número 07 • fevereiro / março 2008 • ano II • R$11,90

.mundo net

editora mundo

ISSN 1980-7996

Melhorando o design de classes através de princípios e padrões

Design Patterns e Refactoring como ferramentas para um melhor design de software.

Métricas de código e design de software

Utilizando métricas de código para melhoria no design de software.

Controle de versões com SVN e VS.NET

Subversion como controlador de versões e sua integração com o Visual Studio.NET.

Interoperabilidade de aplicações VB6 com .NET

Facilidades e estratégias de migração do legado VB6 para .NET

ViewState: amigo ou rival?

Aprenda a fazer uso consciente do recurso ViewState do WebForms.

WebForms vs MVC

Uma análise de WebForms e como os frameworks MVC Monorail e ASP.NET MVC representam uma evolução no desenvolvimento Web em .NET

Colunas

Dicas e Truques
Dicas interessantes e úteis para desenvolvimento

Porta25
Carnaval na interoperabilidade: projeto Samba

Radar.NET
Silverlight para Desenvolvedores, parte 2: mais recursos, performance e scripts gerenciados

Tools
Processo de builds automatizados

Alisson Vale

(alisson.vale@phidelis.com.br)

Tem mais de 15 anos de experiência com desenvolvimento de software. Lidera projetos de software desde 2001 com atenção especial à tecnologia.Net desde 2003. Hoje é líder de Projeto e diretor da Phidelis Tecnologia, onde lidera a equipe de desenvolvimento da solução acadêmica oferecida pela empresa. Mantém textos e idéias sobre desenvolvimento ágil de software em seu blog http://www.phidelis.com.br/blogs/alissonvale.

Utilizando Métricas de Código para Melhoria no Design de Software

Já há alguns anos temos observado, dentro da comunidade de desenvolvedores, a crença de que o código-fonte é a melhor representação de design do software que produzimos. Hoje, sabemos que é necessária uma atenção contínua no sentido de manter o código limpo, fácil de entender e de manter. A refatoração e o design orientado a testes são armas poderosas nessa luta. Neste artigo, apresentaremos uma abordagem que utiliza a checagem contínua de métricas de código como instrumento para a melhoria da qualidade do design.

O dia-a-dia de um projeto de software é muito intenso. Sempre corremos atrás de prazos, entregas e cronogramas apertados. Mesmo assim, ainda precisamos nos preocupar em manter nosso código-fonte minimamente bem estruturado, de forma que, em um futuro próximo, ele continue sendo fácil de ser alterado, estendido e compreendido. Conhecimento técnico, disciplina e estímulo à execução de revisões contínuas para melhoria de design são elementos importantes que evitam o deterioramento progressivo do potencial de evolução e manutenção do código-fonte. Esse é um desafio pelo qual nós, líderes de projeto e desenvolvedores, passamos todos os dias. Cada intervenção no código-fonte nos dá uma oportunidade de melhorá-lo ou de piorá-lo. Nesse cenário, qualquer mecanismo que nos ajude a evitar o sucessivo desgaste que o design do nosso software sofre ao longo do tempo, ou ainda, que nos imponha um ritmo de aperfeiçoamento contínuo pode ser de grande valia em nossos projetos.

O uso de métricas como instrumento de monitoramento e avaliação contínua do código-fonte pode ser poderoso no combate à crescente degradação que ele sofre ao longo do tempo. Mesmo considerando que a qualidade do design de um software é de avaliação subjetiva, não há números capazes de médi-lá, há determinados parâmetros que nos levam a "perceber" um bom design. Tais parâmetros podem ser obtidos por alguns tipos de medições que, quando realizadas periodicamente, revelam informações importantes sobre o processo de evolução (ou deterioração) que o design do software sofre. Em termos práticos, ferramentas podem ser organizadas e integradas de forma a fazer coletas periódicas de métricas de código, sujeitá-las a certas análises automatizadas e, assim, tomar algum tipo de decisão baseada na melhora ou na piora da qualidade de design de um novo código introduzido no produto. Neste artigo, discutiremos algumas técnicas e ferramentas para coleta de métricas de design. Tais técnicas nos ajudarão a criar uma espécie de "estado de alerta" que nos manterá preocupados com o design mesmo nas situações adversas em que a pressão e os prazos de entrega insistem em nos afastar das melhores soluções.

A importância do código-fonte como representação de design

No final da década de 90, houve uma reação à incômoda tendência que vinha se apresentando nos anos anteriores na direção de subvalorizar o código-fonte como importante elemento de representação do design de uma aplicação. Métodos e metodologias convergentes com o Movimento Ágil [1] trouxeram o código-fonte e quem o escreve para o mais alto grau de importância dentro de um projeto de software. Desenvolvedores experientes e consagrados, como Martin Fowler, Kent Beck e Ward Cunningham, formalizaram técnicas que nos ajudariam a entender por que o código-fonte é o principal elemento de qualidade em nosso projeto de software, e como ele pode ser trabalhado para permitir que o software e suas funcionalidades evoluam de maneira sustentável e segura.

Quando se pensa em design de software, é comum a associação direta com modelos de classes, diagramas de seqüência e uma série de outras representações que a UML e outras técnicas de modelagem nos oferecem. Ao contrário do que muitos pensam, o código-fonte é um artefato que tem a mesma natureza de um modelo ou de um diagrama. Ambos são representações. São, mais precisamente, especificações que descrevem, em diferentes níveis de abstração, regras de funcionamento ou elementos estruturais que geram um comportamento repetido quando automatizados. Tais especificações, na forma de regras e elementos lógicos abstratos, formam o que conhecemos por design de software.

> Se de um lado um diagrama ou modelo é mais fácil de ler e de se utilizar em sessões de análise para discutir, comunicar ou divulgar um processo de negócio, o código-fonte é, por sua vez, a representação mais fidedigna do seu comportamento real, bem como a representação executável mais importante.

Na hora de agregar valor para o processo de negócio da sua empresa ou de seus clientes, quem de fato precisará ser modificado ou estendido é o seu código-fonte. Modelos e diagramas, por sua vez, são importantes porque nos ajudam a pensar sobre "o quê" e sobre "como" alterar ou implementar uma característica de sistema. A representação do design do software, quando observada em termos de modelos, ajudará a organizar e a planejar uma estratégia para executar uma intervenção no código-fonte. É por isso que desenvolvedores ágeis, em especial, valorizam mais o processo de modelagem do que o modelo em si. Tanto o código-fonte quanto os modelos e diagramas são instrumentos de mesma natureza que possuem diferentes papéis. Porém, para efeitos de aplicabilidade das idéias descritas neste artigo, é importante que o leitor reconheça que partimos da premissa de que o código-fonte é a representação mais importante do design do software.

Um bom design

Se quando falamos em código-fonte falamos em design, como podemos trabalhar o código-fonte para que ele se torne uma representação eficiente o bastante para nos ajudar a manter e evoluir nosso software rapidamente? Na verdade, quando fazemos isso trabalhamos para conseguir um bom design. O americano James Shore fala sobre a idéia do "bom design" no recém-lançado livro The Art Of Agile Development [2]: "Um bom design minimiza o tempo requerido para criar, modificar e manter o software enquanto resulta em um nível aceitável de performance". A avaliação sobre o que é ou não um bom design é abstrata. Design também está relacionado à elegância, ao bom senso e à clareza. É difícil qualificá-lo ou desqualificá-lo por meio de análises objetivas ou métricas. Não há consenso entre os especialistas quanto ao valor que uma determinada métrica deveria ter para considerá-lo ruim, bom ou muito bom. Como diz Martin Fowler "nenhum conjunto de métricas pode se comparar à intuição humana" [3]. No entanto, métricas podem ser muito úteis se utilizadas de forma a nos ajudar a encontrar ou evitar potenciais problemas de design. Suponha, por exemplo, que sua experiência lhe diga que classes muito grandes são elementos sintomáticos de um problema de design. Suponha também que, com a ajuda de uma ferramenta capaz de coletar informações sobre o seu código-fonte, você descubra que o conjunto completo de classes do seu sistema tem em média 7,25 métodos por classe. Essa não é uma informação muito relevante se tomada isoladamente. Se, no entanto, em uma segunda medição, algumas semanas depois, esta média sobe para 8,7, você obteve uma informação valiosíssima! O fato de novos métodos terem sido criados em uma ou outra classe não é um problema. Mas o aumento de 20% no número médio de métodos por classe de todo o seu sistema é um grande acontecimento! É claramente um indicador de explosão de métodos e de descuido com o design da aplicação. Algo certamente acontece durante o dia-a-dia de desenvolvimento que afeta o design do código-fonte. Quais foram as intervenções que nos levaram a esse crescimento brusco? Como usar esse tipo de informação a nosso favor e evitar que isso aconteça novamente?

> É o senso de constante observância a determinados comportamentos de evolução no design do software que nos levará a sermos capazes de atuar preventivamente na manutenção de sua qualidade.

Que métricas utilizar?

Há dezenas de métricas que podem ser facilmente coletadas por todo o tipo de ferramenta de análise de código. Cada métrica tem sua utilidade, mas nem todas serão úteis para observar o design de sua aplicação. É importante saber quais são as métricas que vão ajudar e que tipo de informação elas irão fornecer para essa análise. A melhor forma de descobrir métricas interessantes não é procurando-as em ferramentas, mas nas sensações que nos incomodam quando nos deparamos com um código que pode ser melhorado. Ou seja, nos próprios princípios básicos da orientação a objetos e nos famosos "maus cheiros" criados por Kent Beck e Martin Fowler e descritos no livro Refactoring – Improving the Design Of Existing Code [3] . Os maus cheiros são como pistas que sugerem um trecho do código com alta probabilidade de enfrentar um problema de design. Nesse livro (leitura obrigatória para desenvolvedores), Martin Fowler relaciona os maus cheiros e descreve as possíveis técnicas que poderiam ser utilizadas para removê-los do código e, com isso, melhorar o design. O termo Refatoração ficou conhecido desde então como a técnica de alterar o código-fonte com o intuito de melhorar o seu design sem modificar o comportamento do software. Métricas podem dar pistas sobre a existência de maus cheiros que nos ajudarão a identificar as refatorações necessárias no nosso código.

> Tudo é uma questão de procurar medir o que você não quer ter no seu código-fonte.

Os tópicos a seguir mostram alguns exemplos sobre os problemas mais comuns que poderiam ser evitados por meio do acompanhamento de métricas de design.

Código duplicado

É um sintoma muito comum em códigos com o design comprometido. Ocorre quando um mesmo trecho de código aparece duplicado em dois ou mais lugares. Talvez o cheiro mais forte no design de um software. Não há muita chance de se encontrar código duplicado e isso não representar de fato um grande problema. A remoção de replicações deve ser um "mantra" na cabeça de qualquer desenvolvedor.

Saiba Mais

Você deve encontrar boas explicações sobre males e soluções para códigos duplicados em Avoiding Repetition [10], em Pragmatic Programmer – Don't Repeat Yourself [11] e em Extreme Programming Explained – Once and Only Once – [12].

Se a lógica está em mais de um lugar e precisa ser alterada, essa alteração demandará mais esforço. Esse esforço pode ser materializado em mais codificação, mais testes, ou na correção de mais bugs causados pela desnecessária complexidade da alteração.

A Listagem 1 apresenta um exemplo de dois métodos que geram um mesmo relatório de extrato de pedidos em formatos diferentes. O design desse código é ruim por vários motivos, mas à primeira vista, é a duplicação de código que nos chama mais atenção. É bem provável que essa classe, quando da sua criação, só imprimisse extratos de pedidos em um único formato. Em algum momento houve a necessidade de gerar a mesma informação em um segundo formato. O código foi então copiado e editado para executar a nova função. É um ótimo exemplo de uma prática muito comum e extremamente danosa ao design do software.

Esse tipo de problema nos remete a uma primeira métrica útil que podemos utilizar para observar problemas de design: o número de linhas duplicadas. Uma ferramenta de análise de similaridade, como o Simian [4], pode ser muito útil no processo de recuperação de informações relacionadas com essa métrica. A figura 1 mostra o Simian detectando o problema no arquivo da Listagem 1

A solução imediata para o problema é aparentemente simples. Primeiro identifica-se o que é comum e o que varia. Isola-se o que é comum e decide-se que variação utilizar quando for necessário. Vamos ver mais adiante que, dependendo da solução aplicada, esta poderá gerar outros problemas de design. Uma possível solução para o problema da duplicação é indicada na Listagem 2.

Listagem 1. Exemplo de duplicação de código.

```
public class ExtratoPedido
{
    public string EmitirEmFormatoTexto()
    {
        string extrato = "";
        extrato += "=====================================";
        extrato += "Extrato do Pedido" + Environment.NewLine;
        extrato += "=====================================";
        decimal valorTotal = 0;
        foreach (ItemPedido eachItem in _pedido.Items)
        {
            extrato += "Produto: " + eachItem.ObterNomeDoProduto()
                + Environment.NewLine;
            extrato += "Quantidade: " + eachItem.Quantidade.ToString()
                + Environment.NewLine;
            extrato += "Valor: " + eachItem.Valor.ToString("#0,00")
                + Environment.NewLine;
            valorTotal += eachItem.Valor;
        }
        extrato += "=====================================";
        extrato += "Total: " + valorTotal.ToString("#0,00");
        extrato += "=====================================";
        extrato += "Emitido em " + DateTime.Now.ToString();
        return extrato;
    }

    public string EmitirEmFormatoHtml()
    {
        string extrato = "";
        extrato += "<hr><h1>";
        extrato += "Extrato do Pedido" + "<br>";
        extrato += "</h1><hr>";
        decimal valorTotal = 0;
        foreach (ItemPedido eachItem in _pedido.Items)
        {
            extrato += "Produto: " + eachItem.ObterNomeDoProduto()
                + "<br>";
            extrato += "Quantidade: " + eachItem.Quantidade.ToString()
                + "<br>";
            extrato += "Valor: " + eachItem.Valor.ToString("#0,00")
                + "<br>";
            valorTotal += eachItem.Valor;
        }
        extrato += "<hr>";
        extrato += "Total: " + valorTotal.ToString("#0,00");
        extrato += "<hr>";
        extrato += "<i>Emitido em " + DateTime.Now.ToString()
            + "</i>";
        return extrato;
    }
}
```

Na Listagem 2 nós movemos o que é comum para um único método e, dentro dele, fizemos o tratamento em cima do que varia. Repare, no entanto, que, apesar de a questão da duplicação ter sido resolvida, isso não significa que os problemas de design tenham acabado. É possível que uma refatoração resolva um problema, mas evidencie outro. No caso específico da Listagem 2, o código não está mais duplicado. Isso é um bom sinal, pois alterações na rotina agora poderão ser realizadas em um único ponto. No entanto, o novo método criado absorveu toda a complexidade relacionada com as variações de formato do resultado. A conseqüência disso foi a geração de um método longo e complexo assunto para os tópicos descritos a seguir.

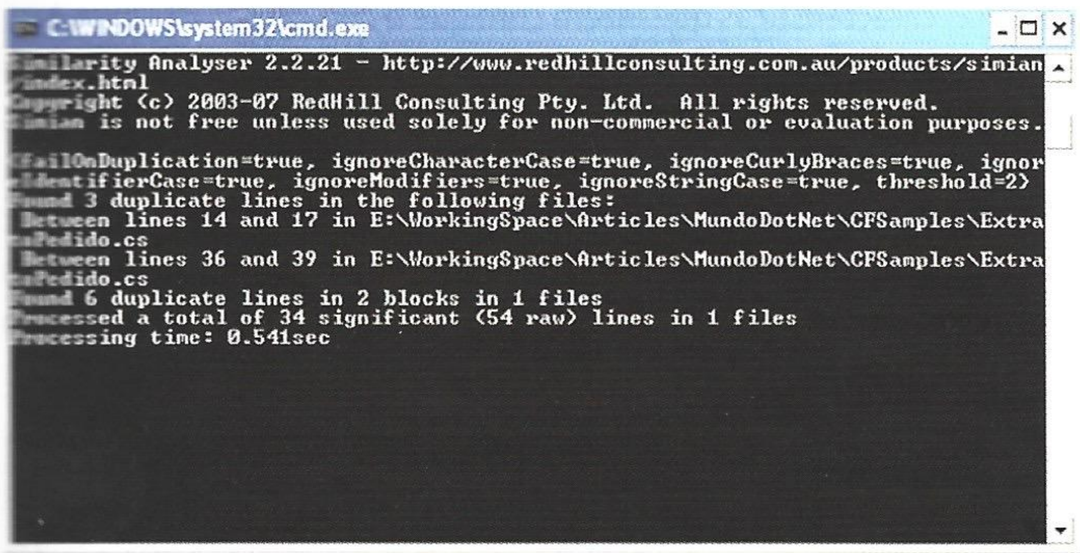


Figura 1. Detecção do número de linhas duplicadas utilizando o Simian.

Listagem 2. Duplicação de código resolvida.

```
public class ExtratoPedido
{
    const string FORMATO_HTML = "html";
    const string FORMATO_TXT = "txt";
    public string EmitirEmFormatoTexto()
    {
        return Emitir(FORMATO_TXT);
    }

    public string EmitirEmFormatoHtml()
    {
        return Emitir (FORMATO_HTML);
    }

    private string Emitir(string formato)
    {
        string extrato = "";
        extrato += ImprimeSeparador(formato);
        string enter = "";
        if (formato == FORMATO_HTML)
            enter = "<br>";
        else
            enter = Environment.NewLine;
        extrato += "Extrato do Pedido" + enter;
        extrato += ImprimeSeparador(formato);
        decimal valorTotal = 0;
        foreach (ItemPedido eachItem in _pedido.Items)
        {
            extrato += "Produto: " + eachItem.ObterNomeDoProduto()
                + enter;
            extrato += "Quantidade: " + eachItem.Quantidade.ToString()
                + enter;
            extrato += "Valor: " + eachItem.Valor.ToString("#0,00")
                + enter;
            valorTotal += eachItem.Valor;
        }
        extrato += ImprimeSeparador(formato);
        extrato += "Total: " + valorTotal.ToString("#0,00");
        extrato += ImprimeSeparador(formato);

        if (formato == FORMATO_HTML)
            extrato += string.Format("{0}Emitido em {1}{2}", "<i>",
                DateTime.Now.ToString(), "</i>");
        else
            extrato += string.Format("Emitido em {1}",
                DateTime.Now.ToString());
        return extrato;
    }
    private string ImprimeSeparador(string formato)
    {
        string separador = "==============================";
        if (formato == FORMATO_HTML)
            separador = "<hr>";
        return separador;
    }
}
```

Métodos longos e complexidade

Ocorre quando há muito código escrito em um único método. Quando um método tem muitas linhas de código ele certamente quebrará vários princípios de orientação a objetos. Um método longo dificulta a leitura e o entendimento do que ele faz. Ele incentiva novas duplicações e prejudica o reuso. A regra geral é mantê-los curtos o suficiente para fazerem apenas aquilo que eles dizem que fazem em seus nomes.

Há várias ferramentas que nos dão a possibilidade de analisar métricas de tamanho dos nossos métodos. A ferramenta SourceMonitor [5] é uma boa opção para obter esses dados. Para esse propósito ela nos ajudará fornecendo métricas como o Número médio e absoluto de statements por método, além de métricas de complexidade máxima e média. As figuras 2 e 3 mostram medições realizadas no código respectivamente antes e depois da remoção de duplicação ilustrada na Listagem 2.

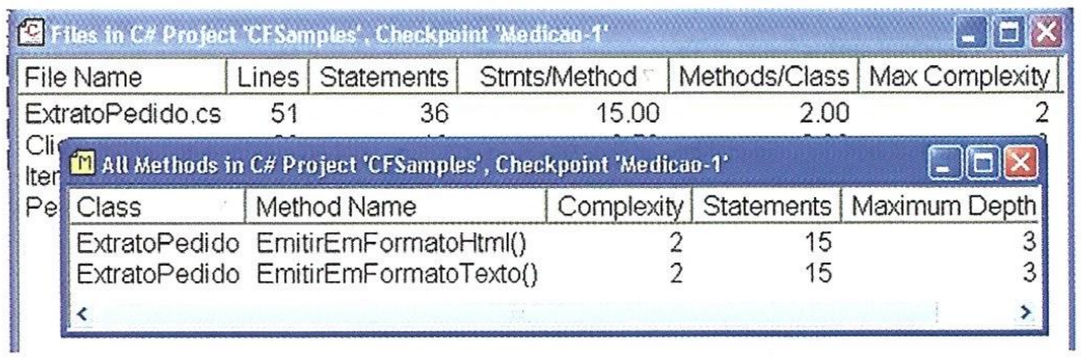


Figura 2. Métricas obtidas com o SourceMonitor no código-fonte com duplicação.

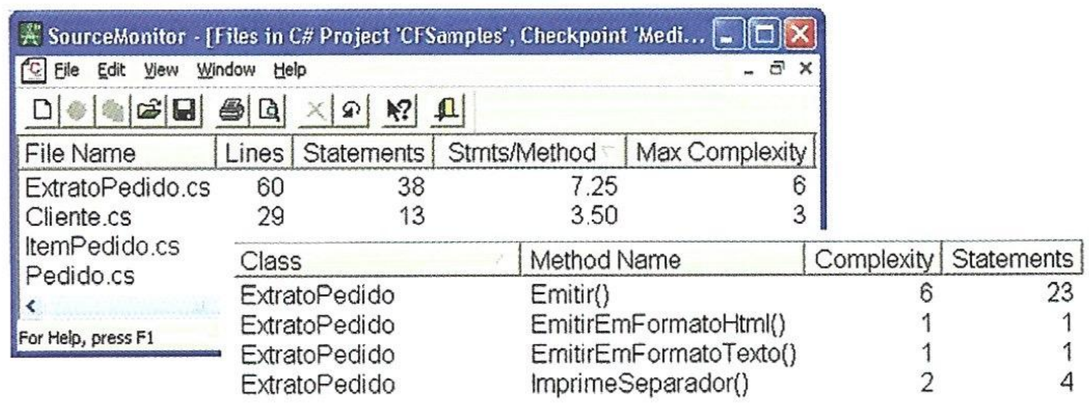


Figura 3. Métricas após remoção de duplicação.

É difícil definir um número de statements a partir do qual um método passe a ser considerado longo. No entanto, 23 certamente indica um. Obter uma lista com outros métodos que possuam métrica semelhante será um bom ponto de partida para planejar ações de melhoria no design. É comum também que o aumento no tamanho dos métodos venha acompanhado do aumento nas métricas de complexidade. O SourceMonitor utiliza o algoritmo de complexidade sugerido por Steve McConnel no livro clássico Code Complete [6]. Cada statement que abre um bloco lógico no código (if, else, for, while, try, catch, switch e outros) acrescenta 1 à contagem. Cada operador lógico & ou | dentro desses statements também acrescenta 1 à contagem. Na verdade, o que se vê é que o comportamento das métricas acaba evidenciando essa "má impressão" subjetiva que temos quando nos deparamos com métodos longos e complexos.

Uma maneira mais rápida de detectar que um método está longo é analisar a quantidade de tempo que você gasta olhando para ele quando precisa entendê-lo. Métodos curtos e objetivos poderão ser compreendidos em alguns poucos segundos. Veja na Listagem 3 como é mais fácil entender o que método "Emitir" faz depois que ele é refatorado.

Listagem 3. Resultado de refatoração em método longo.

```
public class ExtratoPedido
{
    const string FORMATO_HTML = "html";
    const string FORMATO_TXT = "txt";
    public string EmitirEmFormatoTexto()
    {
        return Emitir(FORMATO_TXT);
    }
    public string EmitirEmFormatoHtml()
    {
        return Emitir(FORMATO_HTML);
    }
    private string Emitir(string formato)
    {
        string extrato = EmitirCabecalho(formato);
        extrato += EmitirListaDeProdutos(formato);
        extrato += EmitirRodape(formato);
        return extrato;
    }
    private string EmitirListaDeProdutos(string formato)
    {
        decimal valorTotal = 0;
        string lista = "";
        foreach (ItemPedido eachItem in _pedido.Items)
        {
            lista += "Produto: " + eachItem.ObterNomeDoProduto()
                + ObterQuebraDeLinha(formato);
            lista += "Quantidade: " + eachItem.Quantidade.ToString()
                + ObterQuebraDeLinha(formato);
            lista += "Valor: " + eachItem.Valor.ToString("#0,00")
                + ObterQuebraDeLinha(formato);
            valorTotal += eachItem.Valor;
        }
        lista += EmitirTotal(valorTotal, formato);
        return lista;
    }
    private string EmitirCabecalho(string formato)
    {
        string cabecalho = ImprimirSeparador(formato);
        cabecalho += "Extrato do Pedido" + ObterQuebraDeLinha(formato);
        cabecalho += ImprimirSeparador(formato);
        return cabecalho;
    }
    private string EmitirRodape(string formato)
    {
        string rodape = "";
        if (formato == FORMATO_HTML)
            rodape += string.Format("{0}Emitido em {1}{2}", "<i>",
                DateTime.Now.ToString(), "</i>");
        else
            rodape += string.Format("Emitido em {1}",
                DateTime.Now.ToString());
        return rodape;
    }
    private string EmitirTotal(decimal valorTotal, string formato)
    {
        string total = ImprimirSeparador(formato);
        total += "Total: " + valorTotal.ToString("#0,00");
        total += ImprimirSeparador(formato);
        return total;
    }
    private string ImprimirSeparador(string formato)
    {
        string separador = "==============================";
        if (formato == FORMATO_HTML)
            separador = "<hr>";
        return separador;
    }
    private string ObterQuebraDeLinha(string formato)
    {
        string quebra = "";
        if (formato == FORMATO_HTML)
            quebra = "<br>";
        else
            quebra = Environment.NewLine;
        return quebra;
    }
}
```

Há várias características que fazem com que o código da Listagem 3 re
presente um melhor design do que aquele da Listagem 2. Os método
agora realmente fazem o que eles dizem que fazem; eles são mais fácei
de serem testados por meio de testes unitários; também são de leitur
mais fácil (a maioria pode ser compreendida com uma rápida olhada
aumenta a possibilidade de reuso dos novos métodos criados; e o mai
importante, é mais fácil de ser alterado. Vamos dar uma nova olhad
nas métricas (figura 4).

Files in C# Project 'CFSamples', Checkpoint 'Medicao-3'

File Name	Lines	Statements	Stmts/Met...	Max Complexity
ExtratoPedido.cs	82	53	4.33	3

All Methods in C# Project 'CFSamples', Checkpoint 'Medicao-3'

Class	Method Name	Complexity	Statements	Maximum Depth	Calls
ExtratoPedido	Emitir()	1	4	2	3
ExtratoPedido	EmitirCabecalho()	1	4	2	3
ExtratoPedido	EmitirEmFormatoHtml()	1	1	2	1
ExtratoPedido	EmitirEmFormatoTexto()	1	1	2	1
ExtratoPedido	EmitirListaDeProdutos()	2	9	3	7
ExtratoPedido	EmitirRodape()	3	6	3	4
ExtratoPedido	EmitirTotal()	1	4	2	3
ExtratoPedido	ImprimirSeparador()	2	4	3	0
ExtratoPedido	ObterQuebraDeLinha()	3	6	3	0

Figura 4. Métricas depois de refatoração em método longo

Veja como o número médio de statements por método foi consideravel
mente reduzido na classe ExtratoPedido. Mais interessante é a brusca re
dução da métrica de complexidade da classe, o que corrobora com nossa
argumentação de que temos agora uma classe mais simples. Apesar de
estarmos na direção de melhorarmos o design do nosso código, há ainda
o que fazer. Uma conseqüência direta do uso de métodos pequenos é a
degradação de outra métrica importante que veremos a seguir.

Classes grandes

Uma classe grande também não exala um cheiro muito bom. E é exata-
mente essa a sensação que um desenvolvedor experiente tem quando
se depara com um código como o da Listagem 3. Classes com muito
código têm uma boa chance de ter um problema de coesão e de violar
o Princípio da Responsabilidade Única (Single Responsibility Principle).
O SRP nos remete à idéia de que uma classe deve ter um único motivo
para ser alterada. Na Listagem 3, a classe ExtratoPedido tem vários
motivos para mudar. Mudanças nos formatos HTML e Texto ou suporte
a novos formatos. Outros princípios importantes também são violados
nesse trecho de código de modo que nossa melhor opção é continu-
ar refatorando. Uma métrica que pode ser utilizada para detectar ou
analisar este problema é a de número de métodos por classe também
fornecida pelo SourceMonitor. Uma possível solução é apresentada na
Listagem 4.

Veja que agora nosso código foi distribuído em uma nova hierarquia de
classes que encapsula o conceito de formatação de um extrato de pedi-
do. Todo o código que variava de acordo com o formato de saída agora
está encapsulado em sua própria classe. A criação de um novo formato
(rtf, por exemplo) demandará apenas a criação de uma nova subclasse,
não sendo necessária nenhuma alteração no restante do código. Esse
é o objetivo do princípio de orientação a objetos que nos diz que uma
classe deve estar aberta para extensões e fechada para modificações
(Open-Closed Principle). Repare também como todos os comandos "If"
simplesmente sumiram do código. Além de reduzir a complexidade,
a redução na quantidade de caminhos lógicos facilita a testagem dos
métodos. Vamos dar uma nova olhada na evolução das métricas depois
dessa modificação (figura 5).

Listagem 4. Resultado de refatoração em método longo.

```
public class ExtratoPedido
{
    public string EmitirEmFormatoTexto()
    {
        FormatadorExtratoPedido formatador =
            new FormatadorExtratoPedidoTxt();
        return Emitir(formatador);
    }
    public string EmitirEmFormatoHtml()
    {
        FormatadorExtratoPedido formatador =
            new FormatadorExtratoPedidoHtml();
        return Emitir(formatador);
    }
    private string Emitir(FormatadorExtratoPedido formatador)
    {
        string extrato = formatador.EmitirCabecalho();
        extrato += EmitirListaDeProdutos(formatador);
        extrato += formatador.EmitirRodape();
        return extrato;
    }
    protected string EmitirListaDeProdutos(FormatadorExtratoPedido
        formatador)
    {
        decimal valorTotal = 0;
        string lista = "";
        foreach (ItemPedido eachItem in _pedido.Items)
        {
            lista += "Produto: " + eachItem.ObterNomeDoProduto()
                + formatador.ObterQuebraDeLinha();
            lista += "Quantidade: " + eachItem.Quantidade.ToString()
                + formatador.ObterQuebraDeLinha();
            lista += "Valor: " + eachItem.Valor.ToString("#0,00")
                + formatador.ObterQuebraDeLinha();
            valorTotal += eachItem.Valor;
        }
        lista += formatador.EmitirTotal(valorTotal);
        return lista;
    }
}
public abstract class FormatadorExtratoPedido
{
    public abstract string ImprimirSeparador();
    public abstract string EmitirRodape();
    public abstract string ObterQuebraDeLinha();
    public string EmitirCabecalho()
    {
        string cabecalho = ImprimirSeparador();
        cabecalho += "Extrato do Pedido"
            + ObterQuebraDeLinha();
        cabecalho += ImprimirSeparador();
        return cabecalho;
    }
    public string EmitirTotal(decimal valorTotal)
    {
        string total = ImprimirSeparador();
        total += "Total: " + valorTotal.ToString("#0,00");
        total += ImprimirSeparador();
        return total;
    }
}

public class FormatadorExtratoPedidoTxt
    : FormatadorExtratoPedido
{
    public override string ImprimirSeparador()
    {
        return "========================================";
    }
    public override string EmitirRodape()
    {
        return string.Format("Emitido em {1}",
            DateTime.Now.ToString());
    }
    public override string ObterQuebraDeLinha()
    {
        return Environment.NewLine;
    }
}

public class FormatadorExtratoPedidoHtml
    : FormatadorExtratoPedido
{
    public override string ImprimirSeparador()
    {
        return "<hr>";
    }
    public override string EmitirRodape()
    {
        return string.Format("{0}Emitido em {1}{2}", "<i>",
            DateTime.Now.ToString(), "</i>");
    }
    public override string ObterQuebraDeLinha()
    {
        return "<br>";
    }
}
```

Files in C# Project 'CFSamples', Checkpoint 'Medicao-4'

File Name	Lines	Statements	Stmts/Method	Methods/Class	Max Complexity	% Comments	%
ExtratoPedido.cs	41	26	4.25	4.00	2	0.0	
FormatadorExtratoPedido.cs	23	15	1.60	5.00	1	0.0	
FormatadorExtratoPedidoHtml.cs	19	9	1.00	3.00	1	0.0	
FormatadorExtratoPedidoTxt.cs	19	9	1.00	3.00	1	0.0	
ItemPedido.cs	21	8	1.00	3.00	1	0.0	
Pedido.cs	35	19	1.00	4.00	1	0.0	

Figura 5. Métricas depois da reorganização do modelo de classes.

Veja como agora as métricas de statements por método, métodos por classe e complexidade apresentam valores mais uniformes, com números baixos. A tabela 1 mostra a evolução das métricas nas medições de cada listagem para a Classe ExtratoPedido.

Métrica	Medição 1	Medição 2	Medição 3	Medição 4
Linhas de código duplicado	6*	0	0	0
Número de statements por método	16*	7,25*	4,33	4,25
Número de métodos por classe	2	4	9*	4
Complexidade máxima	2	6*	3	2
				* * Números distoantes

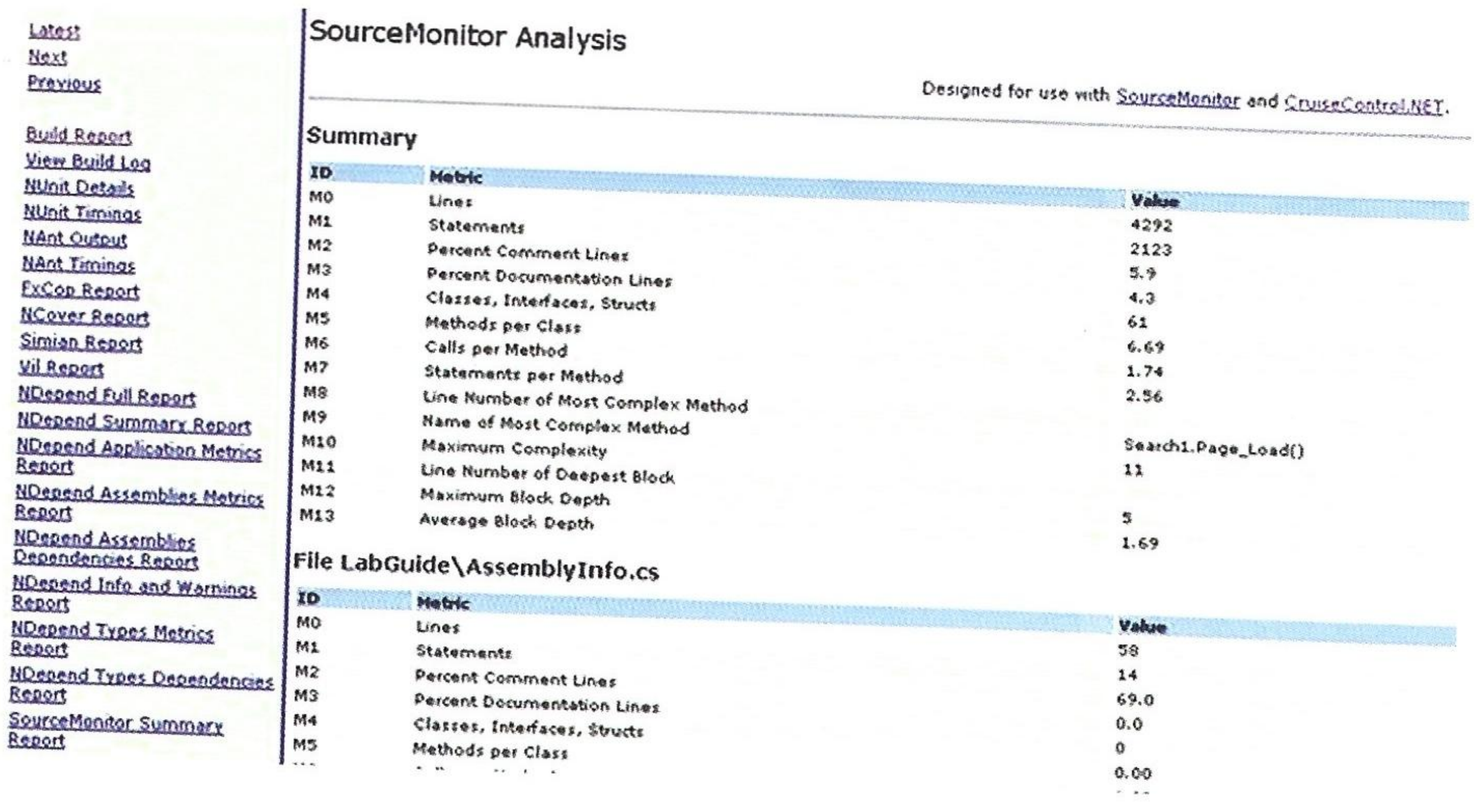
Tabela 1. Resumo das medições realizadas.

A tabela 1 apresenta um modo interessante de analisar as medições. Repare que sempre há um número destoante enquanto o design não se estabiliza. Cada distorção está associada a um problema específico. Na medição 1 havia o problema do código duplicado, o que é revelado na métrica. No entanto, ainda nessa medição havia outra métrica que apontava para outro problema. Como o problema do código duplicado apresentava um cheiro mais forte, nós nem precisamos analisar a outra perspectiva do problema. Naturalmente, ela apareceu na segunda medição, o que nos fez resolvê-lo. Ao resolvê-lo e realizar a terceira medição, outro sintoma se evidenciou. Na verdade, o problema na terceira medição já existia na primeira e na segunda, apenas não estava evidente. Assim, embora as métricas não sejam capazes de substituir a capacidade humana de perceber os "maus cheiros" no código, elas certamente podem ser mais um bom instrumento na busca por um melhor design.

Estratégias sistêmicas

Incorporar a análise de métricas no dia-a-dia da forma como foi exemplificado neste artigo não é algo muito fácil. A quantidade de classes e métodos existentes no código-fonte de sistemas de qualquer tamanho é grande o suficiente para inviabilizar uma análise cotidiana manual. O código-fonte de um software é uma rede complexa, cujo comportamento evolutivo é de difícil controle. Intervenções pontuais para melhoria serão ineficientes. É preciso agir sistemicamente, capacitando os desenvolvedores em técnicas que os permita identificar um bom design (ou um ruim) durante o trabalho de desenvolvimento, não depois. Prevenir um design ruim antes que o código saia do computador do desenvolvedor é a melhor forma de mantê-lo saudável.

Se o seu projeto já sofre de problemas de design e a capacitação dos desenvolvedores é algo que ainda está em andamento, você pode utilizar uma estratégia de controle que evite que novos problemas surjam e, ao mesmo tempo, que aponte onde estão os problemas mais crônicos. Uma das maiores dificuldades em se monitorar esses cenários de degradação é a disciplina necessária para se manter vigilância sobre as variações nas métricas. Se essa observação for manual e sob demanda, a chance de não ser efetiva será muito grande. Uma possível saída é tentar automatizar essas medições, tornando-as tão freqüentes quanto possível. Isso é importante pois cria as condições para a execução freqüente da coleta de métricas, sua análise e conseqüente tomada de decisão sobre a qualidade do código que é introduzido. Ferramentas como o Cruise Control.Net [7], o NAnt [8] ou o MsBuild [9] são boas opções para esse propósito, oferecendo inclusive um modo de divulgar as métricas geradas pelas ferramentas de análise de código mais comuns (veja figura 6).


Latest
Next
Previous

Build Report
View Build Log
NUnit Details
NUnit Timings
NAnt Output
NAnt Timings
FxCop Report
NCover Report
Simian Report
Vil Report
NDepend Full Report
NDepend Summary Report
NDepend Application Metrics Report
NDepend Assemblies Metrics Report
NDepend Assemblies Dependencies Report
NDepend Info and Warnings Report
NDepend Types Metrics Report
NDepend Types Dependencies Report
SourceMonitor Summary Report

SourceMonitor Analysis

Designed for use with SourceMonitor and CruiseControl.NET.

Summary

ID	Metric	Value
M0	Lines	4292
M1	Statements	2123
M2	Percent Comment Lines	5.9
M3	Percent Documentation Lines	4.3
M4	Classes, Interfaces, Structs	61
M5	Methods per Class	6.69
M6	Calls per Method	1.74
M7	Statements per Method	2.56
M8	Line Number of Most Complex Method	
M9	Name of Most Complex Method	Search1.Page_Load()
M10	Maximum Complexity	11
M11	Line Number of Deepest Block	
M12	Maximum Block Depth	5
M13	Average Block Depth	1.69

File LabGuide\AssemblyInfo.cs

ID	Metric	Value
M0	Lines	58
M1	Statements	14
M2	Percent Comment Lines	69.0
M3	Percent Documentation Lines	0.0
M4	Classes, Interfaces, Structs	0
M5	Methods per Class	0.00

Figura 6. Informações coletadas do SourceMonitor e disponibilizadas pelo dashboard do Cruise Control.net.

Saiba Mais

Um roteiro de como integrar o SourceMonitor ao CruiseControl.Net pode ser encontrado em:
http://www.robincurry.org/blog/IntegratingSourceMonitorCodeMetricsIntoAnNAntCruiseControlNETBuild.aspx

Outro ponto a se considerar é que a simples medição, quando dissociada de qualquer incentivo de uso, pode gerar uma "montanha" de dados que nunca são utilizados. De fato, métricas serão mais úteis se: (1) forem coletadas e analisadas pela própria equipe de desenvolvedores; (2) orientarem políticas de prevenção de forma a minimizar os efeitos da degradação; e (3) apontarem um caminho para melhorias no design.

O envolvimento da equipe de desenvolvimento na coleta e análise das métricas é importante na medida em que coloca os problemas levantados sob a responsabilidade de todos. Qualquer membro da equipe deve ter acesso a essas informações tão logo elas sejam disponibilizadas, bem como deve estar capacitado para analisá-las e, assim, discutir sobre elas.

Políticas de prevenção serão mais fáceis de serem formalizadas se houver alguma forma de monitorar mudanças repentinas que prejudiquem o design do software. Submeter código recém-criado a processos automatizados de integração e geração de releases viabilizam o monitoramento das métricas. É no momento da integração que o desenvolvedor perceberá que uma política de design é violada e isso poderá ser contornado antes que o novo código introduzido atue degradando o design. O grande desafio é que enquanto a natureza objetiva das métricas facilitará esse trabalho, a natureza subjetiva de qualidade do design o dificultará.

Uma possível estratégia

Utilize o Simian, o SourceMonitor ou outra ferramenta semelhante para procurar pelas métricas que você deseja monitorar. Integre as ferramentas a seu script de build fazendo-as gerar um output com o resultado das métricas (praticamente todas permitem a geração em formato xml). Verifique dentre os métodos e classes do sistema, aquele que apresenta as piores métricas. Fixe esses números como políticas básicas de design e faça a build ser interrompida com erro toda vez que esse número aumentar. Refatore periodicamente aquelas classes com os piores valores. Configure o processo de build para atualizar as políticas toda vez que uma refatoração for feita e esse número diminuir. Isso criará alguma proteção no seu design ao longo do tempo e permitirá que você o melhore com mais calma e tranqüilidade.

Tão importante quanto definir uma estratégia de melhoria baseada no uso de métricas é a constante atenção que a equipe dedica à evolução do design. Uma idéia interessante é monitorá-las em um quadro informativo e discutir periodicamente sobre tendências de melhoria ou de degradação. Isso resultará na vigilância necessária para que o design do software esteja em discussão todos os dias.

A figura 8 mostra uma forma interessante de monitorar métricas por meio de gráficos em ambientes informativos. Gráficos definem tendências que nos dirão se as coisas estão evoluindo ou não. Quando colocados em nossas salas de trabalho, eles vão nos lembrar diariamente das tendências que perseguimos.

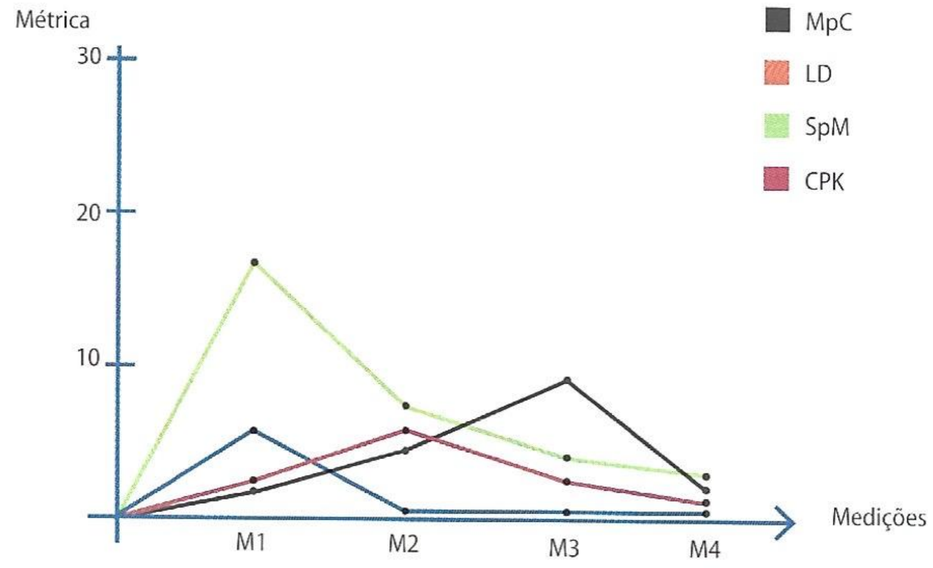


Figura 8. Acompanhando a progressão das métricas por meio de quadros informativos.

Considerações finais

Métricas de código são, enfim, um importante instrumento para a análise e compreensão do design das nossas aplicações. No entanto, elas são somente mais um instrumento que de maneira nenhuma pode substituir as habilidades dos desenvolvedores. Será o uso de suas habilidades que o tornará poderoso. É importante estabelecer estratégias mais amplas para melhoria do design, em que a análise de métricas seja parte de um entendimento maior sobre as forças que atuam sobre o código-fonte. Ela oferecerá, no entanto, novas perspectivas para tomada de ações e ajudará a estabelecer a atenção cotidiana necessária para manter a equipe focada em seus objetivos, mas sem ignorar as importantes questões associadas ao design do código-fonte.

Referências

[1] *Movimento Ágil: Descrito pelo manifesto ágil em http://www.agilemanifesto.org*

[2] *Shore, James; Warden, Shane. The Art of Agile Development, O'reilly, 2007.*

[3] *Fowler et all. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999*

[4] *Simian: http://www.redhillconsulting.com.au/products/simian/*

[5] *SourceMonitor: http://www.campwoodsw.com/sourcemonitor.html*

[6] *McConnel, Steve. Code Complete 2nd Edition, Ed, Ano.*

[7] *CruiseControl.Net: http://ccnet.thoughtworks.com/*

[8] *NAnt: http://nant.sourceforge.net*

[9] *MSBuild Reference: http://msdn2.microsoft.com/en-us/library/0k6kkbsd.aspx*

[10] *Fowler, Martin. Avoiding Repetition, IEEE Software, 2001.*

[11] *Hunt, Andrew; Thomas, Dave. Pragmatic Programmer. Addison Wesley, 1999.*

[12] *Beck, Kent. Extreme Programming Explained: Embrace Change. Addison Wesley, 1999.*