas pela própria empresa que está desenvolvendo o sistema ou pelo cliente. Muitas vezes a obrigatoriedade do cumprimento destas regras fica definida em contrato formal entre o cliente e a empresa.

As regras em geral se aplicam à fase de implementação, a fim de garantir um padrão de código-fonte entre os diferentes sistemas da empresa. A seguir, são listados alguns exemplos de regras de empresa/cliente:

- Padronização de nomenclatura de tabelas no banco de dados, como estabelecer que todas as tabelas devem possuir o prefixo "TB_", e todos os caracteres devem estar em maiúsculo. Exemplos: TB_EMPRESA e TB_ENDERECO.

  Apesar de ser uma regra direcionada ao banco de dados, ela possui influência direta no código das classes de persistência do projeto. Por isso, consideramos que ela é uma regra de codificação.

- Padronização de nomenclatura de variáveis seguindo padrão *Camel-Case*[1]:
  As variáveis devem começar com letras minúsculas. As letras subseqüentes devem também ser minúsculas, exceto se forem a primeira letra de novas palavras;

- Códigos *JavaScript* devem utilizar a biblioteca *JQuery*[2]:
  A utilização desta biblioteca padroniza e diminui a quantidade de código *JavaScript* gerado para o lado cliente de uma aplicação Web.

As regras definidas pelo cliente/empresa interferem diretamente na arquitetura e precisam fazer parte da visão comum que será seguida por toda a equipe de codificação envolvida no projeto.

### 2.1.2 Projeto Detalhado

Concluído o projeto arquitetônico, o desenvolvedor possui a informação sobre quais estruturas de classes padronizadas devem ser utilizadas no projeto. Porém, ele ainda poderá decidir como os métodos e chamadas serão dispostos no sistema. Se a equipe for formada por diferentes desenvolvedores, o projeto poderá possuir implementações muito diferentes para funcionalidades similares.

O projeto detalhado apresenta a seqüencia de classes e métodos que devem ser utilizados na implementação de uma determinada funcionalidade do sistema. Desta maneira, a utilização de documentos como diagramas de classe, diagramas de seqüência e a construção de métodos utilizando-se pseudo-código facilitam o ciclo de vida do desenvolvimento ajudando na padronização do código, além de distribuir o conhecimento sobre a implementação de cada funcionalidade para os desenvolvedores.

Segundo Pressman (7), uma classe descreve um elemento do domínio do problema, focalizando aspectos do problema que são visíveis ao usuário ou ao cliente. Uma classe é composta por atributos e operações. Os atributos são o conjunto de objetos de dados que definem completamente uma classe no contexto do problema, enquanto as operações definem o comportamento do objeto.

---

[1] *Camel-Case* é a denominação em inglês para a prática de escrever palavras compostas ou frases, onde cada palavra é iniciada com uma letra maiúscula e unida com as demais sem espaços em branco. É um padrão largamente utilizado em diversas linguagens de programação, como Java, Ruby, PHP e Python.

[2] *jQuery* é uma biblioteca JavaScript projetada para simplificar a criação de scripts de manipulação de HTML no lado cliente de uma aplicação Web. Ela foi lançada em Janeiro de 2006 no BarCamp, Nova York, por John Resig.

Cada classe tem sua função separada dentro do sistema. Para que as classes cumpram papéis maiores do que suas responsabilidades individuais, são criadas associações, ou colaborações, entre elas, para que cada uma complemente a função da outra. Um diagrama de classes apresenta estas relações, além dos atributos e operações de cada classe e as relações de hierarquia entre elas.

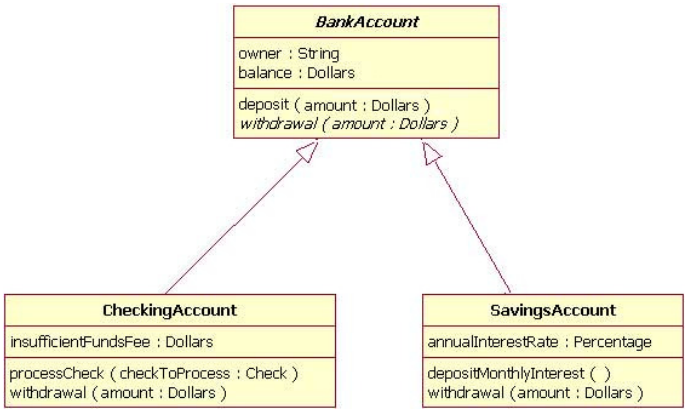


Figura 2.2 – Exemplo de diagrama de classes.

Na Figura 2.2 pode ser observado o exemplo de um diagrama de classes contendo as classes *BankAccount*, *CheckingAccount* e *SavingsAccount*. Podemos observar que cada classe é representada por um quadro dividido em três seções. Na primeira seção, é especificado o nome da classe; na segunda seção, são apresentados os atributos da classe, ou seja, as características que descrevem e diferenciam objetos de uma mesma classe; e na terceira seção, as operações oferecidas pela classe para o restante do sistema. As setas indicam a hierarquia entre as classes, demonstrando que as classes *CheckingAccount* e *SavingsAccount* herdam os atributos e funções da classe *BankAccount*. Isto significa que as classes filhas possuem as mesmas propriedades da classe pai, e ainda implementam seus próprios atributos e funções.

É grande a importância destes diagramas, pois se a lógica de negócio implementada pelo sistema for uniformemente distribuída entre as classes, cada objeto terá responsabilidades pequenas e focadas, fazendo que eventuais alterações tenham escopo localizado. Sendo assim, desenvolvedores que tenham acesso a estes diagramas, tem o conhecimento do motivo da existência de cada classe, conhecendo suas respectivas hierarquias e funções atribuídas, o que ajuda na manutenção de um padrão e facilita o trabalho de auditoria.

Um diagrama de sequência que representa a seqüência de mensagens passadas entre objetos das classes que compõem um sistema. Este diagrama descreve como os

objetos interagem (9) para cumprir um determinado objetivo, geralmente um cenário de um caso de uso.

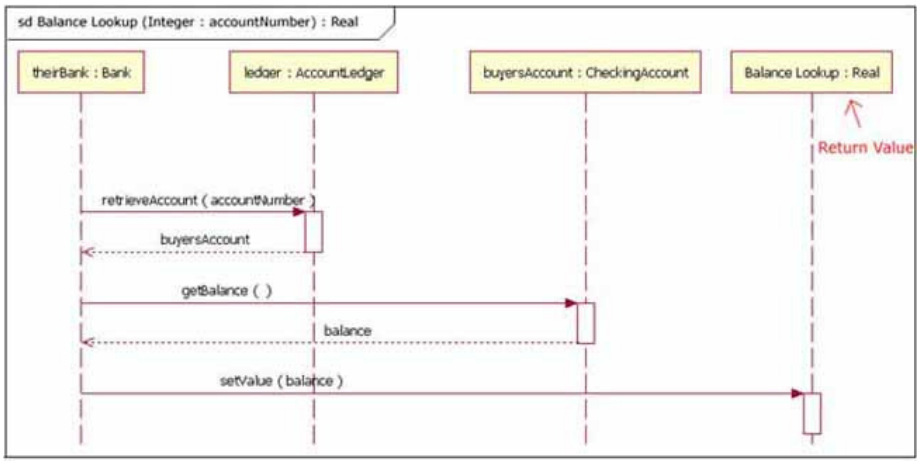


Figura 2.3 – Exemplo de diagrama de seqüência.

Na Figura 2.3 pode ser observado o exemplo de um diagrama de seqüência que demonstra a interação entre as classes *Bank*, *AccountLedger*, *CheckingAccount* e *Real*. Neste, pode-se verificar chamadas de diferentes métodos em momentos distintos para se alterar o saldo da conta de um cliente após a realização de uma compra. No caso, o desenvolvedor que possuir acesso a este documento saberá exatamente os passos necessários para executar este cenário.

Além dos diagramas de classes e sequência, um maior nível de detalhamento pode ser definido no processo de desenvolvimento, exigindo também a documentação de pseudo-código nos métodos das classes e deixando para os codificadores apenas o trabalho de tradução da lógica já escrita para a linguagem de programação do sistema.

Apesar da importância dos documentos construídos na fase de projeto detalhado, nem todas as empresas os adotam. Nos casos em que esta documentação é construída de maneira sucinta, lacunas de conhecimento são deixadas e cada codificador decide, da maneira que lhe for conveniente, a forma de implementar a lógica na solução do problema. Como os desenvolvedores que não carecem de uma documentação completa podem não ter total conhecimento sobre a organização do projeto, este pode apresentar falta de padronização ao longo de sua implementação, tornando-o mais difícil de alterar, corrigir e, de forma mais geral, evoluir.

## 2.2 Ferramentas de Auditoria de Código-Fonte

Para determinar se as regras definidas na arquitetura do sistema foram seguidas pelos desenvolvedores, é comum que auditores especializados, líderes de equipe ou o próprio arquiteto realizem auditorias durante o processo de codificação. Estas auditorias têm como objetivo verificar se a implementação de cada funcionalidade está de acordo com os padrões identificados e documentados como parte da arquitetura do projeto. Com o crescimento do tamanho dos projetos e, conseqüentemente, das equipes de desenvolvimento, fica cada vez mais trabalhoso realizar esta auditoria de forma manual sobre todo o código desenvolvido. Em um projeto de larga escala, as ferramentas de auditoria automatizada de código apóiam a análise.

As ferramentas automatizadas de auditoria reduzem consideravelmente o tempo gasto pelos auditores, pois várias regras simples são disponibilizadas nas versões básicas destas ferramentas. Além disso, algumas delas possibilitam a criação de regras customizadas e capazes de ampliar o poder de verificação e auditoria do código-fonte.

Em um projeto em que todas as informações necessárias para o desenvolvimento são fornecidas de maneira detalhada e estão disponíveis ferramentas de auditoria, a padronização do código tende a ser beneficiada, pois todos os desenvolvedores terão acesso a detalhes de como as funcionalidades devem ser construídas e uma implementação considerada fora dos padrões definidos pela empresa, devido a erros cometidos por um membro da equipe de desenvolvimento, seria facilmente identificada.

Ferramentas de auditoria de codificação são responsáveis por analisar o código-fonte em busca de não conformidades, podendo registrar as regras previamente estabelecidas pelo arquiteto do sistema e garantir que todos os desenvolvedores estejam seguindo as mesmas diretrizes durante a escrita do código do projeto.

Nas ferramentas de auditoria que foram estudadas no decorrer deste trabalho é possível definir quais regras serão utilizadas para um determinado projeto, sendo que algumas regras são definidas na ferramenta, enquanto outras são criadas e incorporadas especificamente para o projeto que está sendo desenvolvido. Existem algumas ferramentas similares utilizadas em outras linguagens, como a ferramenta XDepend, utilizada para a linguagem JAVA, que é semelhante ao NDepend (feito para o .NET Framework).

### 2.2.1 FXCop

O FXCop[3] é uma ferramenta de análise de código fornecida gratuitamente pela Microsoft e direcionada para a verificação de projetos construídos sobre o *Framework .NET*. Diferente das ferramentas de análise de código tradicionais, o FXCop não analisa o código-fonte em si, mas o código binário gerado pelos compiladores .NET em linguagem CIL (*Common Intermediate Language*).

A ferramenta permite uma análise bastante completa da codificação, uma vez que o binário possui muitas informações sobre os metadados[4] utilizados na aplicação. Outra vantagem da análise do CIL, em detrimento a do código-fonte, é que o analisador não fica restrito a uma determinada linguagem do .NET Framework, podendo fazer uma análise independente da linguagem de programação escolhida para o projeto.

Apesar de ter ganho notoriedade apenas recentemente, o FXCop é uma ferramenta madura. Sua primeira versão foi disponibilizada pouco depois do lançamento do .NET Framework, no ano de 2002. Em 2007, o FXCop foi o vencedor do maior prêmio interno da Microsoft, direcionado para a equipe de engenharia, conhecido como *Chairman's Award for Engeneering Excellence*. O prêmio, entregue por Bill Gates, representa um nível de reconhecimento que poucos engenheiros da Microsoft conseguem alcançar. Este evento refletiu em melhorias substanciais no desenvolvimento e testes de código gerenciado dentro da própria Microsoft. A ferramenta está disponível em duas versões:

- Uma versão *stand-alone,* gratuita e disponível para *download* no site da Microsoft, que é utilizada para executar verificações de modo manual. É necessário selecionar o projeto para análise e as regras que serão verificadas;
- Uma versão integrada com o *Visual Studio Team System* versões 2005 e 2008, que pode efetuar verificações automaticamente cada vez que um desenvolvedor estiver enviando seu código para o sistema de controle de versão. Caso o código enviado não esteja de acordo com as regras, o sistema de controle de versão não irá aceitar o envio, retornando uma mensagem que indica a regra com a qual o código não está aderente.

[3] http://www.microsoft.com/downloads/details.aspx?FamilyID=9aeaa970-f281-4fb0-aba1-d59d7ed09772

[4] Dados produzidos pelo compilador que descrevem os tipos declarados e utilizados no código, incluindo a definição de cada tipo, as assinaturas de membros de cada tipo, os membros que seu código referência e outros dados que o ambiente de tempo de execução usa em tempo de execução.

Uma das melhores características do FXCop, e uma das que serviram de motivação para escolha desta ferramenta neste trabalho, é que sua atuação não fica limitada às regras que acompanham a versão original da ferramenta. É possível criar novas regras de acordo com a necessidade de cada projeto, utilizando a SDK do FXCop e acoplá-las na lista de verificações.

### 2.2.2 NDepend

A ferramenta NDepend, assim como o FXCop, tem como objetivo facilitar a auditoria em código-fonte. No entanto, a ferramenta segue uma abordagem diferente da apresentada pelo FXCop: sua proposta é auditar o código-fonte através de uma linguagem de busca própria, chamada CQL, ou *Code Query Language* (10). Na Tabela 2.1 segue um exemplo de consulta nesta linguagem.

```
SELECT TOP 10 METHODS ORDER BY CyclomaticComplexity
```

Tabela 2.1– Exemplo de consulta utilizando Code Query Language.

No exemplo da Figura 2.1, a consulta tem como resultado os dez métodos com maior Complexidade Ciclomática (11) no sistema. Consultas deste tipo podem ser predefinidas e executadas durante o desenvolvimento do sistema. O NDepend pode ser integrado ao Visual Studio desde a versão 2005, podendo ser configurado para executar regras construídas em CQL a cada compilação do código-fonte.

Além de prover a linguagem CQL como possível forma de auditoria, o NDepend também disponibiliza formas gráficas de demonstrar a hierarquia entre classes. Para demonstrar esta forma de exibição, foi construído o projeto *UniversitySampleApplication*, para simular um sistema de controle de disciplinas de uma universidade. Este projeto foi submetido à análise do NDepend, que gerou a Figura 2.4. Ao realizar consultas utilizando CQL, a imagem apresentará tons de vermelho sobre os métodos e classes que forem retornados como resultado.

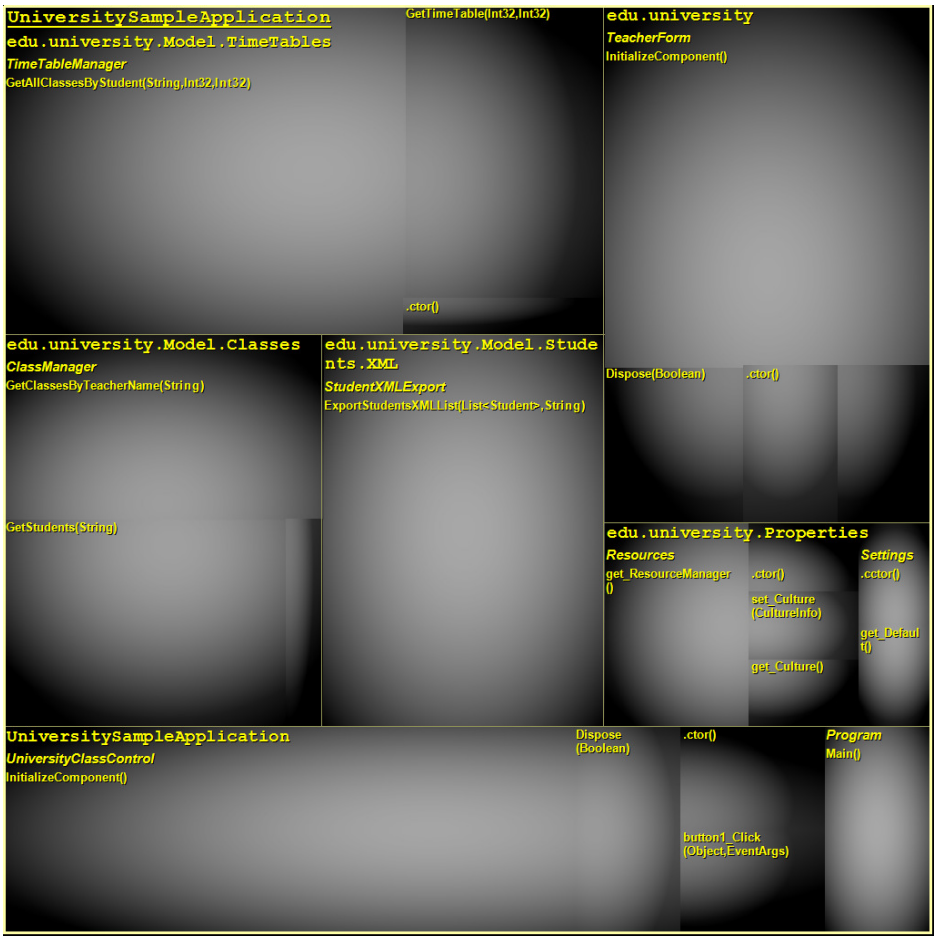

Figura 2.4 – Hierarquia de classes do programa UniversitySampleApplication segundo a representação do NDepend.

Apesar de todas estas funcionalidades disponibilizadas pelo NDepend, o programa tem uma desvantagem quando comparado ao FXCop: o programa não é extensível, ou seja, caso o CQL não atenda a alguma regra pretendida pelo usuário, este não tem a liberdade de desenvolvê-la. Um exemplo de uma regra que não poderia ser desenvolvida a partir do CQL é a análise da documentação gerada pelo código-fonte, que é feita a partir da leitura do XML de documentação gerado pelo Visual Studio.

### 2.2.3 Visual Studio

O Visual Studio é a *IDE* oficial da *Microsoft* para desenvolvimento usando as linguagens do framework .NET. Na versão 2010, a *IDE* disponibiliza uma ferramenta para análise de código-fonte da aplicação que está sendo desenvolvida, a partir de algumas regras predefinidas. Dentro da ferramenta de análise podem ser escolhidas algumas regras separadamente, ou como conjunto de regras com temas separados como segue na Figura 2.5.

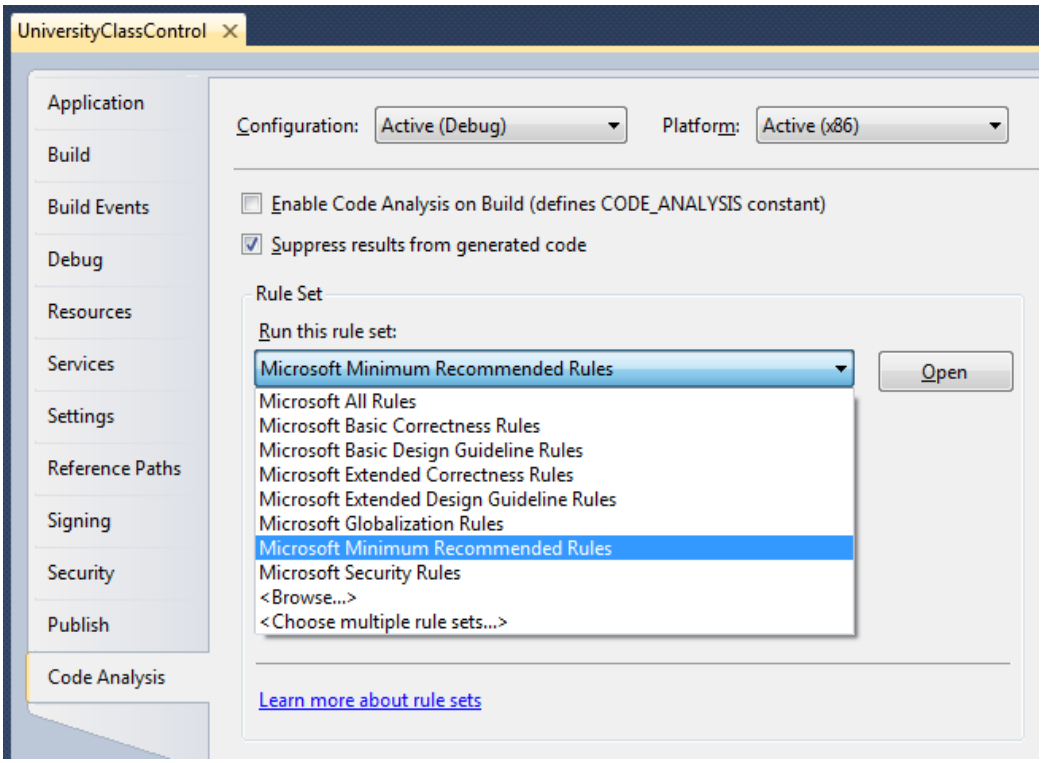

Figura 2.5 - Sessão de análise de código do Visual Studio 2010.

A iniciativa desta ferramenta de análise ser disponibilizada comprova que a preocupação com a qualidade em código-fonte tem ganhado importância no cenário de desenvolvimento de sistemas.

### 2.2.4 Comentários sobre as ferramentas analisadas

As ferramentas analisadas neste trabalho podem ter grande êxito para o propósito de automação de auditoria de código-fonte se utilizadas de maneira correta e com freqüência durante o processo de desenvolvimento. Cada uma das ferramentas apresenta suas particularidades e pode ser utilizada de diferentes formas, possuindo tanto pontos positivos como pontos negativos quando comparadas. A Tabela 2.2 apresenta um comparativo entre as ferramentas, em relação a características relevantes:

| | **FXCop** | **NDepend** | **Visual Studio** |
|---|---|---|---|
| **Ferramenta gratuita** | X | | |
| **Regras de validação prontas já incluídas** | X | X | X |
| **Integração com o ambiente de desenvolvimento** | | X | X |
| **Linguagem de consulta CQL** | | X | |
| **Suporte a criação de regras personalizadas** | X | | |

Tabela 2.2 - Comparativo de características entre as ferramentas analisadas.

Enquanto o *NDepend* apresenta a vantagem de possuir muitas regras prontas e facilmente acessíveis através de sua linguagem CQL (*Code Query Language*), o FXCop apresenta dois grandes diferenciais: sua possibilidade de extensão, através da criação de novas regras a partir de seu SDK, e a possibilidade de utilizar esta ferramenta de forma gratuita, no modo *stand-alone*, ou de forma paga integrada ao *Team System*.

Cada uma das ferramentas pode ser melhor aproveitada em diferentes cenários, mas a viabilidade de implementar novas regras, conforme previsto no FXCop, é sem dúvida um grande diferencial, assim sendo escolhida como ferramenta para a realização deste trabalho.

## 2.3 Considerações finais

Um processo de auditoria bem planejado promove a padronização do código-fonte gerado por uma equipe heterogênea de desenvolvedores, e assim procura garantir uma maior qualidade no produto final, favorecendo a manutenibilidade do código e o cumprimento das determinações sugeridas pela equipe de arquitetura.

A utilização de ferramentas de auditoria automatizadas facilita bastante o trabalho de auditoria, pois é possível realizar varreduras em busca de diversos tipos de não-conformidades, inclusive com a criação de verificações específicas para um determinado projeto.

# 3 Solução Proposta

A solução abordada neste trabalho consiste em fornecer ferramentas que ajudam a garantir o padrão de codificação estabelecido para uma determinada aplicação. Algumas ferramentas foram pesquisadas como o NDepend (2), o FXCop (1) e a funcionalidade *code metrics* do Visual Studio (12). O projeto foi construído baseando-se na ferramenta Microsoft FXCop versão 1.36, devido a sua capacidade de customização, através da criação de novas regras de análise de código de acordo com a necessidade do projeto. A Microsoft disponibiliza a biblioteca do FXCop que torna possível esta construção.

Serão apresentadas algumas regras nativas do FXCop e alguns exemplos de uso. Também será apresentado o modelo de interfaces da ferramenta para criação de regras personalizadas, seguido do conjunto de regras que foram implementadas neste trabalho.

## 3.1 Utilizando as Regras Oferecidas pelo FXCOP

Para o uso do FXCop e suas regras, são necessários: o .NET Framework[5] 3.5 e o FXCop[6] 1.36.

Após a instalação do FXCop, a interface da ferramenta é demonstrada na Figura 3.1. A seleção do projeto a ser analisado é feita através da aba *Targets*, como segue abaixo na Figura 3.1:

[2] http://www.microsoft.com/downloads/details.aspx?FamilyId=333325fd-ae52-4e35-b531-508d977d32a6
[3] http://www.microsoft.com/downloads/details.aspx?FamilyID=9aeaa970-f281-4fb0-aba1-d59d7ed09772

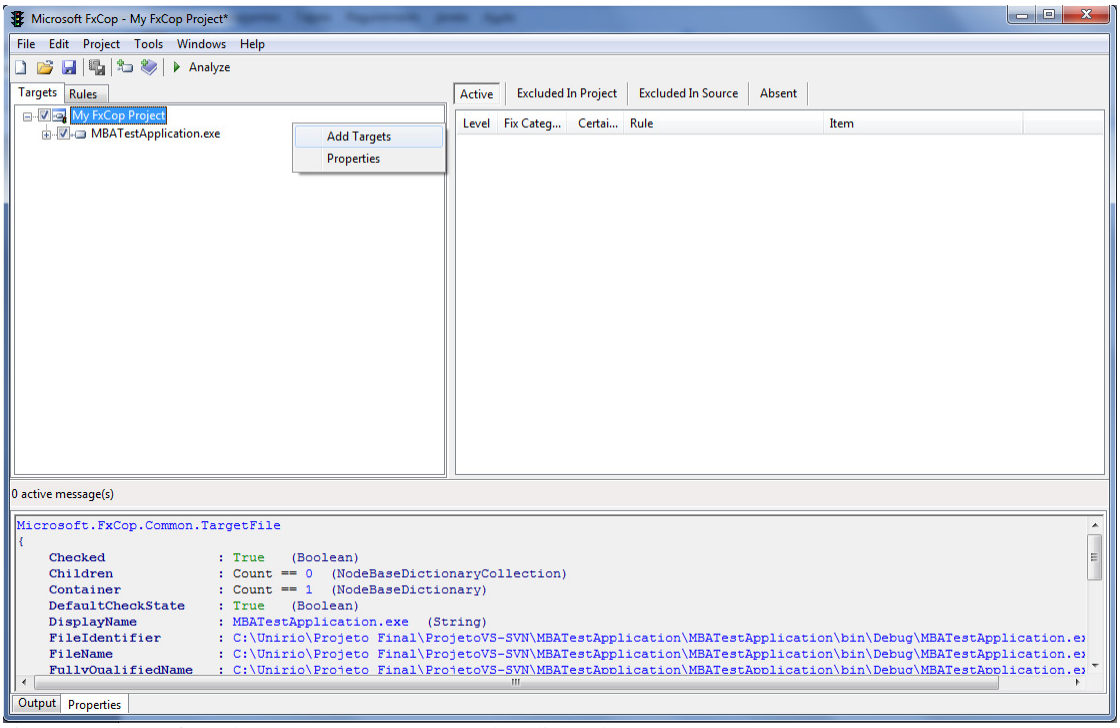


Figura 3.1- Interface do FXCop no momento da adição de uma Target.

No lado esquerdo da tela principal, a ferramenta apresenta duas abas: *Rules e Targets*. A aba *Rules* apresenta a lista de regras que são oferecidas pelo FXCop, separadas em grupos, como pode ser visto na Figura 3.2. Nesta aba, é possível selecionar as regras que devem ser analisadas no projeto.

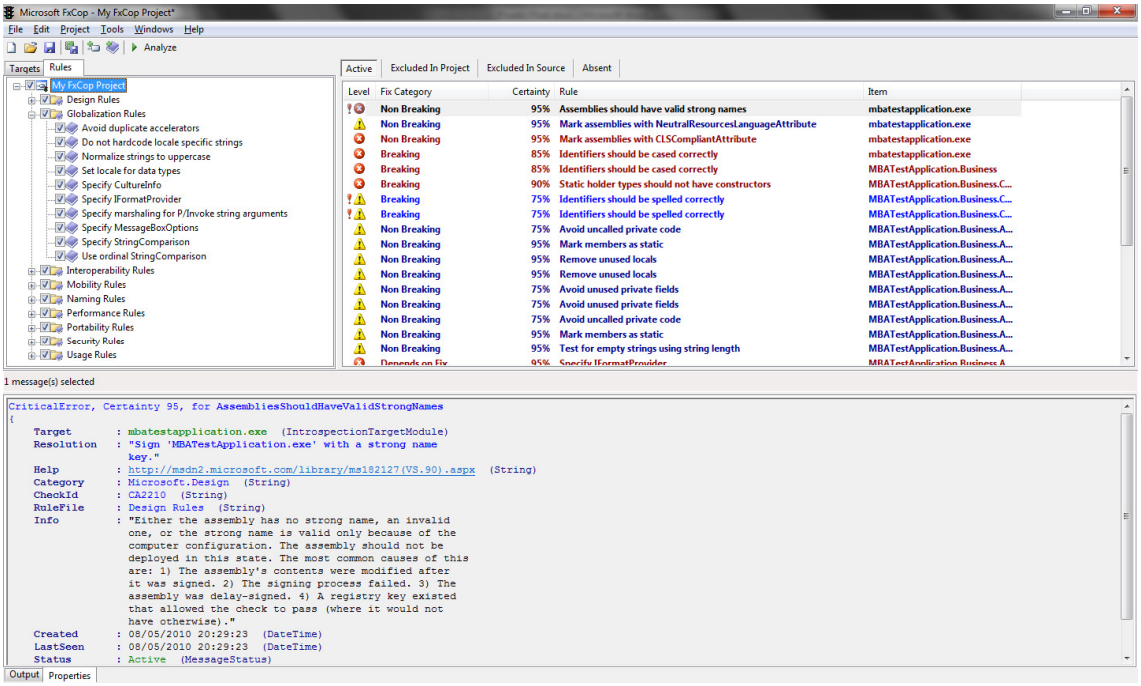


Figura 3.2 - FXCop após analisar regras (rules) em um determinado projeto (target).

Ao lado direito, são exibidas as não conformidades que foram encontradas de acordo com as regras selecionadas para análise. Ao clicar sobre um item encontrado, o quadro localizado na parte inferior da janela exibe informações detalhadas sobre o problema, como fontes de referência para solucioná-lo, localização do problema no código-fonte e uma descrição da regra.

Na Figura 3.2 temos um projeto exemplo com alguns erros, forçando que o FXCop os aponte. Será utilizada a regra pré definida *Remove Unused Locals*, que está localizada no grupo *Performance Rules,* e a aplicação de teste chamada: *University Class Control*, que foi criada para esta demonstração.

Podemos notar no trecho de código da Tabela 3.1, que o método *GetStudents()* possui uma variável local *tempClass* que não está sendo utilizada em nenhum momento do código-fonte.

```
public List<Students> GetStudents(String ClassID)
{
        //Local variable that points to the current class being analyzed
        Class tempClass = null;

        //FOR Statement replaced by a FOREACH for code cleaning progress
        /*
        for (int i =0; i < CurrentClasses.Count; i++)
        {
                tempClass = CurrentClasses[i];
                if (tempClass.ID.Equals(ClassID))
                {
                        return tempClass.Students
                }
        }
        */
        foreach(Class oneClass in this.CurrentClasses)
        {
                if (oneClass.ID.Equals(ClassID))
                {
                        return oneClass.Students
                }
        }
        return null;
}
```

Tabela 3.1 – Código de GetStudents utilizado para demonstração do esquecimento da variável tempClass.

Adicionando o Projeto University Class Control como target no FXCop podemos observar que a falta de uso da variável é acusada, conforme indicado na Figura 3.3.

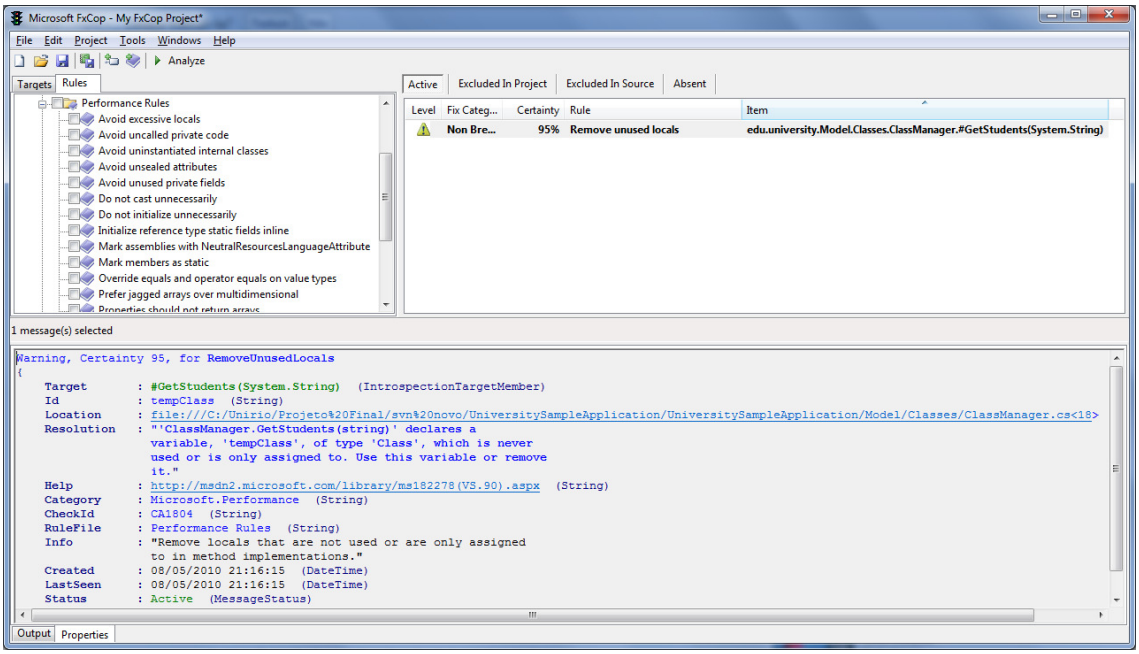

Figura 3.3 – Demonstração da Regra *Remove Unused locals*.

## 3.2 Construindo Regras para o FXCop

O FXCop possui algumas características interessantes, mas sua capacidade de extensão é sem dúvida um de seus pontos fortes. Através da FXCop SDK, podemos construir regras customizadas de análise de código, tornando a ferramenta adaptável a boa parte das possíveis necessidades em uma auditoria de código.

A Microsoft fornece e encoraja os desenvolvedores a construir suas próprias regras com sua API. Porém, até o momento da criação deste trabalho, a empresa não disponibilizou nenhuma documentação oficial sobre sua utilização, o que é uma falha considerável. Felizmente, existem artigos na Internet de onde é possível extrair boa parte do conhecimento necessário para a construção deste trabalho.

A FXCop SDK é composta por duas DLLs, que fornecem todas as interfaces que devem ser utilizadas para criação de novas regras. Para criar uma regra customizada, foram implementadas as interfaces que fornecem os dados necessários Após inspecionados os dados (fornecidos pelo FXCop através do MSIL), é retornada uma lista de problemas encontrados.

Esta análise não é feita sobre o código-fonte, mas sim sobre o código pré-compilado, chamado *Microsoft Intermediate Language* (MSIL), que é gerado pelo compilador .NET e que faz parte do assembly compilado (Arquivos EXE ou DLL). A análise das regras só é possível devido à alta disponibilidade de metadados, fornecidos pelo MSIL.

### 3.2.1 MSIL

*Microsoft Intermediate Language* (MSIL) (13) é uma plataforma independente de linguagem, o que significa que compiladores .NET gerarão a mesma IL independente da linguagem em que o código-fonte foi escrito. A Figura 3.4 apresenta o processo de compilação do código até sua execução no sistema operacional.

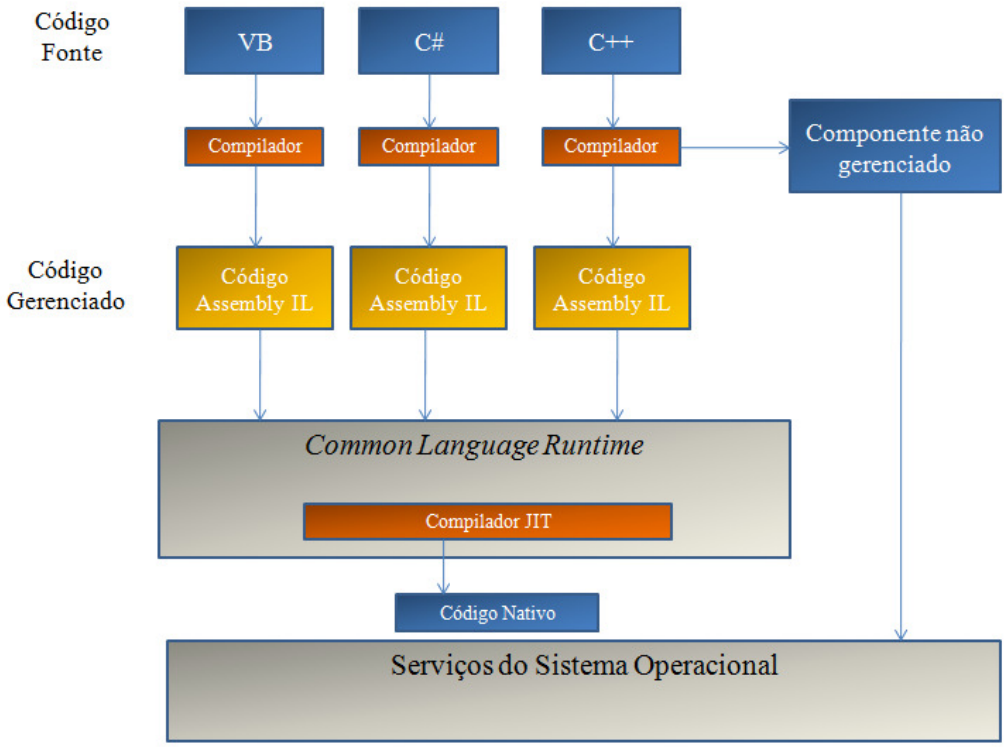


Figura 3.4 – Caminho do código-fonte até a transformação em código nativo.

Quando o .NET Framework compila o código-fonte para código gerenciado, converte seu código em *Microsoft Intermediate Language* (MSIL), que é um conjunto de instruções independentes de CPU, o qual pode ser convertido em código nativo com eficiência. O MSIL inclui instruções para carregamento, armazenamento, inicialização e chamada de métodos em objetos, alem de instruções para operações aritméticas e lógicas, fluxo de controle, acesso direto à memória, tratamento de exceção e outras operações.

Antes de o código poder ser executado, o MSIL deve ser convertido no código específico da CPU, geralmente por um Compilador Just-In-Time (JIT). O *Common Language Runtime* é um ambiente de tempo de execução que executa o código e fornece serviços que facilitam o processo de desenvolvimento. Além disso, fornece um

ou mais compiladores JIT para cada arquitetura de computador para a qual oferece suporte. O mesmo conjunto de MSIL pode ser compilado JIT e executado em qualquer arquitetura suportada.

Quando um compilador produz MSIL, ele também produz metadados. Metadados descrevem os tipos declarados e utilizados no código, incluindo a definição de cada tipo, as assinaturas de membros de cada tipo, os membros que seu código referencia e outros dados que o ambiente de tempo de execução usa em tempo de execução. O MSIL e os metadados estão contidos em um arquivo executável portátil (PE), que é baseado e estende os formatos Microsoft PE e COFF públicos e usados historicamente para conteúdo executável. Este formato de arquivo, que acomoda MSIL ou código nativo além de metadados, permite ao sistema operacional reconhecer imagens do *Common Language Runtime*. A presença de metadados no arquivo junto com o MSIL permite que o código descreva a si mesmo, o que significa que não há necessidade de bibliotecas de tipos. O ambiente de tempo de execução localiza e extrai os metadados do arquivo na medida em que for necessário durante a execução.

Com o programa *Intermediate Language Disassembler* se consegue traduzir código-fonte de um arquivo executável de uma aplicação para *Intermediate Language*. Esta aplicação mostra toda a estrutura, classes, métodos, variáveis e todos os dados locais e globais de uma aplicação. As Tabelas 3.2 e 3.3 apresentam um exemplo simples de código que foi traduzido para IL, utilizando ILDasm (14).

```
public double GetVolume()
{
    double volume = height*width*thickness;
    if(volume<0)
        return 0;
    return volume;
}
```

Tabela 3.2 – Trecho de código-fonte a ser traduzido para MSIL.

```
.method public hidebysig instance float64
GetVolume() cil managed
{
// Code size 51 (0x33)
.maxstack 2
.locals init ([0] float64 volume,
[1] float64 CS$00000003$00000000)
IL_0000: ldarg.0
IL_0001: ldfld float64 OOP.Aperture::height
IL_0006: ldarg.0
IL_0007: ldfld float64 OOP.Aperture::width
IL_000c: mul
IL_000d: ldarg.0
IL_000e: ldfld float64 OOP.Aperture::thickness
IL_0013: mul
IL_0014: stloc.0
IL_0015: ldloc.0
IL_0016: ldc.r8 0.0
IL_001f: bge.un.s IL_002d
IL_0021: ldc.r8 0.0
IL_002a: stloc.1
IL_002b: br.s IL_0031
IL_002d: ldloc.0
IL_002e: stloc.1
IL_002f: br.s IL_0031
IL_0031: ldloc.1
IL_0032: ret
} // end of method Aperture::GetVolume
```

Tabela 3.3 – Código no formato MSIL referente ao código da Tabela 3.2.

### 3.2.2 Interfaces usadas na construção de regras

O *framework .NET* inclui uma API de análise de metadados chamada *System.Reflection*. Esta API era utilizada nas versões iniciais do FXCop para a análise das regras. Com o passar do tempo, os desenvolvedores perceberam que utilizar a *Reflection* API possuía limitações, que ficavam evidentes quando utilizada em uma ferramenta de análise de código, simplesmente porque não havia sido construída para esta finalidade. Para resolver isso, foi desenvolvida uma API similar, chamada *Introspection*, que apresenta funcionalidades de análise bastante melhoradas em relação a *Reflection*.

A *Introspection* divide os metadados em uma lista hierárquica de tipos, chamadas de *Node,* ou *nó* em português. Cada nó é representado por uma classe .NET da *Introspection API*, que é utilizada para acessar atributos relevantes do metadado. A Figura 3.5 apresenta o diagrama de classes fornecido pela *Introspection*:

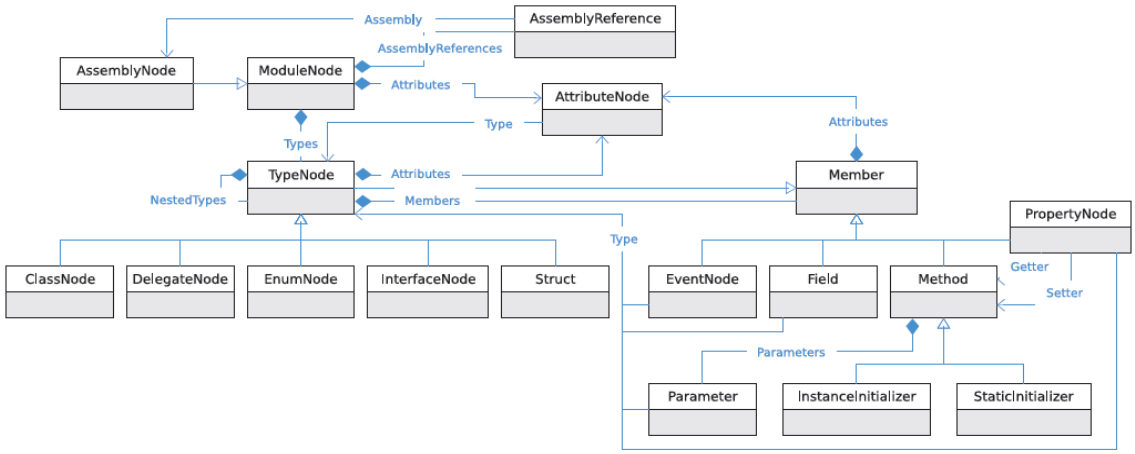


Figura 3.5 – Diagrama de Classes do Introspection.

O processo de análise do FXCop é baseado em checar e visitar cada nó apresentado pela *Instrospection* do *assembly* sendo analisado. Os nós mais relevantes são *ModuleNode*, *TypeNode* (como, por exemplo, *ClassNode*) e *Member* (como, por exemplo, *PropertyNode* e *Method*).

Quando o FXCop encontra um nó relevante para uma regra, ele faz uma chamada ao método *Check* implementado na regra. É possível sobrescrever qualquer um ou todos os métodos de checagem disponíveis. Por padrão, os métodos de checagem não fazem nenhuma operação, a não ser que sejam sobrescritos na classe da nova regra customizada. Os métodos de checagem são os seguintes:

- **`Check(ModuleNode module)`**: analisa um módulo (um assembly é composto por um ou mais módulos);
- **`Check(Resource resource)`**: analisa um resource (como uma imagem BMP) que esteja definido dentro do módulo;
- **`Check(String namespaceName, TypeNodeCollection types)`**: analisa tipos, dentro de um determinado namespace;
- **`Check(TypeNode type)`**: analisa um tipo (exemplos: Classe, Struct, Enum, etc.);
- **`Check(Member member)`**: analisa um membro de um tipo (exemplos: Method, Field, PropertyNode, etc.);
- **`Check(Parameter parameter)`**: analisa um parâmetro de um membro.

Para criar uma nova regra, basta criar um projeto no Visual Studio (foi utilizada a versão 2010 Beta neste trabalho), do tipo *Class Library* e adicionar referências aos

arquivos FxCopSdk.dll e Microsoft.Cci.dll. Eles são a FXCop SDK e a Instrospection API, respectivamente.

Após isto, deve ser criada a classe da nova regra, estendendo a classe *BaseIntrospectionRule*, e sobrescrevendo os métodos que desejamos para executar as verificações.

A Tabela 3.4 apresenta um exemplo de classe de verificação, que implementa o método *Check (Member member)* para verificar se os métodos do assembly verificado possuem mais de 10 linhas de código.

```
using System.Text;
using Microsoft.FxCop.Sdk;

namespace MBARules
{
    public class NumeroMaximoLinhas : BaseIntrospectionRule
    {


        public override ProblemCollection Check(Member member)
        {

            Method metodo = member as Method;
            if (metodo != null )
            {
                int startLine = metodo.SourceContext.StartLine;

                int endLine = metodo.Instructions[metodo.Instructions.Count -
1].SourceContext.EndLine;

                int numeroLinhas = endLine - startLine;

                 if (numeroLinhas > 10)
                {

                    this.Problems.Add(new Problem(this.GetResolution()));

                }
            }
            return this.Problems;
        }

    }


}
```

Tabela 3.4 – Implementação da regra personalizada de tamanho máximo de linhas para um método.

### 3.2.3 Depurando as Regras

Como as regras são personalizações feitas para serem utilizadas como extensão do FXCop, para viabilizar a depuração, foi utilizado um recurso do Visual Studio onde se pode anexar um processo ativo no depurador, no caso, o processo do FXCop. No

Visual Studio, para se anexar um processo a ser depurado, deve-se ir ao menu *Debug* / *Attach to Process*, que apresentará a tela demonstrada na Figura 3.6.

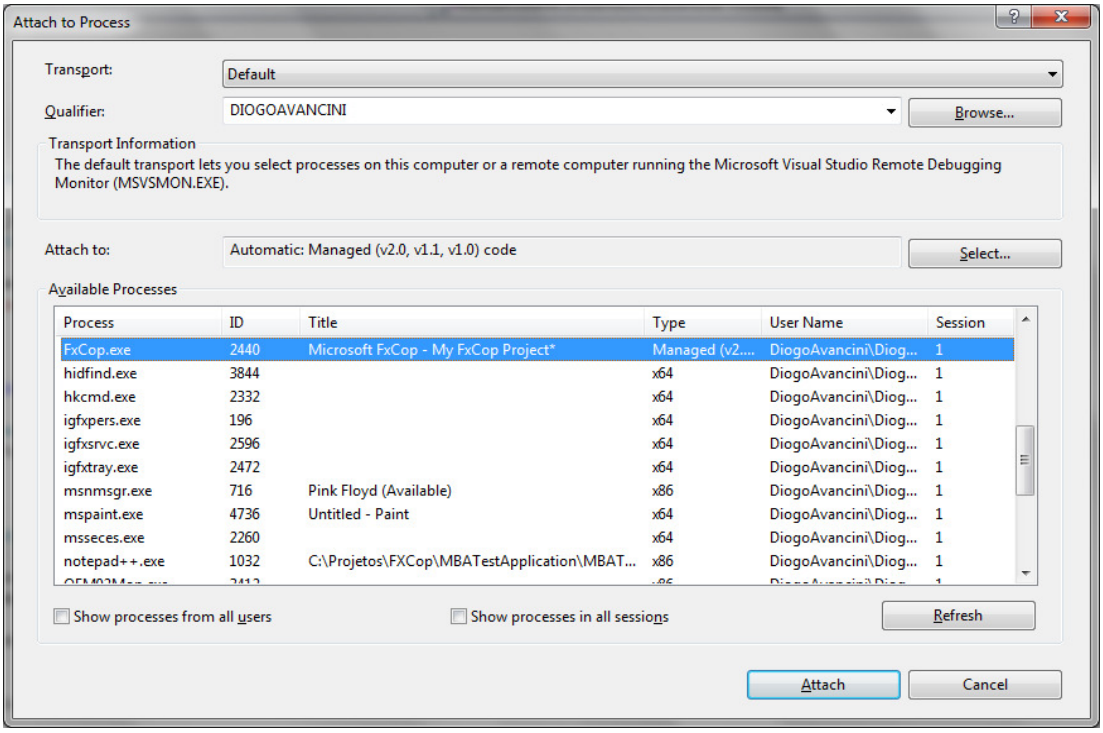

Figura 3.6 – Tela Attach to process utilizada para depurar processos externos ao Visual Studio.

Nesta tela, o processo FxCop.exe deve ser escolhido. Este deve estar executando com a última versão de código-fonte das regras a serem depuradas já adicionadas.

Quando a análise do FXCop é iniciada, vários processos (*threads*) são iniciados ao mesmo tempo para a análise das regras escolhidas. Isto pode confundir a depuração, pois ao depurar linha a linha com os comandos *Step-Out* ou *Step-In*, um comportamento diferente do esperado pode acontecer. Outra *thread* que está na fila para ser executada poderá ser reconhecida pelo depurador, e esta pode estar em um ponto diferente do código do que o esperado. Isto foi contornado utilizando-se muitos *breakpoints* em pontos estratégicos e *Quick Watch,* recurso do Visual Studio para depuração, para se descobrir os valores de variáveis no momento da execução. Só assim foi possível um entendimento melhor e desenvolvimento de regras personalizadas com mais eficiência.

## 3.3 O Pacote de Regras Desenvolvido

Para a personalização de regras para o FXCop, foram escolhidas seis situações que ajudariam na vida cotidiana de um desenvolvedor. Este pacote de regras foi definido em comum acordo com orientador do projeto, baseando-se em experiências

obtidas no mercado de trabalho. As regras escolhidas não tem nenhum tipo de ligação entre elas, mostrando que o FXCop pode atender a variadas situações. O pacote inclui regras como:

- Uso de listas *ArrayList* com tipo de objetos não definido;
- Tamanho máximo de um método;
- Análise de palavras proibidas em comentários;
- Complexidade ciclomática (11);
- Chamadas de método com parâmetros de tipo string fixos;
- Nomenclaturas na criação e manipulação de arquivos XML.

Para facilitar a configuração de alguns itens, foi criado o arquivo de configuração *FXConfiguration.xml*, que auxiliará com dados configuráveis para algumas regras.

### 3.3.1 Utilização de listas ArrayList sem definir tipo de seus objetos

Um problema facilmente encontrado em código-fonte de projetos é a falta de definição dos tipos dos objetos dentro de uma lista. Em código .NET isto é caracterizado pelo uso de *ArrayList*, ou outros tipos de coleção, sem nenhuma especificação de tipo, como segue exemplo na Tabela 3.5:

```
ArrayList listaNaoTipada = new ArrayList();
```

Tabela 3.5 – Exemplo de lista sem tipo.

Neste caso, o correto seria utilizar a classe *List* com a especificação do tipo de seus objetos, como apresentado na Tabela 3.6:

```
List<String> listaTipada = new List<String>();
```

Tabela 3.6 – Exemplo de lista com tipo.

Com esta declaração, sabemos que a variável "*listaTipada*" contém uma lista de objetos do tipo String. Esta regra tem como objetivo identificar todos os casos que não se encaixem nesta padronização de definição de tipos de listas.

### 3.3.2 Tamanho Máximo de um Método

Algumas empresas definem um tamanho máximo de linhas de código para os seus métodos a fim de manter o código organizado da melhor maneira possível. Apenas

um conjunto de métodos pequenos obviamente não seria garantia de um código organizado, porém esta regra, se combinada com outras já pré-existentes, consegue reduzir bastante a possibilidade de um código com má qualidade (15). No caso desta implementação, foi utilizado o arquivo XML de configuração, chamado *FXConfiguration.xml*, para se definir o número máximo de linhas desejado. Para configurar este número, basta definir seu valor dentro da tag *numeroMaximoLinhasPorMetodo*. A Tabela 3.7 sugere um máximo de 100 linhas para um método:

```
<FXCOP>
  <Configuration>
    <numeroMaximoLinhasPorMetodo>100</numeroMaximoLinhasPorMetodo>
  </Configuration>
</FXCOP>
```

Tabela 3.7 – Demonstração do XML de configuração para máximo de linhas permitidas em um método.

### 3.3.3 Análise de Palavras Proibidas na Documentação do Código

É comum desenvolvedores cometerem gafes e escrever palavras chulas na documentação do código, e muitas destas palavras não são retiradas, acabando sendo publicadas nos servidores de homologação e produção. Para resolver este problema, esta regra foi definida de maneira que seja possível configurar as palavras que serão buscadas na documentação no arquivo de configuração *FXConfiguration.xml*.

A documentação de código não é inclusa no código intermediário MSIL. Então para, que esta regra funcione, é necessária uma configuração no momento do build da aplicação para que seja gerado um arquivo XML de documentação. É necessário também que este XML de documentação seja gerado na mesma pasta que o arquivo executável do projeto a ser analisado.

Para habilitar esta documentação, basta seguir os seguintes passos: no menu *Project* / <nomeDoProjeto> *Properties*, na aba *Build*, seção *output*, deve ser marcada a opção *XML documentation file* e definir o nome do arquivo XML a ser gerado, como segue na Figura 3.7.

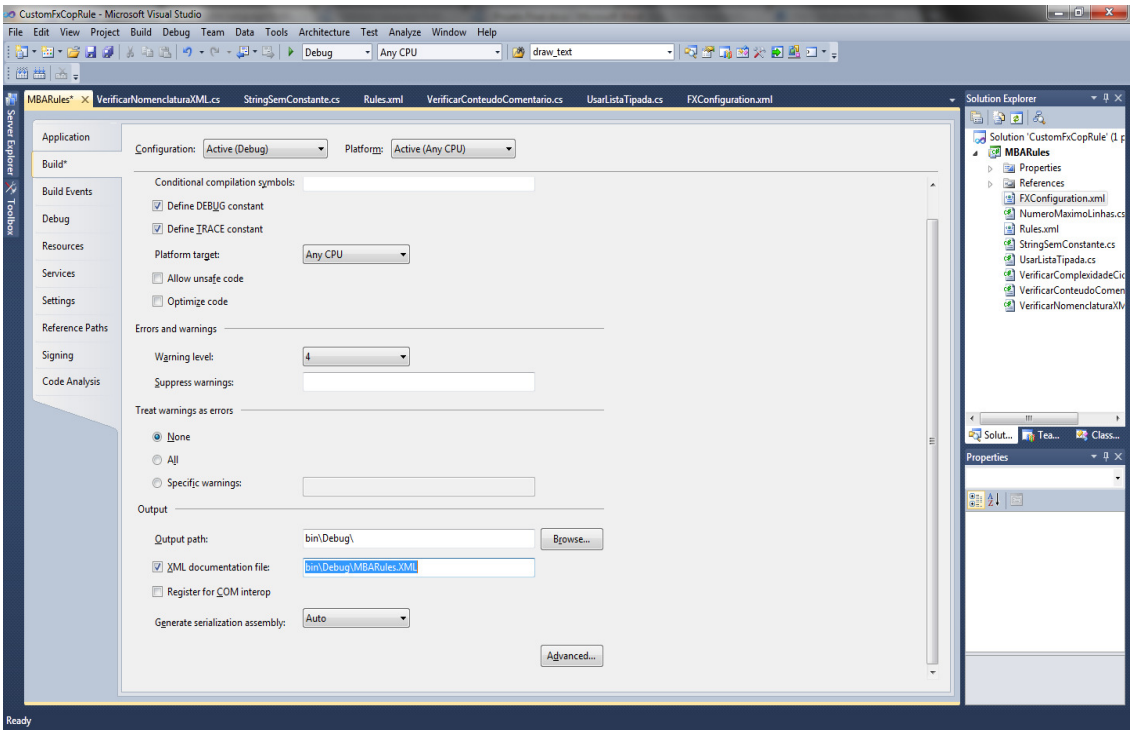

Figura 3.7 – Demonstração do local da configuração do arquivo XML de documentação do código-fonte.

Após o arquivo ser gerado no mesmo local do executável, deve-se configurar as palavras que serão proibidas na documentação de métodos no arquivo de configuração, como apresenta a Tabela 3.8.

```
<FXCOP>
  <Configuration>
    <palavrasProibidas>
      <palavraProibida>Palavra1</palavraProibida>
      <palavraProibida>Palavra2</palavraProibida>
    </palavrasProibidas>
  </Configuration>
</FXCOP>
```

Tabela 3.8 – XML de configuração de palavras proibidas na documentação de um método.

### 3.3.4 Complexidade Ciclomática

Complexidade Ciclomática (11) é uma métrica criada em 1976 por Thomas J. McCabe para medir a complexidade de um método ou uma expressão a partir de sua árvore de decisão, mapeando o número de caminhos independentes que um programa pode tomar durante a sua execução. Não é difícil chegar à conclusão que quanto maior a quantidade de caminhos que o sistema pode percorrer, maior será a quantidade de testes sobre o código (16).

Para medir a Complexidade Ciclomática de um código-fonte, deve-se somar todas as possíveis decisões, do início do código ao final, acrescidos de um. Com isso temos a fórmula:

Complexidade Ciclomática = Número de Decisões +1

Para esta regra foi configurado como padrão que um método que tenha complexidade ciclomática calculada com um valor inferior a 20, apresenta baixa complexidade, se o valor estiver entre 20 e 40 é constatada média complexidade, e no caso do valor estar acima de 40, alta complexidade. Estes parâmetros ficaram configuráveis a partir do arquivo *FXConfiguration.xml*, onde se pode aumentar ou diminuir a rigidez a auditoria. A regra aponta o problema como *Warning* se a complexidade ciclomática se encaixar como média e como *Error* no caso da complexidade ser calculada como alta, como pode ser visto no exemplo de configuração da Tabela 3.9:

```
<FXCOP>
  <Configuration>
    <complexidadeCiclomatica media="20" alta="40" />
  </Configuration>
</FXCOP>
```

Tabela 3.9 – Demonstração do XML de configuração para complexidade ciclomática de um método.

Para a confecção desta regra houve a necessidade de um estudo sobre os comandos MSIL gerados pela aplicação, pois deveria ser incrementado o valor da complexidade ciclomática apenas para os nós da árvore de decisão do código, ou seja, apenas em alguns comandos de decisão específicos que significam que um novo caminho pode ser percorrido. A Tabela 3.10 apresenta todas as operações que acrescem pontos no cálculo da Complexidade Ciclomática de um método.

| Operação | Complexidade Ciclomática |
|---|---|
| If | +1 |
| else IF | +1 |
| Else | 0 |
| select case | +1, para cada caso |
| select default | 0 |
| for/foreach | +1 |
| do while | +1 |
| Do | 0 |
| While | +1 |
| Catch | +1 |

Tabela 3.10 – Demonstração de operações e seus valores para a complexidade ciclomática.

Na Tabela 3.11, é apresentado um exemplo de código-fonte com suas operações e valores computados para o cálculo da complexidade ciclomática.

```
if( c1() )      //+1
   f1();
else            //0
   f2();

while( c2() )   //+1
   f3();
```

Tabela 3.11 – Exemplo de código com complexidade ciclomática igual a 3.

### 3.3.5 Chamadas de método com parâmetros de tipo String fixos

Uma prática muito comum entre desenvolvedores é deixar algumas chamadas de métodos com parâmetros passados de maneira fixa, o que torna o sistema dificilmente configurável e também pode gerar uma grande refatoração do sistema, para se implementar algo como, por exemplo, sua internacionalização (15).

A regra procura alertar sobre qualquer chamada de método que utilize pelo menos um argumento String em que esteja sendo criada uma constante no ato da chamada do método. Na melhor das hipóteses, deveria ter sido declarada uma constante no início da classe ou em um arquivo de constantes, a fim de facilitar a localização de constantes "*hard-coded"* no projeto. A Tabela 3.12 apresenta um exemplo de chamada de método com parâmetro string fixo.

```
//Chamada utilizando String Fixa.

SqlConnection conexao = new SqlConnection("DataSource=SQL05");
```

Tabela 3.12 - Exemplo de chamada de método onde foi utilizada uma string fixa.

Infelizmente, utilizando o FXCop não é possível alertar sobre todas as ocorrências de chamadas utilizando parâmetro string fixo. Caso uma string esteja mapeada em uma constante em alguma classe do projeto, e seja feita uma chamada de método passando uma string fixa de valor exatamente igual aquela já mapeada, a situação não será detectada, como pode ser observado na Tabela 3.13, pois o MSIL analisado pelo FXCop não informa se a constante foi ou não declarada no momento da chamada. Sendo assim, a regra implementada iria mapear a constante e não alertaria o problema, ainda que a constante estivesse mapeada em outra classe.

```
public class ExemploHardCoded
{
    private const String CONSTANTE_MAPEADA = "DataSource=SQ01";

    private void AbrirConexao1()
    {
        //Chamada OK! Utiliza constante mapeada.
        SqlConnection conexao1 = new SqlConnection(CONSTANTE_MAPEADA);

        //Chamada Problematica NAO DETECTADA!
        //O Valor fixo utilizado esta mapeado em uma constante.
        SqlConnection conexao2 = new SqlConnection("DataSource=SQ01");
    }
}
```

Tabela 3.13 - Exemplo de chamada de método não detectável.

### 3.3.6 Nomenclaturas na manipulação e criação de arquivos XML

Durante o desenvolvimento de funcionalidades de geração de arquivos XML é considerado boa prática criar um arquivo de constantes que centralize as informações dos nomes das *Tags* e *Attributes* utilizados pelo XML gerado, a fim de facilitar qualquer alteração no nome destes elementos.

A regra em questão foi escrita com o intuito de garantir que os nomes das constantes declaradas sigam um prefixo "TG_" e "AT_" para *Tags* e Atributos, respectivamente, para que seja facilmente identificado qual é o tipo de elemento a qual a constante se refere. Uma vez em execução, a regra faz um mapeamento de todas as contantes do projeto e analisa todas as chamadas aos métodos *WriteElement* e *WriteAttribute* da biblioteca *System.XML* do .NET, que é a API mais utilizada na

geração de arquivos XML utilizando o *Framework* da Microsoft. A análise das chamadas consiste em verificar as constantes passadas para o parâmetro *localname*, que sempre recebe o nome do elemento a ser criado no arquivo XML final. Caso esta chamada tenha sido feita através de uma constante mapeada, a regra verifica se o nome da constante está de acordo com a convenção de prefixos "TG" e "AT" adotada.

Na Tabela 3.14 segue um exemplo de XML gerado por uma aplicação:

```
<AuthorsList>
   <Author au_id="BR02" au_fname="Luís Fernando" au_lname="Veríssimo">
      <Title>A Mãe de Freud</Title>
      <Title>Novas Comédias da Vida Privada</Title>
   </Author>
</AuthorsList>
```

Tabela 3.14 - Exemplo de XML gerado por uma aplicação.

Para tal, as constantes são mapeadas no começo do arquivo ou em um arquivo de constantes, conforme apresentado na Tabela 3.15.

```
private const string TG_AUTHORLIST = "Authorlist";          //OK
private const string AUTHOR = "Author";                     //NAO OK
private const string AT_AUTHOR_ID = "au_id";                //OK
private const string AT_AUTHOR_FIRST_NAME = "au_fname";     //OK
private const string ATT_AUTHOR_LAST_NAME = "au_lname";     //NAO OK
private const string ATT_TITLE = "Title";                   //NAO OK
```

Tabela 3.15 - Exemplo de código C# com os elementos mapeados em constantes.

A Tabela 3.16 mostra um trecho do código em que os elementos XML foram escritos.

```
//(...)

writer.WriteStartDocument();
writer.WriteStartElement(TG_AUTHORLIST);
writer.WriteStartElement(AUTHOR);
writer.WriteAttributeString(AT_AUID, "BR02");
writer.WriteAttributeString(AT_AUTHOR_FIRST_NAME, "Luís Fernando");
writer.WriteAttributeString(ATT_AUTHOR_LAST_NAME, "Veríssimo"); //NOK
writer.WriteElementString(ATT_TITLE, "A Mãe de Freud"); //NOK
writer.WriteEndElement();
writer.WriteElementString(ATT_TITLE,
                  "Novas Comédias da Vida Privada");  //NOK
writer.WriteEndElement();

//(...)
```

Tabela 3.16 - Exemplo de trecho de código C# que monta o XML em questão.

Os pontos do código com o comentário NOK são as linhas onde o FXCop detectaria problemas, pois o nome das constantes utilizadas nas chamadas não estão seguindo a convenção "AT_" / "TG_".

## 3.4 Considerações Finais

Através do FXCop pode-se verificar um projeto de aplicação .NET em busca de trechos onde o programador, por ventura, tenha se desviado do padrão de codificação adotado. O FXCop é capaz de analisar projetos criados em quaisquer das linguagens adotadas no Framework .NET, pois a ferramenta não analisa código-fonte, mas sim o MSIL, que é um pré-compilado e independente da linguagem.

Diversas regras interessantes estão disponíveis na instalação padrão do FXCop, e é possível ainda, criar novas regras que façam verificações mais direcionadas à necessidade da equipe, através da FXCop SDK, bastando estender as Classes e implementar os métodos de verificação desejados.

Apesar de ser possível construir muitas verificações com a SDK, é importante ressaltar que a análise do MSIL, em detrimento a analise do código-fonte, apresenta também algumas desvantagens que são percebidas durante a implementação de novas regras, como, por exemplo, a regra de mapeamento de chamadas de métodos com strings fixas, onde não foi possível garantir 100% de eficácia devido a uma característica do MSIL.

# 4 Exemplo de Uso

Para exemplificar o uso das regras construídas com o FXCop, foi utilizada uma aplicação chamada *UniversitySampleApplication*. No desenvolvimento deste sistema, foram cometidos alguns erros referentes às regras de auditoria apresentadas no capítulo 3. Ao longo deste capítulo serão exibidos os códigos-fonte do sistema, o FXCop executando as regras e apontando os problemas encontrados.

## 4.1 Sistema Exemplo

O sistema *UniversitySampleApplication* simula um software de controle de disciplinas de uma universidade. A aplicação possui uma estrutura simples, apenas contando com o controle de turmas e de geração da grade escolar por período.

Alguns problemas que foram encontrados na auditoria do código-fonte da aplicação *UniversitySampleApplication*, serão demonstrados nesta seção.

Uma boa prática de programação é definir o tipo contido em listas declaradas no código-fonte (17). No código da Tabela 4.1 pode ser verificado que a coleção *TimeTable,* pertencente à Classe *TimeTableManager,* não tem tipo definido. A regra *Utilização de listas ArrayList sem definir tipo de seus objetos*, proposta neste trabalho, procura evitar este tipo de declaração.

```
ArrayList TimeTables = new ArrayList();
```

Tabela 4.1 - Variável TimeTables declarada como ArrayList com seu tipo não estipulado.

A regra "Análise de Palavras Proibidas na Documentação do Código", apresentada na seção 3.3.3, realiza uma auditoria de palavras proibidas, previamente definidas, na documentação de métodos. Nesta regra podem ser colocados nomes de empresas, pessoas, dentre outras palavras que seja acordado como não convenientes

para aparecer na documentação. Como exemplo, utilizamos o nome da instituição de ensino *Universiexemplo* no comentário do método e a definimos como palavra proibida para ilustração da regra, como segue na Tabela 4.2.

```
/// <summary>
/// Este método retorna o horário de um determinado período.
/// O código foi copiado do sistema da Universiexemplo.
/// </summary>
/// <param name="Period"></param>
/// <param name="Year"></param>
/// <returns></returns>
public TimeTable GetTimeTable(int Period, int Year)
{…
```

Tabela 4.2 - Documentação de método citando a instituição Universiexemplo.

Na classe *ClassManager*, por esquecimento de um trecho de código comentado, o método *GetStudents* ultrapassa 20 linhas de código, como pode ser visto na Tabela 4.3. Esta quantidade de linhas de código foi estipulada como limite em outra regra de auditoria implementada para esta aplicação (ver seção 3.3.2).

```
01        public List<Student> GetStudents(String ClassID)
02        {
03
04            //Local variable that points to the current Class being analysed
05            Class tempClass = null;
06
07            //FOR Statement replaced by a FOREACH for code cleaning purposes.
08            /*
09            for (int i = 0; i < CurrentClasses.Count; i++)
10                      {
11                          tempClass = CurrentClasses[i];
12                if (tempClass.ID.Equals(ClassID)
13                {
14                    return tempClass.Students;
15                }
16                      }
17            */
18
19            foreach (Class oneClass in this.CurrentClasses)
20            {
21                if (oneClass.ID.Equals(ClassID))
22                {
23                    return oneClass.Students;
24                }
25            }
26
27            return null;
28        }
29
30    }
```

Tabela 4.3 - Método GetStudents ultrapassa 20 linhas de código como definido como limite para esta aplicação.

Outro problema encontrado na aplicação foi a passagem de um parâmetro *hard-coded* na chamada do método *GetClassesByTeacherName*, como segue na Tabela 4.4.

No caso, devem ser declaradas variáveis locais ou constantes para a passagem do parâmetro que representa o nome do professor, conforme indicado na seção 3.3.5.

```
ClassManager classManager = new ClassManager();
List<Class> classes = classManager.GetClassesByTeacherName("Fulano");
```

Tabela 4.4 - O método GetClassesByTeacherName é chamado com o parâmetro hard-coded, quando uma variável deveria ser declarada para tal.

Em um método utilizado para gerar um XML com a lista de alunos cadastrados no sistema, temos os nomes das *Tags* XML mapeados em constantes. Porém, estas constantes não estão seguindo corretamente a nomenclatura, onde devem ser utilizados os prefixos TG e AT para *Tags* e Atributos, respectivamente. A Tabela 4.5 detalha o código-fonte em questão.

```
public class StudentXMLExport
{
        private const String TG_STUDENTS = "StudentsList";
        private const String STUDENT = "Student";           //Falta do prefixo TG_

        private const String AT_UID = "UID";
        private const String Name = "Name";                 //Falta do prefixo AT_

        public void ExportStudentsXMLList(List<Student> students, String outputXmlPath)
        {
            XmlTextWriter writer = new XmlTextWriter(outputXmlPath, System.Text.Encoding.UTF8);
            writer.Formatting = System.Xml.Formatting.Indented;

            writer.WriteStartDocument();

            writer.WriteStartElement(TG_STUDENTS); //StudentsList TAG

            foreach (Student oneStudent in students)
            {
                writer.WriteStartElement(STUDENT); //Student TAG
                writer.WriteAttributeString(AT_UID, oneStudent.UID);
                writer.WriteAttributeString(Name, oneStudent.Name);
                writer.WriteEndElement(); //End Student TAG
            }


            writer.WriteEndElement(); //End StudentsList TAG

            writer.WriteEndDocument();
            writer.Close();

        }
}
```

Tabela 4.5 - O método ExportStudentsXMLList, que exporta uma lista de alunos em XML, utiliza mapeamentos de nomenclatura de tags em constantes sem utilizar os atributos TG e AT.

Na aplicação, existe ainda um método que faz a busca de todas as matérias que um aluno está inscrito em um determinado período. Este código varre a lista de períodos e turmas oferecidas, procurando a matrícula do aluno em questão. Ele apresenta

complexidade ciclomática (11) igual a 7, sendo considerado um método que exige atenção (ver seção 3.3.4). O código encontra-se detalhado na Tabela 4.6.

```
public List<Class> GetAllClassesByStudent(String studentUID, int year, int period)
{
    List<Class> returnClasses = new List<Class>();

    foreach (TimeTable timeTable in TimeTables)
    {
        if (timeTable.Year.Equals(year))
        {
            if (timeTable.Period.Equals(period))
            {
                foreach (Class oneClass in timeTable.Classes)
                {
                    foreach (Student oneStudent in oneClass.Students)
                    {
                        if (oneStudent.UID.Equals(studentUID))
                            returnClasses.Add(oneClass);
                    }
                }
            }
        }
    }

    return returnClasses;
}
```

Tabela 4.6 - O método GetAllClassesByStudent, que retorna uma lista de turmas de um determinado aluno em um determinado período, apresenta complexidade ciclomática igual a 7.

## 4.2 Exemplo de Uso das Regras

No item anterior foram apontados erros para cada regra apresentada no capítulo 3. Nesta seção, os problemas apontados serão verificados e alertados pelo FXCop, através da utilização das regras desenvolvidas neste trabalho. Foi criado um novo projeto no FXCop chamado "Análise *UniversitySampleApplication*", e nele foram adicionadas as regras customizadas e definido o alvo como sendo o projeto *UniversitySampleApplication*. A Figura 4.1 mostra o projeto FXCop aberto, com as regras carregadas e prontas para execução.

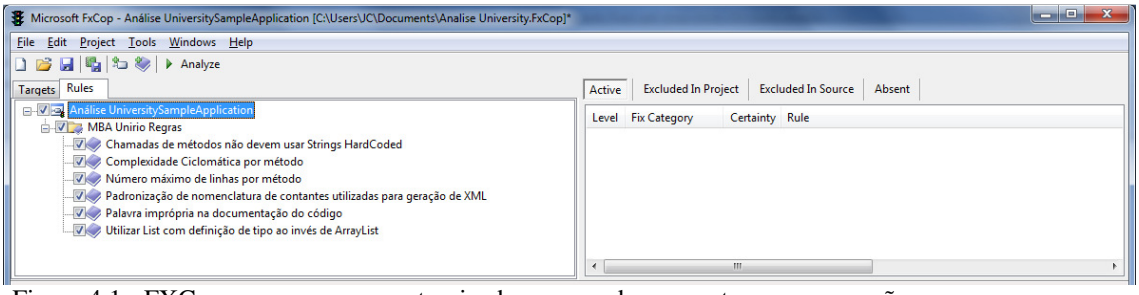


Figura 4.1 - FXCop com as regras customizadas carregadas e prontas para execução.

Após executar a análise do código com as regras, o FXCop exibe a lista dos problemas encontrados, mostrando o nome de cada regra na qual o código apresenta alguma não-conformidade. Esta listagem pode ser vista na Figura 4.2.

| Level | Fix Category | Certainty | Rule | Item |
|---|---|---|---|---|
| | Depends on Fix | 70% | Chamadas de métodos não devem usar Strings HardCoded | universitysampleapplication.exe |
| | Depends on Fix | 100% | Palavra imprópria na documentação do código | universitysampleapplication.exe |
| | **Depends on Fix** | **100%** | **Padronização de nomenclatura de contantes utilizadas para geração de XML** | **universitysampleapplication.exe** |
| | **Depends on Fix** | **100%** | **Padronização de nomenclatura de contantes utilizadas para geração de XML** | **universitysampleapplication.exe** |
| | Depends on Fix | 100% | Número máximo de linhas por método | edu.university.Model.Classes.ClassManager.#GetStudents(System.String) |
| | **Depends on Fix** | **100%** | **Complexidade Ciclomática por método** | **edu.university.Model.TimeTables.TimeTableManager.#GetAllClasses...** |
| | Depends on Fix | 60% | Utilizar List com definição de tipo ao invés de ArrayList | edu.university.Model.TimeTables.TimeTableManager.#TimeTables |

Figura 4.2 - FXCop apontando os problemas encontrados no projeto UniversitySampleApplication.

O primeiro problema encontrado foi uma chamada de método utilizando um parâmetro do tipo *String* fixo (ou *hard-coded*), que foi explicitada pela Tabela 4.4. A Figura 4.3 mostra como a regra customizada detalha o problema e sugere sua correção.

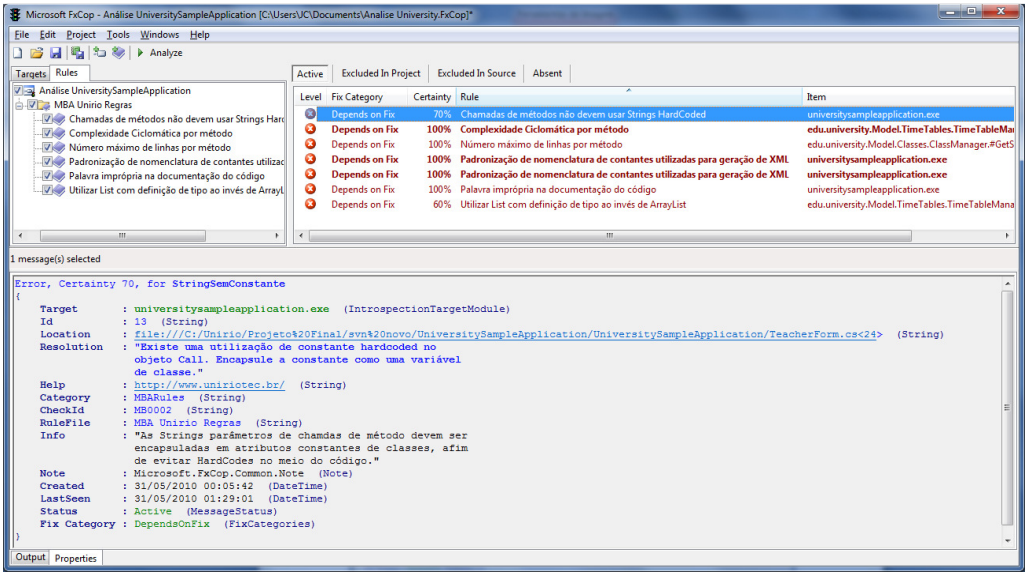


Figura 4.3- FXCop aponta não-conformidade com a regra de Chamadas de método com strings fixas.

O próximo problema apontado pelo FXCop é uma palavra imprópria na documentação do código-fonte. Para este exemplo, a palavra “Universiexemplo” foi adicionada a lista de palavras não permitidas e a documentação do método *GetTimeTable* faz a utilização desta palavra, como pode ser visto na Tabela 4.2. A Figura 4.4 mostra como a regra customizada detalha o problema e sugere sua correção.

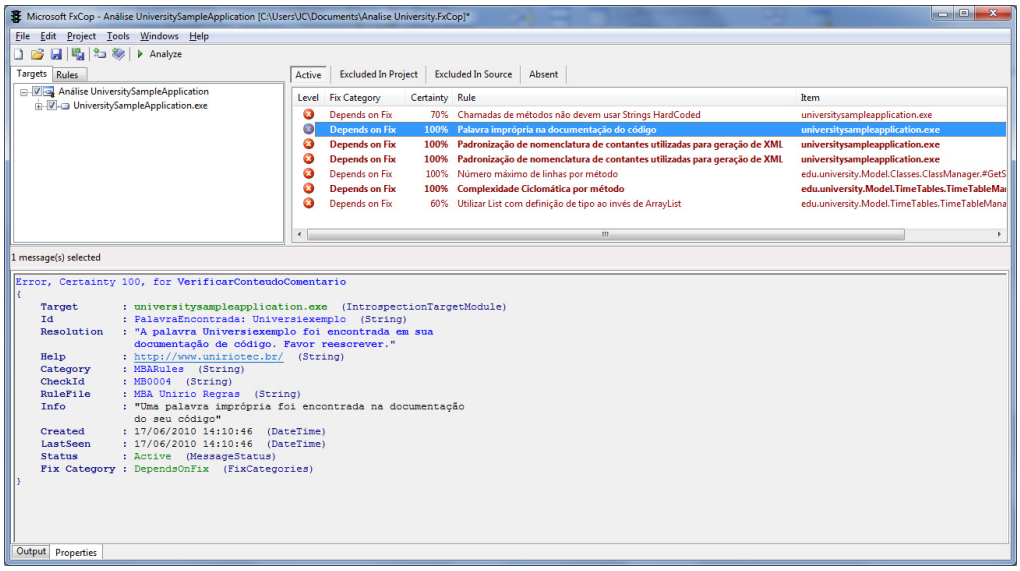


Figura 4.4 - FXCop aponta não-conformidade com a regra de palavras impróprias na documentação.

Na Tabela 4.5 foi descrito o método *ExportStudentsXMLList*, que exporta a lista de alunos cadastrados no sistema para um arquivo XML. Este método apresenta não-conformidade com a regra de padronização de nomenclatura de constantes utilizadas na montagem das *tags* e atributos de XML. As Figuras 4.5 e 4.6 mostram como a regra customizada detalha os problemas, para a constante de atributo e para a de *tag* respectivamente, e sugere suas correções.

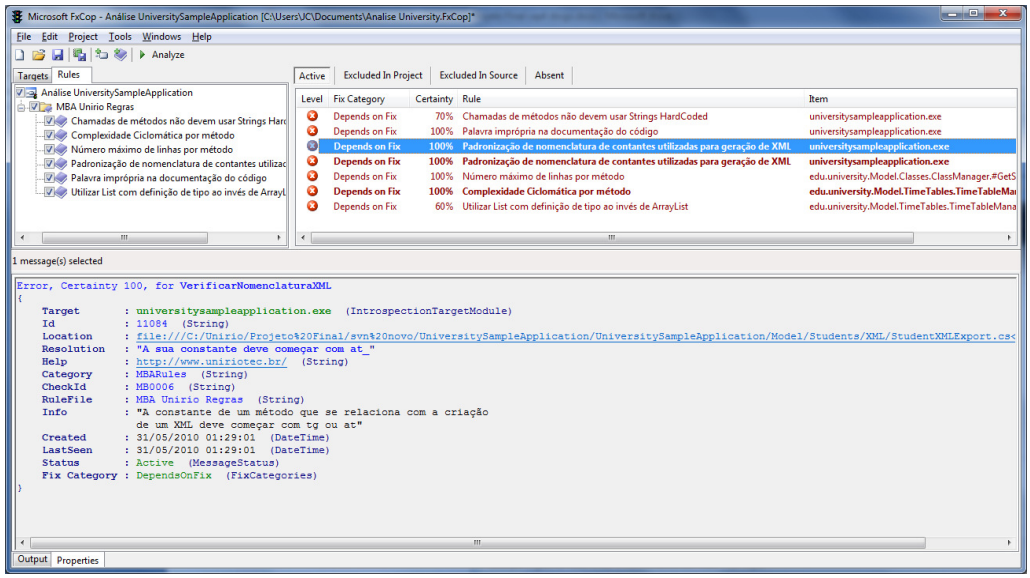


Figura 4.5 - FXCop aponta não-conformidade com a regra de padronização de nomenclatura de constantes utilizadas na geração de atributos de arquivos XML.

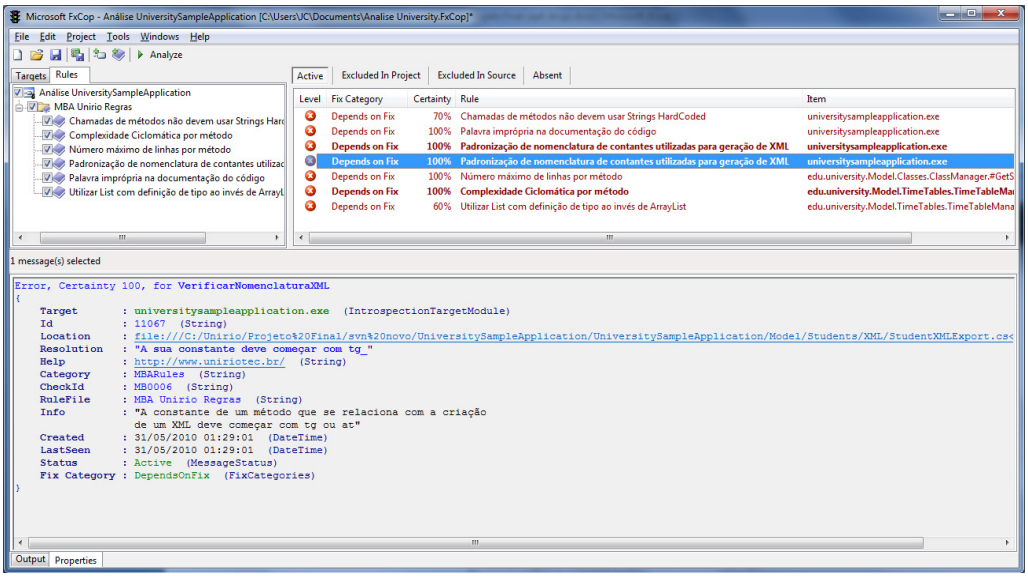


Figura 4.6 - FXCop aponta não-conformidade com a regra de padronização de nomenclatura de constantes utilizadas na geração de tags de arquivos XML.

Na Tabela 4.3 foi descrito o método *GetStudents*, que ultrapassa o limite de 20 linhas de código. Este limite foi estabelecido para exemplificar esta regra e pode ser alterado através do XML de configuração das regras desenvolvidas neste trabalho. A Figura 4.7 mostra como a regra customizada detalha o problema e sugere sua correção.

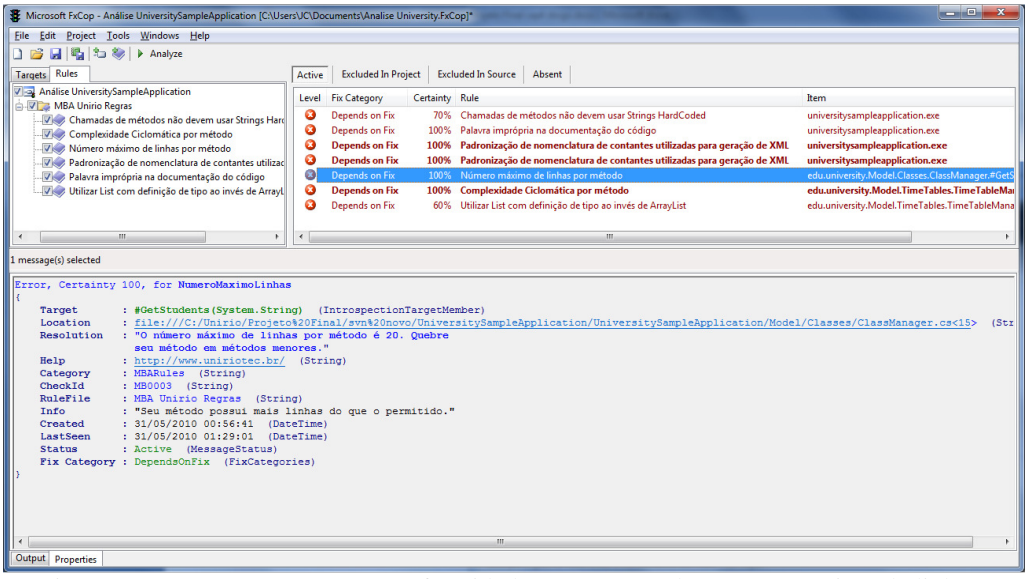


Figura 4.7 - FXCop aponta não-conformidade com a regra de número máximo de linhas.

Na Tabela 4.6 foi apresentado o método *GetAllClassesByStudent*, que retorna a lista de turmas em que um aluno está inscrito em um determinado período. O método possui uma seqüência de busca de turmas, representada por várias operações de loop e comparação, aumentando assim o número de caminhos possíveis para a execução do programa e, conseqüentemente, sua complexidade ciclomática (igual a 7). Para exemplificar esta regra, algoritmos com valor calculado maior do que 5 para a

complexidade ciclomática foram considerados como de média complexidade. A Figura 4.8 mostra como a regra customizada detalha o problema e sugere sua correção, já que a complexidade do método é maior do que a média ideal.

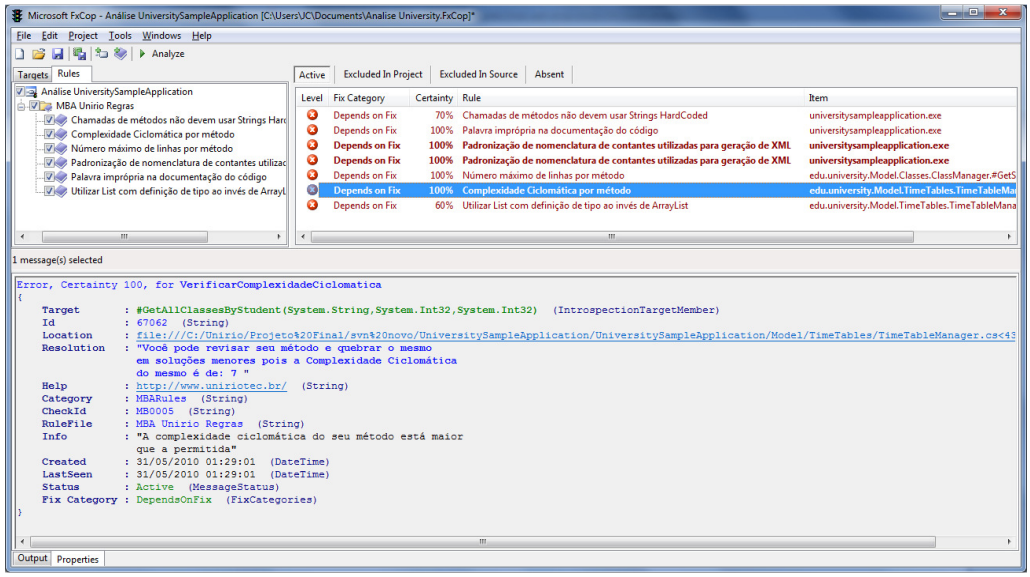

Figura 4.8 - FXCop aponta não-conformidade com a regra de complexidade ciclomática.

Na Tabela 4.1 foi apresentada uma declaração de lista, utilizando a classe *ArrayList*. Esta classe não possui suporte a definição do tipo de dados que será armazenado, permitindo armazenar qualquer tipo de dado em seu interior. Este código apresenta não conformidade com a regra "*Utilização de listas ArrayList sem definir tipo de seus objetos*", apresentada na seção 3.3.1. A Figura 4.9 mostra como a regra customizada detalha o problema e sugere sua correção.

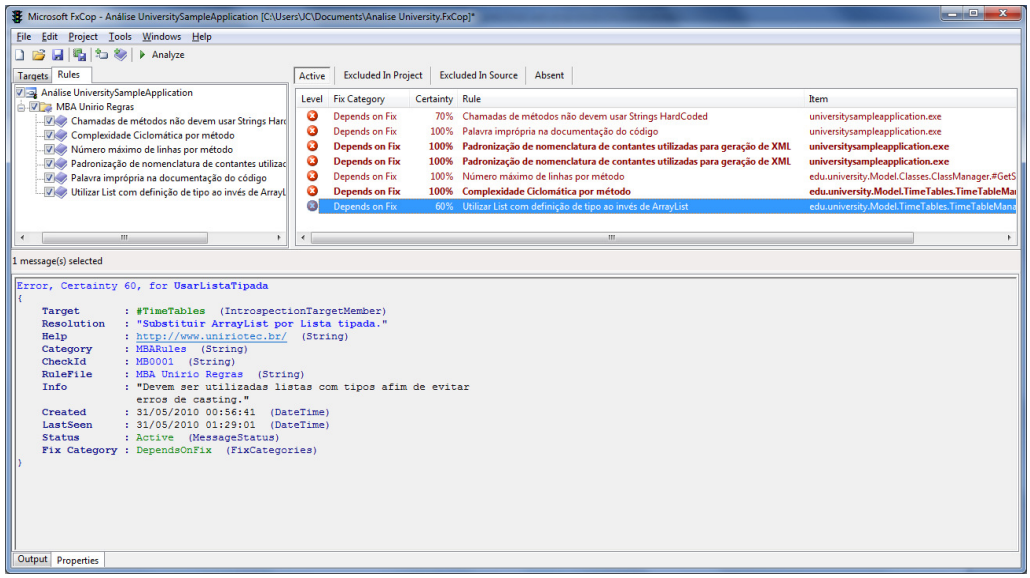

Figura 4.9 - FXCop aponta não-conformidade com a regra de uso de List<Type> ao invés de ArrayList.

## 4.3 Considerações Finais

Neste capítulo foram apresentados diversos trechos de código do sistema *UniversitySampleApplication*, construído com fins de apresentar a implementação das regras criadas no FXCop. Os trechos de código apresentam não-conformidades com as regras no capítulo 3. Os códigos foram submetidos à análise pelas regras customizadas e o resultado desta análise foi detalhado nas seções deste capítulo.

Com a utilização destas regras, pode-se concluir que para buscar qualidade e padronização de código em um projeto de software, o FXCop munido de regras personalizadas pode oferecer uma grande ajuda, automatizando o processo de verificação das regras definidas para o projeto.

# 5 Conclusão

Foi realizada neste trabalho uma pesquisa que associa padronização de projetos de desenvolvimento de software e auditoria de código-fonte. Com o crescimento da complexidade dos softwares e, por conseqüência, tamanho das equipes envolvidas, para garantir que um projeto siga os padrões determinados em sua arquitetura, pode ser adotado como solução a auditoria de código automatizada.

## 5.1 Contribuições

O trabalho apresenta conceitos de auditoria e fornece informações sobre verificação automatizada de código-fonte escrito em linguagens componentes do *.Net Framework*. Em projetos onde a equipe de desenvolvimento seja heterogênea, ferramentas de auditoria automatizada podem ser de grande ajuda em busca da manutenção de um padrão de desenvolvimento, melhorando o entendimento do código entre os diferentes desenvolvedores. Com a criação de regras customizadas, é possível utilizar este tipo de ferramenta em qualquer metodologia ou padrão de desenvolvimento de código.

Em projetos menores, ou onde o orçamento seja reduzido, pode-se utilizar a ferramenta FXCop gratuitamente. As regras criadas para esta versão do FXCop podem ser reaproveitadas pela versão paga (*Team Foundation Server*), afim de realizar análises automaticamente após cada atualização de código no repositório de versões da ferramenta.

Além disso, este trabalho apresenta-se como uma das poucas referências de estudo sobre a ferramenta FXCop na língua portuguesa, que exemplifica seu uso e fornece informações sobre o desenvolvimento de regras customizadas para verificação de diretrizes específicas de um determinado projeto, inclusive exemplos, o que o torna um documento que facilitaria a criação de novas regras por outros desenvolvedores.

## 5.2 Trabalhos futuros

Para a continuação deste trabalho podem ser construídas novas regras com base na análise de pesquisas, que identificam padrões mais comumente utilizados em processos de desenvolvimento de software, tanto em empresas como no meio acadêmico. Além disto, pode ser desenvolvida uma camada intermediária entre o servidor de controle de versão e o programador, identificando a cada *Commit* a regra que foi violada instantaneamente, acusando o padrão violado e a solução para o problema. Finalmente pode ser feita uma avaliação de integração de regras construídas com a IDE de desenvolvimento, acusando regras aferidas a cada compilação.

## 5.3 Limitações do estudo

O trabalho poderia ser aprofundado sobre o processo de desenvolvimento de regras para o FXCop, pois é difícil encontrar documentos sobre o tema, já que até a data de publicação deste trabalho, a Microsoft não disponibilizou uma documentação oficial sobre a SDK da ferramenta.

Apesar de apresentar regras customizadas úteis para diferentes cenários, estas foram disponibilizadas em pequena quantidade e um maior número de regras poderia ter sido construído.

Outra limitação apresenta-se na regra de palavras impróprias. Inicialmente, o objetivo desta regra era buscar palavras impróprias nos comentários e documentações do código-fonte. Porém, durante o estudo, percebemos que seria impossível tecnicamente verificar os comentários, pois o processo de verificação do FXCop é baseado em MSIL e este não fornece nenhum comentário do código-fonte original. Sendo assim, nossa regra é restrita a evitar palavras proibidas contidas nas somente na documentação dos métodos de um determinado projeto.

# Referências


1. **Microsoft.** MSDN - FXCop. *MSDN - Microsoft Developer Network.* [Online] http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx.

2. **SMACCHIA.COM S.A.R.L.** NDepend Tips - Advanced Documentation. *NDepend.* [Online] http://www.ndepend.com/Tips.aspx.

3. **Dias, Cláudia.** *Segurança e Auditoria da Tecnologia da Informação.* Rio de Janeiro : Axcel Books do Brasil, 2000.

4. **Joshua, Onome Imoniana.** *Auditoria de Sistemas de Informação.* s.l. : Atlas, 2008.

5. **J. Ko, Andrew, DeLine, Robert e Venolia, Gina.** Information Needs in Collocated Software Development Teams. 2007.

6. **LaToza, Thomas D., Venolia, Gina e DeLine, Robert.** Maintaining Mental Models: A Study of Developer Work Habits. 2006.

7. **Pressman, Roger.** *Engenharia de Software.* s.l. : McGraw Hill, 2006.

8. **Appleton, Brad.** Patterns and Software: Essential Concepts and Terminology. *CM Crossroads - The Configuration Management Commnunity.* [Online] 2000. http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

9. **Fowler, Martin.** *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* s.l. : Addison-Wesley, 2003.

10. **SMACCHIA.COM S.A.R.L.** Code Query Language 1.8 Specification. *NDepend.* [Online] http://www.ndepend.com/CQL.htm.

11. **McCabe, T.J.** *A complexity measure, IEEE Transactions on Software Engineering.* 1976. Vol. II.

12. **Microsoft.** Visual Studio on MSDN. *MSDN - Microsoft Developer Network.* [Online] http://msdn.microsoft.com/en-us/vstudio/default.aspx.

13. **Qamar, Kamran.** Demystifying Microsoft Intermediate Language. Part 1. *DevCity.NET.* [Online] 17 de 10 de 2002. http://devcity.net/Articles/54/1/msil_1_intro.aspx.

14. **Microsoft.** MSIL Disassembler (Ildasm.exe) . *MSDN - Microsoft Developer Network.* [Online] http://msdn.microsoft.com/en-us/library/f7dy01k1(VS.80).aspx.

15. **McConnel, Steve.** *Code Complete 2.* s.l. : Microsoft Press, 2004.

16. **Jacobson, Ivar, Booch, Grady e Rumbaugh, James.** *The Unified Software Development Process.* s.l. : Addison-Wesley Professional, 1999.

17. **Microsoft.** .NET Framework Developer Center. *MSDN.* [Online] http://msdn.microsoft.com/pt-br/netframework/default.aspx.

18. **Fowler, Martin.** Use and Abuse Cases. 1998.

19. **Pfleeger, Shari Lawrence.** *Software Engeneering: Theory and Practice.* s.l. : Prentice Hall, 2005.