

Desenvolvimento efetivo na plataforma Microsoft

Como desenvolver e suportar software que funciona



Casa do
Código

TIME DE SUPORTE MICROSOFT MODERN APPS

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

Revisão técnica

Carlos Panato

[2016]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - ~~8º andar~~

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

COPYRIGHT

Este livro é fornecido “tal como está”, expressando exclusivamente as visões e opiniões dos autores, sendo, portanto, uma obra independente. Informações e opiniões expressas neste livro, incluindo URLs e outras referências a sites, podem mudar sem aviso prévio.

Alguns exemplos mencionados neste livro são fornecidos apenas a título demonstrativo e são de natureza fictícia, pelo que não se destinam, de todo, a representar dados nem situações reais, salvo indicação em contrário.

Microsoft e as marcas registradas listadas em <http://www.microsoft.com/trademarks> são marcas registradas do grupo de empresas Microsoft. Todas as outras marcas comerciais pertencem a seus respectivos proprietários.

PREFÁCIO

Rubiana Dalla Rosa

Scientia non habet inimicum nisi ignorantem.

Você pode imaginar como é trabalhar em um time de profissionais com alto desempenho, cujos integrantes possuem grande capacidade técnica, focados em resultados, na busca constante pela excelência na execução e com muita paixão pelo o que fazem? Pois esta é a nossa realidade. Hoje, nosso time de PFEs, além de todas estas características citadas, tem uma integração única e um companheirismo sem igual. A cultura de compartilhar o conhecimento, suas experiências e ajudar o próximo são os pilares que norteiam nosso modo de trabalhar.

Essa paixão por compartilhar conhecimento é uma prática diária, tanto para o time quanto para os clientes. Este livro é um marco para toda equipe, pois essa é a materialização da nossa realidade e daquilo em que acreditamos. A grandeza de compartilhar o que se conhece abre caminhos sem volta, pois, quando se ensina realmente, se aprende duas vezes.

Quando a ideia da criação do livro surgiu, logo identificamos que esse seria um grande desafio e algo diferente daquilo que havíamos feito enquanto time. Quando a equipe possui um objetivo em comum e sabe claramente onde chegar, o caminho até o sucesso é mais fácil. O empenho de todos foi de extrema importância e responsável por mais essa conquista.

Todo este trabalho é parte de uma nova fase que se inicia. É a prova de que tudo o que nos propusermos a fazer será feito com muita dedicação. Não há dúvidas de que faremos o máximo para

que seja executado dentro dos maiores e melhores padrões de qualidade.

Posso afirmar que esta é a melhor equipe com a qual já trabalhei e que tenho muito orgulho do que fazemos diariamente! É uma honra e um prazer trabalhar com esse time! Com muita satisfação, convido a todos a embarcar nesta obra, a desenvolver seus softwares seguindo as melhores práticas e a compartilharem seus conhecimentos, sempre.

Rubiana Dalla Rosa Gerente dos PFEs de Modern Apps da Microsoft Sponsor desta obra

Brian Keller

Speaking on behalf of the Microsoft product team which builds Team Foundation Server and Visual Studio Team Services, we take great pride in building state-of-the-art software tools which help software development teams realize their full potential and ship great software. But as with any tool – whether it's a physical tool like a hammer or a paint brush, or a digital tool like software – your ability to extract value from such a tool is directly correlated with your knowledge of how to use it. That is why I am always pleased to see books such as this one which go beyond pure product documentation to offer real-world expertise from a robust community of experts.

On a personal note, I have visited Brazil several times over the last decade while representing Microsoft and the Visual Studio family of developer tools. I have been particularly impressed with the Brazilian community of experts and their passion for sharing their knowledge with others. This book represents another step in

that journey to educate others. And while I am still struggling with my own personal mastery of Portuguese, I am pleased to see this information being shared in the native language of Brazil.

Obrigado e boa sorte!

*Brian Keller Microsoft Principal Group Program Manager
Visual Studio & ALM*

Scott Hanselman

Computer software is an art as well as a science. Some hard won bits of knowledge come only from failures and experience. There are many computer bills with just one engineer sharing their years of experience. We hope that one engineer is good at their job, and that we can learn from that one engineer's failures and successes.

So why write another book? Why should you buy this book? Because this is a collection of knowledge, pulled from not just 20 years experience of one engineer, but from over 20 engineers and all their years, all working together to bring *you* stories of their success, their favorite patterns, and their best learnings.

This is a uniquely Brazilian book, written by Brazilians, for Brazilians. These engineers are your friends, your community leaders, and your software architects. In this book they share the secrets of release management and source code tracking. How to measure code quality and when to use finalizes vs dispose. How and when to use threading, and when not to. Why are you contractors late and how can you manage that risk? Learn not just from one but from many in this powerful anthology.

Now it is time for you to get in and build your own "Casa do

Código" with this new collection.

Enjoy, friends!

Scott Hanselman Microsoft Principal Program Manager Visual Studio and .NET PM

INTRODUÇÃO

Por Alexandre Teoi

Na década de 1990, os produtos da Microsoft se consolidaram no mundo corporativo, tanto nas estações de trabalho quanto nos servidores. Assim, cada vez mais as empresas começaram a desenvolver aplicações para essa plataforma.

Com essa consolidação, a Microsoft precisou adequar o suporte oferecido para os clientes corporativos. Inicialmente, todo o atendimento de suporte era remoto, e a comunicação era feita por telefone e e-mail. Com o aumento da complexidade dos ambientes tecnológicos dos clientes e, consequentemente, das aplicações, o suporte remoto começou a ter dificuldades de resolver os chamados mais complexos.

Para endereçar esse problema, criou-se um grupo de engenheiros de suporte em campo para auxiliar na resolução desses casos mais complexos. Hoje, esses engenheiros são conhecidos por *Premier Field Engineers* (PFE).

Pela natureza do trabalho de suporte, os engenheiros de suporte são constantemente expostos aos casos mais críticos e que, potencialmente, podem causar prejuízos (financeiros ou de reputação) enormes para os clientes. Devido à criticidade dos casos, esse grupo desenvolveu várias técnicas e ferramentas para acelerar a resolução de problemas na plataforma Microsoft.

Com a evolução do grupo, esses engenheiros desenvolveram novos serviços, e começaram a fazer análises de riscos nas quais os ambientes dos clientes estão expostos, e sugerir melhorias para evitar problemas no futuro. A maior parte dos riscos analisados se origina nos casos em que os engenheiros atuam.

Toda essa experiência em resolução e prevenção de problemas permitiu que esse grupo acumulasse um vasto conhecimento do que funciona e o que pode causar problemas ao se utilizar a plataforma Microsoft. Nos capítulos seguintes, os engenheiros especializados em desenvolvimento de aplicações vão compartilhar esse conhecimento adquirido na prática, em tópicos que cobrem todas as fases do processo de desenvolvimento de aplicações.

Estrutura do livro

O primeiro capítulo deste livro contém conceitos introdutórios necessários para o entendimento dos temas discutidos. Os demais capítulos estão divididos em tópicos que descrevem problemas reais encontrados em campo.

Em alguns destes tópicos, apresentamos as narrativas de clientes sobre estes problemas. Em seguida, mostramos seus riscos, impactos e detalhes técnicos. Estes detalhes técnicos funcionam como base para os parágrafos e subseções posteriores, nos quais explanações e recomendações feitas pelos engenheiros apresentam o modo como estes problemas são resolvidos, além de detalhar cada técnica e recurso recomendado.

Público-alvo

Este livro é recomendado para desenvolvedores e arquitetos que possuam o objetivo de aperfeiçoar a qualidade e disponibilidade de seus sistemas, além de procurarem aumentar seu nível de maturidade em suas entregas.

Pré-requisitos

Como pré-requisitos para este livro, espera-se que o leitor já possua conhecimento sobre a plataforma .NET e esteja familiarizado com o funcionamento do Microsoft Visual Studio.

Este livro não tem por objetivo discutir noções básicas de desenvolvimento, por ser o reflexo de experiências retiradas de cenários avançados de clientes do time de Suporte Modern Apps Microsoft.

Sumário

1 Conceitos introdutórios	1
1.1 Tópicos base para software e sistemas operacionais	1
1.2 O motivo por trás da causa: buscando a origem da causa raiz	6
1.3 Garbage Collector	11
1.4 Suportando o IIS e entendendo o seu funcionamento	27
1.5 Conclusão	32
2 Falhas e problemas recorrentes da produção de software	33
2.1 O que acontece se seu não usar o método Dispose?	33
2.2 Padrão Dispose	38
2.3 O porquê de utilizar threads	42
2.4 Quando devo sobrescrever o método Finalize	45
2.5 Exception Shielding	56
2.6 Propagação de exceções	64
2.7 Busca em memória	68
2.8 Modelo para Serviços Windows	83
2.9 Utilizando Server Name Indications	107
2.10 Conclusão	111
3 Planejamento e gestão de demandas	112
3.1 Problemas na gestão de requisitos e suas principais causas	112

3.2 A fábrica está atrasando todas as entregas, o que posso fazer para melhorar isso?	118
3.3 Planejamento de projeto guiado a feedback	127
3.4 Como gerenciar a entrega de software por fábricas de software utilizando o TFS Git?	133
3.5 Conclusão	143
4 Padrões de desenvolvimento	144
4.1 Por que criar exceções customizadas	144
4.2 Como tratar as exceções	149
4.3 Validação de parâmetros	151
4.4 Não exponha listas em seu modelo de dados	164
4.5 Passagem de parâmetros	172
4.6 Qual a melhor estratégia de branch para o meu sistema?	178
4.7 Qualidade de código	185
4.8 Por que investir em qualidade do código?	202
4.9 Conclusão	218
5 Gestão e monitoramento de releases	220
5.1 Build e release	220
5.2 Rastreabilidade de código-fonte	232
5.3 Como extrair o máximo do Lab Management para garantir a qualidade do seu software	240
5.4 Conclusão	253
6 Boas práticas	254
6.1 Invista em revisão de código	254
6.2 Evite a codificação de métodos complexos	260
6.3 Como devo me preparar para um teste de carga?	265
6.4 Como simular a carga necessária para minha aplicação?	268
6.5 Profiling de aplicações .NET	280
6.6 Cuidados ao definir contratos de serviços	284

6.7 Bundling e minification	290
6.8 Lutando contra alterações inadvertidas no planejamento	296
6.9 Aumentando a disponibilidade e o desempenho de websites por meio de seus application pools	302
6.10 Como o Web Deploy pode ser útil?	311
6.11 Melhores práticas ao escrever expressões regulares	318
6.12 Conclusão	330
7 Bibliografia	332
8 Sobre os autores	334

CAPÍTULO 1

CONCEITOS INTRODUTÓRIOS

Desenvolver software é complexo, e requer estudo e trabalho árduos. Durante o desenvolvimento e o suporte ao software em produção, estamos propensos a diferentes problemas, sejam eles técnicos ou procedimentais. O foco deste livro é compartilhar soluções de problemas e conceitos comuns encontrados em campo pelo time de Suporte a Desenvolvimento Microsoft.

Mas, antes de iniciarmos a discussão alvo deste livro, este capítulo apresentará conceitos básicos necessários para compreensão de muitos pontos citados ao longo de todas as explanações dos próximos capítulos.

1.1 TÓPICOS BASE PARA SOFTWARE E SISTEMAS OPERACIONAIS

Por Sérgio Ramos

Diferentes conceitos são abordados diariamente quando falamos sobre software e sistemas operacionais. Sejam estes conceitos discutidos na fase de desenvolvimento, no planejamento ou durante a manutenção de sistemas. Mas, apesar destes conceitos estarem presentes na nossa rotina, dúvidas conceituais – sobre o que são e para o que servem – ainda estão vivas na cabeça de muitos desenvolvedores.

Desta forma, esta seção descreve conceitos comuns de software e de sistemas operacionais que estão presentes no nosso dia a dia e que são citados nos demais tópicos ao longo do livro. Assim, começamos a discutir a partir da unidade mais expressiva de um sistema operacional: o processo.

O *processo* é uma área de trabalho composta por um programa e seus dados. Apesar de um processo e um programa aparecerem ser similares, eles são fundamentalmente diferentes. Um programa é um conjunto de instruções a serem executadas, além das bibliotecas de comunicação com o sistema operacional. O processo, por sua vez, reúne todos os recursos necessários para a execução de um programa.

Para que os conjuntos de instruções do programa sejam executados, o sistema operacional agrupa-os em uma ou mais *threads*. As *threads* têm um simples objetivo: executar um trabalho. Thread é um bloco de execução de trabalho, que possui uma área especial de memória para rascunhos, chamada *stack*. Elas são vitais para um processo, pois sem elas nenhuma instrução de um programa seria executada.

Dentro de um processo, toda thread compartilha um conjunto de endereçamentos virtuais de espaço (originalmente conhecido como *virtual address space*), assegurando que qualquer thread tenha acesso de leitura e escrita no endereçamento virtual de memória dentro de seu respectivo processo. Procure recordar exemplos de programas passados onde tenha trabalhado com threads. Dentro desses programas era totalmente plausível compartilhar memória e recursos entre os threads, consumindo valores, objetos e referências compartilhadas entre as diferentes threads em execução de um sistema. Com o uso de threads, muito se discute sobre multiprocessamento. Multiprocessamento é a habilidade de poder executar várias tarefas em um mesmo *core* ao mesmo tempo -

assumindo-se que core é uma unidade processadora.

Um core de CPU executa apenas uma função por vez. Para permitir a execução simultânea de tarefas em um mesmo core, é criado o mecanismo de divisão do trabalho em pequenos blocos de execução que são definidos pela aplicação. Para evitar execuções contínuas muito longas, é usado um mecanismo de preempção que controla o tempo máximo de execução, sendo esse tempo chamado *quantum*.

Desta forma, o processamento simultâneo de tarefas em um mesmo core é apenas uma simulação. Visto que, na verdade, existe um revezamento tão rápido, que passa-se a impressão de execução simultânea.

Threads pertencentes a um processo não podem referenciar o espaço de endereço virtual de outro processo, por motivos de escalabilidade e segurança dos sistemas operacionais, a menos que uma área de memória seja referenciada como uma memória compartilhada, e assim liberada para o acesso de outros processos.

O sistema de memória virtual provê uma camada de acesso intermediário sob a memória física, onde estão os dados armazenados. Este sistema corresponde a um mapeamento lógico da memória física, fato que controla o acesso aos dados que estão armazenados na memória física, impedindo que um processo acesse a memória de outro processo, ou sobreponha dados do sistema operacional.

A figura a seguir descreve o mapeamento da memória virtual para a memória física, onde temos um conjunto de endereçamento de memória virtual mapeando endereços da memória física.

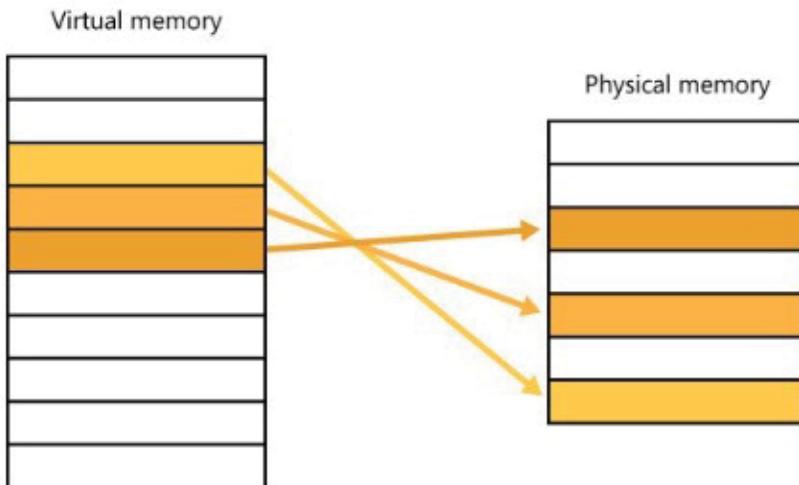


Figura 1.1: Representação do mapeamento da memória virtual sobre a memória física

O sistema de memória virtual cria para o processo a ilusão de que ele é o único dentro do sistema operacional, e que ele tem todo direito de consumir toda a memória endereçável por ele para trabalhar livremente, independentemente da memória instalada na máquina.

Processos 32 bits estão limitados a 4GB de memória virtual, quando configurada a flag de `IMAGE FILE LARGE ADDRESS AWARE`. Já processos 64 bits podem consumir de 8TB até 128TB de memória virtual, dependendo do sistema operacional, conforme a tabela a seguir.

Sistema Operacional	Limite de Memória Virtual
Windows Vista	8TB
Windows 7	8TB
Windows 8	8TB
Windows 8.1	128TB
Windows 10	128TB
Windows Server 2008	8TB

Windows Server 2008 R2	8TB
Windows Server 2012	8TB
Windows Server 2012 R2	128TB

Como a maioria das máquinas possui menos memória física do que a memória virtual oferecida pelo sistema operacional, então o sistema operacional fica responsável por transferir, ou paginar, parte da memória para o disco. Assim, ao liberar espaço na memória física, o sistema operacional permite que outros processos possam executar. E, quando necessário acesso a algum dado que foi transferido para o disco, o gerenciador de memória recupera este dado e o realoca para memória física.

Assim, com este tópico discutiu-se o significado de processos, threads, programas, memórias virtuais e outros conceitos básicos de sistemas operacionais que serão importantes para o restante dos temas discutidos no livro.

Nós acreditamos que, como desenvolvedores de software, não é suficiente que tenhamos conhecimento apenas de linguagens e comandos. É preciso ter compreensão do funcionamento dos sistemas operacionais e do impacto de nossos sistemas sobre eles.

REFERÊNCIAS

Limites de memória - [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912(v=vs.85).aspx)

Memória virtual - [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778(v=vs.85).aspx)

1.2 O MOTIVO POR TRÁS DA CAUSA: BUSCANDO A ORIGEM DA CAUSA RAIZ

Por Robson Araújo

“Tivemos um problema em produção e o corrigi, mas o mesmo tipo de problema surge recorrentemente, seja em outros sistemas, seja em módulos diferentes. O que estamos fazendo de errado? Parece que, para cada problema corrigido, dois outros surgem!”

É evidente a importância de localizar a causa raiz de um problema, uma vez que uma remediação simples evitaria que o problema voltasse a ocorrer. Em muitos casos, localizar e corrigir a causa imediata do problema não elimina a possibilidade de novas ocorrências, se não observamos o motivo do problema ter sido gerado.

Identificar a motivação de uso de um componente ou adoção de um padrão (seja ele de código, de arquitetura ou projeto) é tão importante quanto identificar a causa do problema, pois, ao educar o desenvolvedor, se impede que o problema seja repetido de forma inadvertida em novos projetos.

Além disso, ao entender e tratar a motivação, identificamos novas oportunidades de aprendizado que podem, além de evitar a disseminação de problemas, melhorar a qualidade das aplicações desenvolvidas dali em diante.

Análise de causa raiz

Análise de causa raiz (do inglês *Root Cause Analysis*, ou RCA) é um método de resolução de problemas baseado na identificação de sua origem. Um fator é considerado a *causa raiz* de um problema se sua remoção do ciclo problema-falha impede o evento indesejado de ocorrer, enquanto um *fator causal* é aquele que, se removido, pode

afetar o resultado de um evento, mas, por si só, não é a causa raiz, devendo sua causa ser também investigada.

A remoção de um fator causal pode beneficiar o resultado, mas ele, sozinho, não garante que o problema não vai voltar a se repetir.

A técnica dos Cinco Porquês

A técnica dos Cinco Porquês é uma prática adotada constantemente na resolução de problemas e identificação de causas raízes. Ao perguntar repetidamente "por que", você pode escavar as camadas de sintomas que podem levar à causa raiz de um problema.

O objetivo é fazer com que a resposta de cada pergunta, dada como razão para um problema, leve o time a uma outra pergunta, até que se chegue à origem do problema. Embora esta técnica seja chamada de Cinco Porquês, você pode achar que vai precisar fazer a pergunta menos ou mais vezes do que cinco, antes de encontrar a verdadeira razão de um problema.

Em busca da causa raiz

Para demonstrar um exemplo de uso dos Cinco Porquês, adotemos um exemplo real de uma aplicação ASP.NET desenvolvida e hospedada em um servidor web. Esta aplicação regularmente apresentava a mensagem *Service Unavailable*, e o seu *pool de aplicação* sofria com constantes paradas automáticas, sendo necessária a intervenção manual para o reinicio da aplicação.

Este problema ocorria em intervalos regulares, principalmente nos horários de alta demanda. Assim, começa-se a aplicar a técnica dos Cinco Porquês.

Primeiro Porquê: por que a aplicação está falhando?

Analisando os logs disponíveis no servidor, notou-se que a

aplicação comunicava-se com um serviço que lançava o erro `System.OutOfMemoryException`. E, por conta do número de ocorrências e repetições do mesmo erro, o pool de aplicações acabava por ser descarregado, ativando o mecanismo de *Rapid-Fail Protection*.

O serviço invocado era hospedado por uma aplicação console, que a cada requisição gerava um arquivo a partir de um componente `ReportViewer` do `Microsoft.ReportViewer.WebForms`. Este serviço apresentava um *memory leak*, onde o acúmulo de memória ao longo do seu processamento gerava o `System.OutOfMemoryException`.

Segundo Porquê: por que o serviço está apresentando um Memory Leak?

Aprofundando a investigação, identificou-se que a causa do problema de *memory leak* estava relacionada às consultas utilizadas para geração dos arquivos. Os dados e os próprios arquivos ficavam presos em memória, juntamente com um grande acúmulo de objetos dinâmicos relacionados a instâncias do `ReportViewer`.

Terceiro Porquê: por que o componente de relatórios causa o Memory Leak?

Analisando o comportamento da aplicação, foi identificado que o problema estava relacionado com a execução do método `Dispose` da classe de relatórios, pois, ao executá-lo, era gerada uma exceção. Por definição, o método `Dispose` jamais deveria causar uma exceção.

Estudando o comportamento interno do método `Dispose` da classe `ReportViewer`, o problema ficou mais claro: o método utilizava a coleção `Session` da instância corrente da classe `HttpContext`, onde ocorria a exceção. Como o serviço de geração

dos arquivos era hospedado em uma console, não existia uma instância corrente `HttpContext`, o que é o comportamento esperado em aplicações ASP.NET.

Recomendou-se a substituição do componente `ReportViewer` da biblioteca `Microsoft.ReportViewer.WebForms`, pelo componente `ReportViewer` da biblioteca `Microsoft.ReportViewer.WinForms`. Após a aplicação das recomendações, o *memory leak* foi corrigido.

Quarto Porquê: por que foi utilizada a versão web de um componente em um serviço hospedado em uma console?

Conversando com a equipe responsável pela sustentação da aplicação, foi descoberto que a biblioteca de geração de arquivos foi escrita como parte de um serviço, e que ela é reutilizada em mais de uma aplicação. Porém, a biblioteca está sendo duplicada em mais de um servidor, sendo usada em hosts diferentes, combinada com outros serviços de outras aplicações.

Na sua responsabilidade original, a biblioteca de geração de arquivos executava dentro do contexto de uma requisição Web, pois o host do serviço era uma aplicação web. É possível perceber que o problema já começa a deixar sua esfera técnica, e demonstra que se mais o processo de testes tivesse sido mais eficiente, este problema teria sido identificado antes de a aplicação entrar em produção.

Quinto Porquê: por que o host de serviço da aplicação foi desenvolvido como uma aplicação console?

Por que o serviço foi exposto em uma aplicação console, se a hospedagem de serviços no IIS é mais estável e escalável?

O sistema em questão é composto por um website e um conjunto de robôs que, combinados, efetuam transformações em resultados de processamentos em lote. A necessidade da utilização

de serviços veio do requisito de centralizar processamentos comuns a estes robôs e, por conta disso, a geração de arquivos foi levada para as consoles.

Resumindo

O exemplo deste caso mostra claramente que muitas vezes a causa de um problema é tão superficial quanto a mensagem de erro: só um sintoma do verdadeiro problema. A análise de causa raiz é uma técnica que auxilia na identificação do verdadeiro problema, e a melhor forma de corrigi-lo.

É também muito importante levantar o “fator humano” de um problema como parte da análise, tentar entender não só o problema, mas o sistema em si, onde ele se encaixa no processo da empresa, qual a participação de seus usuários, as motivações para cada modificação, e a história da aplicação por quem a desenvolveu. No fim, os sistemas, seus problemas e suas causas são frutos de interações humanas e de negócio tão complexas que toda a verdade não pode ser expressa apenas em código.

REFERÊNCIAS

Análise da causa raiz -
https://en.wikipedia.org/wiki/Root_cause_analysis

System.OutOfMemoryException -
[https://msdn.microsoft.com/library/system.outofmemoryexception\(v=vs.110\).aspx](https://msdn.microsoft.com/library/system.outofmemoryexception(v=vs.110).aspx)

Rapid-Fail Protection - mais em
<https://www.iis.net/configreference/system.applicationhost/applicationpools/add/failure> e em
[https://technet.microsoft.com/pt-br/library/cc787273\(v=ws.10\).aspx](https://technet.microsoft.com/pt-br/library/cc787273(v=ws.10).aspx)

1.3 GARBAGE COLLECTOR

Por Alexandre Teoi, Fernando H. I. Borba Ferreira e Vinícius dos Santos Martins

“Tenho muitas dúvidas sobre o Garbage Collector. Quando desenvolvia em C++, era obrigado a liberar os recursos manualmente. Com .NET, não preciso fazer isso. Mas como posso ter certeza se esses recursos são manipulados corretamente se cada aplicação funciona de maneira diferente uma da outra?”

Devido à complexidade e aos problemas gerados pelo controle manual de memória, o CLR (*Common Language Runtime*) provê como recurso o *Garbage Collector* (GC). O GC é um gerenciador automático de memória, responsável por alocar e liberar recursos automaticamente, reduzindo a possibilidade de vazamentos de memória (*memory leaks*).

Cada processo tem seu próprio espaço de endereçamento virtual. Durante a construção de aplicações usuárias – aquelas que executam no modo usuário, como websites, consoles, aplicativos e serviços –, trabalha-se apenas com a manipulação de memória virtual, nunca manipulando diretamente a memória física de um computador. O sistema operacional fica responsável por encapsular a memória física e prover os recursos para que a memória virtual trabalhe corretamente. Cada processo gerenciado contém uma instância própria do GC em execução. Instância cujo trabalho é dedicado à manipulação do espaço de endereçamento virtual de memória do processo ao qual pertence.

O funcionamento do GC é bastante complexo. Nesta seção, serão abordados temas relevantes para o entendimento do modo como o GC funciona e, assim, auxiliar na construção de sistemas mais eficientes.

Heap gerenciado

Depois que uma aplicação gerenciada é iniciada, o CLR inicia uma instância do GC. Ao ser inicializado, o GC aloca dois segmentos de memória virtual. O primeiro deles será utilizado para armazenar e gerenciar objetos ao longo da execução da aplicação. O segundo segmento, chamado de *large object heap*, é dedicado a “objetos grandes”, considerando objetos grandes aqueles que possuem mais de 85.000 bytes de tamanho (ainda mais adiante neste tópico, falaremos sobre *large object heap*).

O heap gerenciado mantém um ponteiro que indica onde o próximo objeto será alocado no heap. A figura a seguir demonstra o heap gerenciado com os objetos A, B e C alocados, além do ponteiro (aqui intitulado de `nextObject`) posicionado onde o próximo objeto deve ser alocado.



Figura 1.2: Heap gerenciado com três objetos alocados

Passos para coleta de memória

O GC avalia quais objetos no heap são usados pela aplicação. Aqueles que não são mais utilizados estão prontos para serem expurgados. Porém, o processo de identificação de objetos em uso não é trivial e, por conta disso, é dividido em diversos passos.

O primeiro deles corresponde ao congelamento de threads. Antes que o GC possa entrar em ação, os threads do processo são congelados. Assim, nada é executado além do GC durante a coleta de memória. Isso evita que os objetos tenham seus estados alterados enquanto o GC os examina.

Após o congelamento dos threads, é iniciado o que chamamos de fase de marcação. No início dessa fase, o GC considera que todos os objetos em memória devem ser expurgados. Durante a fase de marcação, o GC varre os roots da aplicação marcando os objetos que estão em uso para, então, excluir aqueles que não estão em uso.

Roots (em português, raízes) são referências a objetos em memória. Eles são estruturas de dados que contém um ponteiro de memória para um objeto de um tipo por referência. Roots podem ser: objetos globais e estáticos, variáveis locais, parâmetros por referência e registradores da CPU.

Toda aplicação gerenciada contém uma lista de roots, e é por meio dessa lista que o GC, durante a coleta, constrói um grafo contendo todos os objetos que são acessíveis pelos roots. Objetos acessíveis aos roots incluem os próprios roots, assim como outras

instâncias de objetos a que eles fazem referência (RICHTER, 2012). Objetos que estão fora deste grafo são tidos como inacessíveis, sendo assim considerados como fora de uso.

Depois de terminada a fase de marcação, o heap contém um conjunto de objetos marcados e um conjunto de objetos desmarcados. Os objetos desmarcados são expurgados, e então é iniciada a fase de compactação.

Na fase de compactação, o GC percorre o heap gerenciado procurando por blocos contínuos de objetos desmarcados. Se blocos pequenos são encontrados, eles são ignorados. Se blocos grandes são encontrados, então os objetos são deslocados para que o heap seja compactado, reduzindo a fragmentação.

Ao deslocar os objetos na memória heap, seus endereços de memória são alterados, invalidando qualquer referência a eles. Essa compactação obriga o GC a percorrer todos os objetos atualizando seus ponteiros de memória para os novos endereços dos objetos deslocados.

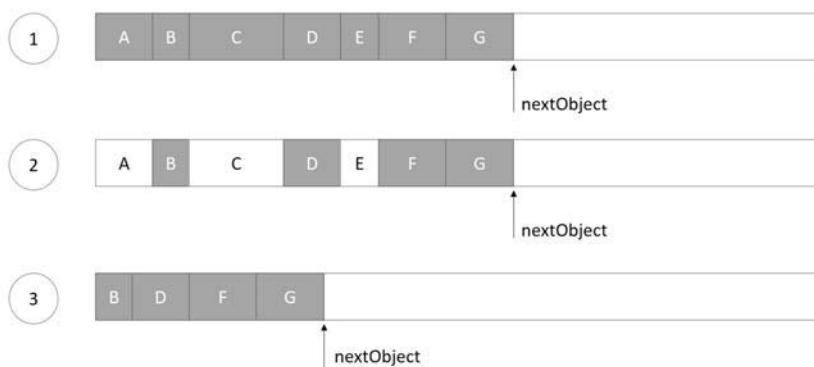


Figura 1.3: Estado do heap gerenciado durante duas fases de coleta do GC

Essa figura demonstra as alterações no heap gerenciado durante as fases de coleta do GC. Na primeira ilustração do heap, temos os

objetos alocados em memória prontos para o início da coleta. Na segunda ilustração, temos os objetos em uso marcados e os objetos que não estão em uso desmarcados. E, por fim, na última ilustração, temos o heap gerenciado compactado e com apenas objetos em uso pela aplicação ativos em memória.

Cenários de coleta e gerações

Quanto menor a quantidade de objetos no heap gerenciado, menor será o trabalho do GC. Existem situações que estimulam a coleta de memória do GC, sendo elas:

- O sistema tem pouca memória física e essa pressão de memória exige a liberação de recursos;
- A quantidade de memória alocada no heap gerenciado ultrapassa seu limite, e uma liberação de recursos é necessária para alocação de mais objetos;
- O método `System.GC.Collect` é executado e, assim, o GC é levado a coleta prematura de recursos;
- O CLR é desligado. Isso acontece quando um processo termina normalmente.

Como prática para otimização do seu trabalho, o GC do .NET Framework organiza a pilha de objetos em gerações de objetos de longa e de curta duração. Esse modelo de algoritmo de garbage collector, baseado em gerações (também conhecido como garbage collector efêmero), possui alguns pressupostos:

- Quanto mais jovem um objeto for, mais curto será seu tempo de vida;
- Quanto mais velho um objeto for, mais longo será seu tempo de vida;
- Coletar porções do heap é mais rápido do que coletar o heap inteiro.

No .NET Framework, o GC utiliza três gerações, sendo elas:

- **Geração 0:** esta é geração que mais sofre coletas de recursos, pois corresponde a geração de objetos de curta duração. Variáveis temporárias e variáveis com escopo de método são as principais candidatas a estarem na Geração 0.
- **Geração 1:** buffer intermediário de objetos entre a Geração 0 e a Geração 2.
- **Geração 2:** contém objetos de longa duração. São tidos como objetos de longa duração aqueles objetos que contém conteúdo estático (*static*), ou que são instâncias ativas durante toda a execução do processo.

Quando iniciado, o heap gerenciado não contém objetos, e todos novos objetos são adicionados na Geração 0. Objetos que estão na Geração 0 são objetos que foram recentemente criados e que nunca sofreram uma coleta do GC, como demonstrado na figura a seguir.

No exemplo desta figura, demonstra-se um conjunto de objetos sendo criados no heap gerenciado sem qualquer execução prévia da coleta de memória do GC.



Figura 1.4: Instâncias de objetos criadas na Geração 0

Se o limite de alocação da Geração 0 for atingido, o GC é acionado para executar sua primeira coleta de memória. Após remover todos os objetos que não estavam mais em uso, os objetos restantes (que são tidos como em uso pela aplicação) são promovidos para Geração 1, ficando vazia a coleção de objetos da Geração 0, conforme a figura seguinte.

A	B	D					
Geração 1							

Figura 1.5: Instâncias de objetos promovidas para Geração 1

Por padrão, durante a continuidade da execução da aplicação, novos objetos serão alocados. E estes novos objetos serão alocados na Geração 0, como apresentado na figura adiante. Nota-se que a Geração 1 mantém-se intocada, contém apenas os objetos sobreviventes da primeira coleta do GC.

A	B	D	I	J	K	L	N					
Geração 1	Geração 0											

Figura 1.6: Novas instâncias de objetos criadas na Geração 0

Quando sob pressão (ou seja, quando a cota de alocação da Geração 0 for atingida), novamente o GC iniciará sua coleta de memória. Nesse instante, o GC deve decidir quais gerações serão examinadas. Se a Geração 1 ainda estiver dentro do seu limite de consumo de memória, ela não será examinada.

Assim, apenas a Geração 0 será alvo da coleta, liberando instâncias de objetos que não são usadas e promovendo as instâncias que ainda estão em uso para a Geração 1, como demonstrado na figura:

A	B	D	K	N								
Geração 1												

Figura 1.7: Promoção de objetos da Geração 0 para Geração 1

Como esperado, a aplicação continuará a criar novas instâncias de objetos, preenchendo a Geração 0 com essas novas instâncias (figura a seguir).

A	B	D	K	N	O	P	Q	R	S	T	
Geração 1						Geração 0					

Figura 1.8: Novas instâncias de objetos adicionadas a Geração 0

Sob nova pressão de memória, o GC vai novamente decidir quais gerações serão examinadas. Caso a Geração 1 esteja acima do seu limite de memória, ela também será examinada, assim como a Geração 0. Se, durante a execução da aplicação, objetos contidos na Geração 1 deixarem de ser utilizados, então serão eliminados da memória, seguindo as mesmas regras da Geração 0.

Os objetos da Geração 1 que sobreviverem à coleta de memória serão promovidos para Geração 2, enquanto a Geração 0, ao fim de cada coleta, está livre de objetos gerenciados, conforme vemos na figura a seguir.

A	D	N	O	S	T	
Geração 2			Geração 1			

Figura 1.9: Promoção de instâncias de objetos da Geração 1 para Geração 2

Objetos contidos na Geração 2 sobreviveram a duas ou mais coletas do GC. Como mencionado anteriormente, o algoritmo de garbage collector do .NET Framework contém apenas três gerações.

O Garbage Collector é um gerenciador de memória que se autoajusta para melhor adaptar sua performance. As três gerações iniciam com limites de alocação de memória que se adaptarão ao longo de sua execução para melhor adequar o funcionamento do GC.

Por exemplo, suponha que a Geração 0 inicia com limite de memória de 256Kb. Se, durante sua execução, o GC notar que poucos objetos estão sobrevivendo a sua coleta na Geração 0, ele

pode reduzir o limite de memória dessa geração para 128Kb. Da mesma maneira, se o GC avaliar que muitos objetos sobrevivem à coleta da Geração 0, ele pode adaptar seu limite para 512Kb. Automaticamente, aumenta-se o limite de memória da geração e diminui-se a quantidade de varreduras, pois quanto maior o limite de memória de uma geração, menor será a quantidade de vezes que o GC vai examiná-la.

O GC pode utilizar essa mesma heurística para manipular os limites das Gerações 1 e 2. Os limites de tamanho de cada geração são adaptáveis a plataforma que está em execução.

O uso da estratégia de gerações no algoritmo do GC melhora inclusive a performance da fase de marcação, pois se um objeto pertencer a uma geração mais antiga, todos os objetos para os quais aquele faz referência são ignorados. Dessa forma, não há necessidade de varrer todos os objetos contidos no heap gerenciado.

Entretanto, é possível que um objeto antigo refencie um objeto mais novo. Neste cenário, o GC usa um mecanismo interno do compilador JIT (*Just-In-Time*), que marca os objetos que tiveram mudanças em seus campos. Somente esses objetos marcados são examinados.

Large Object Heap

O CLR considera um objeto como sendo, ou um objeto pequeno, ou um objeto grande. Entende-se por objetos grandes aqueles que possuem mais de 85.000 bytes ou 83Kb de tamanho. Esses objetos são tratados de forma um pouco diferente daqueles considerados pequenos:

- Objetos grandes não são armazenados dentro do mesmo espaço de endereçamento que os pequenos. São armazenados em outra área do espaço de

endereçamento do processo.

- Esses objetos não sofrem compactação, pois haveria um alto custo de performance para movê-los na memória. Por isso, a fragmentação pode acontecer na estrutura desses objetos.
- São imediatamente considerados como estando na Geração 2. Devido a isso, esses tipos de objetos devem ser criados se a intenção é mantê-los em uso por um longo período.

Objetos grandes são, por exemplo, strings grandes (como um arquivo XML) ou arrays usados em operações de I/O.

Modelos de execução do Garbage Collector

Existem dois modelos de execução do garbage collector do .NET Framework, são eles: *workstation mode* e *server mode*.

O *workstation mode* é o modelo padrão de execução do GC. Este pressupõe que a aplicação executa em uma máquina na qual outras aplicações usuárias concorrem por CPU e, por conta disso, não expõe todo seu poder de processamento. Neste modelo, o GC cria apenas um heap gerenciado para todo o consumo de memória da aplicação, além de um thread para gerenciar a alocação e limpeza de memória.

O *server mode* é o modelo que aperfeiçoa o GC para execução de aplicações em servidores. Neste modelo, é criado um heap gerenciado, além de um thread para administração da memória, para cada núcleo de processamento lógico. Assim, em um ambiente multiprocessado com, por exemplo, quatro núcleos de processamento, a aplicação terá quatro heaps gerenciados e quatro threads para gerenciar a alocação de memória. Com múltiplos

threads trabalhando juntos em paralelo, a coleta de memória é muito mais rápida se comparada à execução com *workstation mode*.

Para configurar uma aplicação para alterar seu modelo de execução para *server mode*, é preciso configurá-la conforme o exemplo a seguir:

```
<configuration>
    <runtime>
        <gcServer enabled="true"/>
    </runtime>
</configuration>
```

Utilizando uma aplicação como exemplo, podemos analisar um processo gerenciado em execução, visualizando o modo como o GC executa em *workstation mode*. Para tanto, é preciso coletar um despejo de memória (*memory dump*) e analisá-lo com a ferramenta WinDbg.

A figura a seguir apresenta detalhes de funcionamento de um processo em *workstation mode*. Note que, na listagem da figura, temos em destaque a descrição do *workstation mode* e do número de heaps igual a um.

```
0:000> !eeversion
4.6.1073.0 free
Workstation mode
SOS Version: 4.6.1073.0 retail build
0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x00000225d20f5978
generation 1 starts at 0x00000225d20f52b0
generation 2 starts at 0x00000225d20c1000
ephemeral segment allocation context: none
    segment           begin           allocated           size
00000225d20c0000 00000225d20c1000 00000225d22af328 0x1ee328(2024232)
Large object heap starts at 0x00000225e20c1000
    segment           begin           allocated           size
00000225e20c0000 00000225e20c1000 00000225e20d1900 0x10900(67840)
Total Size:           Size: 0x1fec28 (2092072) bytes.

GC Heap Size:           Size: 0x1fec28 (2092072) bytes.
```

Figura 1.10: Detalhes internos de um processo gerenciado GC com *workstation mode*

Depois de alterada a aplicação para executar como *server mode*, pode-se notar uma diferença no seu modelo de execução. A figura

seguinte apresenta a mesma coleta e análise. Nela, observa-se um processo executando em *server mode* utilizando 4 heaps. Nota-se que, para cada heap, temos referências às três gerações suportadas pelo .NET Framework, além de um *large object heap* para cada core de processamento.

```
0:000> !eeversion
4.6.1073.0 free
Server mode with 4 gc heaps
SOS Version: 4.6.1073.0 retail build
0:000> !eeheap -gc
Number of GC Heaps: 4

Heap 0 (000002136641fbc0)
generation 0 starts at 0x0000021367ef1030
generation 1 starts at 0x0000021367ef1018
generation 2 starts at 0x0000021367ef1000
ephemeral segment allocation context: none
    segment           begin           allocated           size
0000021367ef0000 0000021367ef1000 000002136808dfe8 0x19cf8(1691624)
Large object heap starts at 0x0000021767ef1000
    segment           begin           allocated           size
0000021767ef0000 0000021767ef1000 0000021767ef01900 0x10900(67840)
Heap Size:          Size: 0x1ad8e8 (1759464) bytes.

Heap 1 (0000021366448750)
generation 0 starts at 0x0000021467ef1030
generation 1 starts at 0x0000021467ef1018
generation 2 starts at 0x0000021467ef1000
ephemeral segment allocation context: none
    segment           begin           allocated           size
0000021467ef0000 0000021467ef1000 000002146808dfe8 0x19cf8(1691624)
Large object heap starts at 0x0000021777ef1000
    segment           begin           allocated           size
0000021777ef0000 0000021777ef1000 0000021777ef1018 0x18(24)
Heap Size:          Size: 0x19d000 (1691648) bytes.

Heap 2 (000002136644c940)
generation 0 starts at 0x0000021567ef1030
generation 1 starts at 0x0000021567ef1018
generation 2 starts at 0x0000021567ef1000
ephemeral segment allocation context: none
    segment           begin           allocated           size
0000021567ef0000 0000021567ef1000 0000021568085fe8 0x194fe8(1658856)
Large object heap starts at 0x0000021787ef1000
    segment           begin           allocated           size
0000021787ef0000 0000021787ef1000 0000021787ef1018 0x18(24)
Heap Size:          Size: 0x195000 (1658880) bytes.

Heap 3 (0000021366450d90)
generation 0 starts at 0x0000021667ef1030
generation 1 starts at 0x0000021667ef1018
generation 2 starts at 0x0000021667ef1000
ephemeral segment allocation context: none
    segment           begin           allocated           size
0000021667ef0000 0000021667ef1000 000002166827bfe8 0x38afe8(3715048)
Large object heap starts at 0x0000021797ef1000
    segment           begin           allocated           size
0000021797ef0000 0000021797ef1000 0000021797ef1018 0x18(24)
Heap Size:          Size: 0x38b000 (3715072) bytes.

GC Heap Size:          Size: 0x86a8e8 (8825064) bytes.
```

Figura 1.11: Detalhes internos de um processo gerenciado GC com server mode

Por padrão, aplicações ASP.NET executadas no IIS (*Internet Information Services*) executam no modo *server mode*.

Finalização

Alguns objetos requerem mais do que simplesmente memória para serem úteis encapsulando, por exemplo, recursos nativos como arquivos, conexões com o banco de dados, sockets etc. O CLR sabe lidar bem com objetos alocados no heap gerenciado, porém nada sabe sobre os recursos nativos. Consequentemente, ao ocorrer um GC, o objeto gerenciado é expurgado corretamente, enquanto o recurso nativo não é liberado, podendo causar vazamentos de memória.

Devido a essa particularidade, é necessário que a limpeza ou liberação do recurso aconteça antes que o objeto que o encapsula tenha sua memória recuperada. Isso é possível por meio de uma funcionalidade provida pelo CLR, conhecida como finalização.

A finalização ocorre em um objeto quando este chama o método `Finalize`. Este pode ser sobreescrito e é definido usando uma sintaxe especial, colocando o símbolo til (~) na frente do nome da classe, como mostra o exemplo:

Definindo o método Finalize

```
public class MinhaClasse {  
    // Esse é o método Finalize.  
    ~MinhaClasse() {  
        // Código para a limpeza de recursos.  
    }  
}
```

Ao projetar uma classe, porém, é melhor evitar usar o método `Finalize` pelas seguintes razões:

- Esse tipo de objeto leva mais tempo para ser alocado.
- Objetos finalizáveis são promovidos para gerações mais

antigas, prevenindo de serem coletados assim que o GC determina que estes devem ser expurgados. Além de que todos os objetos relacionados a eles também são promovidos de geração, forçando estes a viverem muito mais do que o necessário.

- Não há controle sobre quando o método `Finalize` será executado, e os recursos podem ser mantidos até a próxima execução do GC, aumentando, assim, a pressão na memória.
- Não há garantia quanto a ordem de execução com que os métodos `Finalize` serão chamados. Por isso, é preciso evitar que se façam referências, de dentro do método, a objetos que também implementam o `Finalize` pois esses podem já ter sido finalizados.

Ao instanciar um objeto da classe `MinhaClasse`, por ter definido o método `Finalize`, um ponteiro para esse objeto é adicionado na fila de finalização. Essa fila é uma estrutura interna usada pelo GC que mantém os endereços dos objetos que precisam de uma limpeza adicional antes de serem coletados.

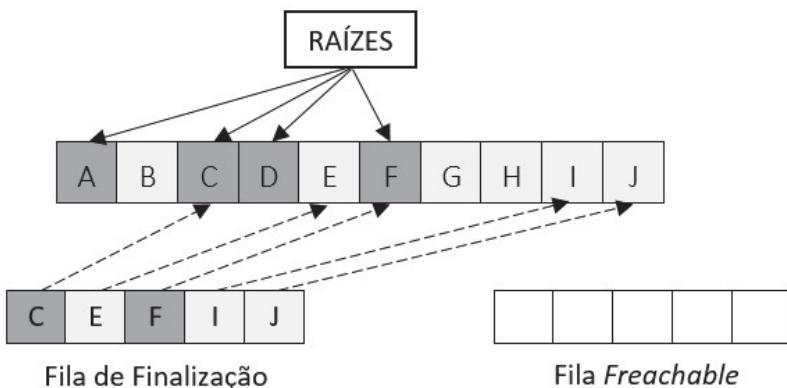


Figura 1.12: Heap Gerenciada mostrando a fila de finalização

Ao ocorrer o GC, os objetos B, E, G, H, I e J são marcados para terem suas memórias coletadas. A fila de finalização é, então, analisada em busca de referências a esses objetos. Ao encontrá-las, elas são removidas da fila e são adicionadas na fila *freachable*. Neste momento, os objetos referenciados por esta fila são trazidos de volta a “vida” (daí o nome *reachable* – alcançável – da fila), pois eles precisam ser acessíveis para que sejam finalizados. Esse fenômeno é conhecido como ressureição. Importante ressaltar que, após a fase de compactação do GC, as referências ainda contidas na fila de finalização têm seus valores atualizados caso os objetos para os quais apontam tenham sido movidos na memória.

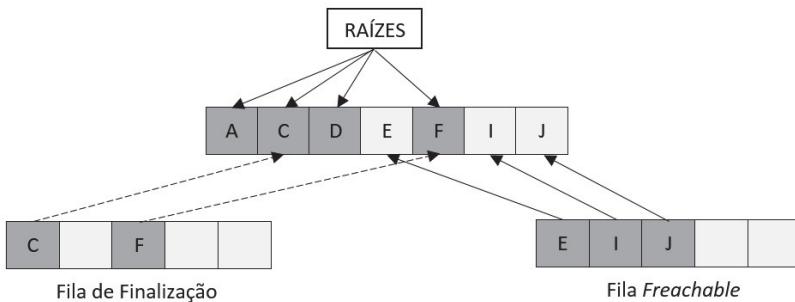


Figura 1.13: Heap gerenciada mostrando os ponteiros que foram movidos para a fila Freachable

Como é possível notar nessa figura, os objetos E, I e J sobreviveram ao GC, logo, sendo promovidos para gerações mais antigas. Após terem seus métodos `Finalize` executados, os objetos têm suas referências removidas da fila *freachable* e estão prontos para serem coletados. Assim, conclui-se que são necessárias de, no mínimo, duas execuções do GC para que suas memórias sejam recuperadas.

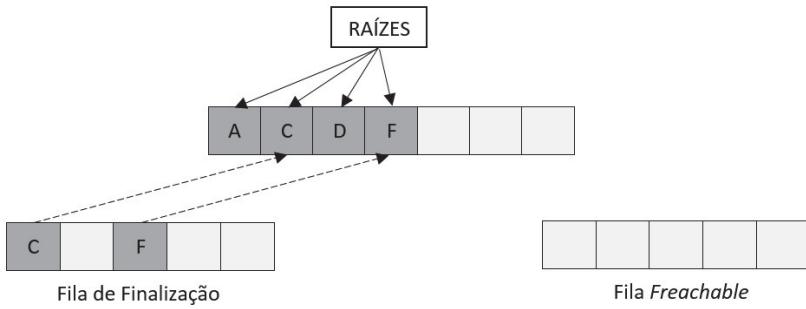


Figura 1.14: Heap gerenciado após a execução de um segundo GC

Ao analisarmos um processo gerenciado internamente, percebe-se que, dentre os threads em execução, um se destaca entre os demais: o de finalização. Esse thread é responsável por executar os métodos finalizadores dos objetos adicionados à fila de finalização.

A figura a seguir apresenta um conjunto de threads em execução de um processo gerenciado. Nota-se que um dos threads possui (ao lado direito da figura) o rótulo (`Finalizer`) , indicando que este é o thread que executa a finalização dos objetos finalizáveis.

	ID	OSID	ThreadOBJ	State	GC Mode	GC Alloc Context	Domain	Lock Count	Apt	Exception
0	1	794c	0000021316639e960	203a20	Preeemptive	0000021367EF3FD0	00000213663953e0	0	MTA	
8	2	7820	00000213367ddff110	b2b20	Preeemptive	0000000000000000	00000213663953e0	0	MTA	(Finalizer)
32	3	7810	00000213367e4e40	20420	Preeemptive	0000000000000000	00000213663953e0	0	Ukn	
XXX	4	0	00000213367eacc0	30220	Preeemptive	0000000000000000	00000213663953e0	0	Ukn	
XXX	5	0	00000213367e1e90	49820	Preeemptive	0000000000000000	0000000000000000	0	Ukn	
14	6	7a98	00000213367e68350	102920	Preeemptive	000002146800DF90	000002146800DF90	0	MTA	(Threadpool Worker)
17	7	7a99	00000213367e68350	402920	Preeemptive	0000000000000000	00000213663953e0	0	MTA	
18	8	415c	00000217b561ab0	b2b20	Preeemptive	0000021567EF4650	0000021567EF5FD0	0	MTA	
19	9	71fc	00000217b2637330	20420	Preeemptive	0000021467F790D0	0000021467F29FD0	0	MTA	
21	10	7a90	00000217b2637330	102920	Preeemptive	000002146800AAB0	000002146800BF00	0	MTA	(Threadpool Worker)
22	11	7a91	00000217b2637330	402920	Preeemptive	000002146800AAB0	000002146800BF00	0	MTA	(Threadpool Worker)
23	12	7560	00000217b2634e0	102920	Preeemptive	00000215680083138	00000215680083FD0	0	MTA	(Threadpool Worker)
24	13	7aeb	0000021367ee6e800	102920	Preeemptive	000002156800845C0	00000215680085FD0	0	MTA	(Threadpool Worker)

Figura 1.15: Threads de finalização em destaque de um processo gerenciado

Mais detalhes sobre o uso do método `Finalize` podem ser encontrados no tópico *Quando devo sobrescrever o método Finalize*, do próximo capítulo.

REFERÊNCIAS

- Fundamentals of Garbage Collection -
[https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx)
- Object.Finalize Method() - [https://msdn.microsoft.com/en-us/library/system.object.finalize\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.object.finalize(v=vs.110).aspx)
- Garbage Collector Basics and Performance Hints -
<https://msdn.microsoft.com/en-us/library/ms973837.aspx>
- RICHTER, J. *CLR via C#* - *Developer Reference*. 4. ed. Microsoft Press, 2012.

1.4 SUPORTANDO O IIS E ENTENDENDO O SEU FUNCIONAMENTO

Por Adilson Coutrin

“Somos do time de suporte a servidores web, estamos com um problema no ambiente produtivo de uma aplicação e não encontramos absolutamente nada de evidências nos logs do IIS. Por onde começamos a análise?”

Para responder essa pergunta, há necessidade de entender o funcionamento do IIS e seus componentes.

O *Internet Information Services* (IIS) é uma plataforma da Microsoft destinada a hospedagem de sites, serviços e aplicações que podem integrar diferentes tecnologias disponíveis no mercado, como: ASP.NET, ASP, WCF, Node.JS e PHP. A tabela a seguir mostra versões do IIS disponíveis até momento e seus respectivos

sistema operacional.

Sistema operacional	Versão do IIS
Windows Server 2003	IIS 6.0
Windows Server 2008	IIS 7.0
Windows Server 2008 R2	IIS 7.5
Windows Server 2012	IIS 8.0
Windows Server 2012 R2	IIS 8.5
Windows Server 2016	IIS 10

Embora aparentemente o funcionamento do IIS não seja complexo, deve-se ficar atento no momento de suportar os clientes e responder uma questão: em qual componente da arquitetura pode estar acontecendo o problema?

A figura a seguir exibe os diferentes componentes da arquitetura do IIS:

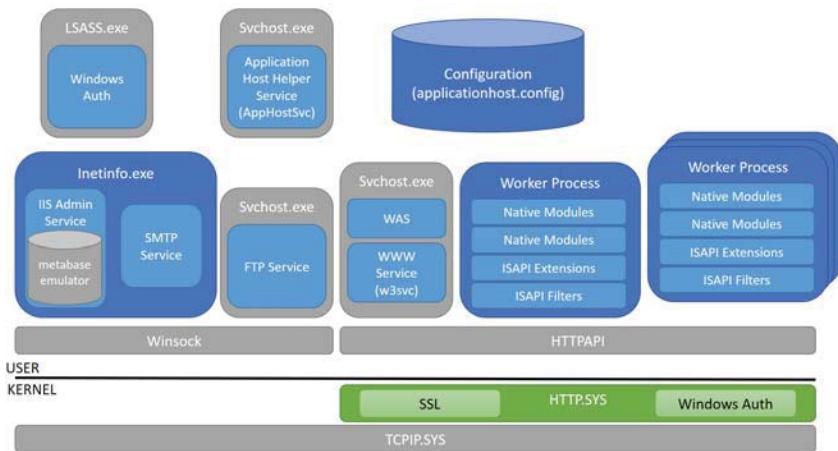


Figura 1.16: Arquitetura do Internet Information Services a partir da versão 7

Além disso, é importante saber qual é responsabilidade de cada

componente na arquitetura.

- **HTTP.sys** - É o componente do modo kernel que escuta e recebe as requisições vindas da rede (`tcp.sys`), além de hospedar as filas criadas pelo W3SVC. O driver HTTP.sys também responde por caches e grava logs de requisições do IIS, por padrão.
- **ApplicationHost.config** - Arquivo onde são armazenadas as configurações comuns do IIS, por exemplo, site e Application Pool.
- **Windows Process Activation Services (WAS) 5** - É um serviço hospedado pelo `svchost.exe` no modo usuário. Esse componente tem uma grande importância na arquitetura do IIS, pois além de ler as configurações do `applicationhost.config`, também gerencia o ciclo de vida e saúde do processo denominado *work process* (`w3wp.exe`). Ele ainda é responsável por receber as conexões que não são HTTP. A figura mostra o fluxo de acesso ao WAS:

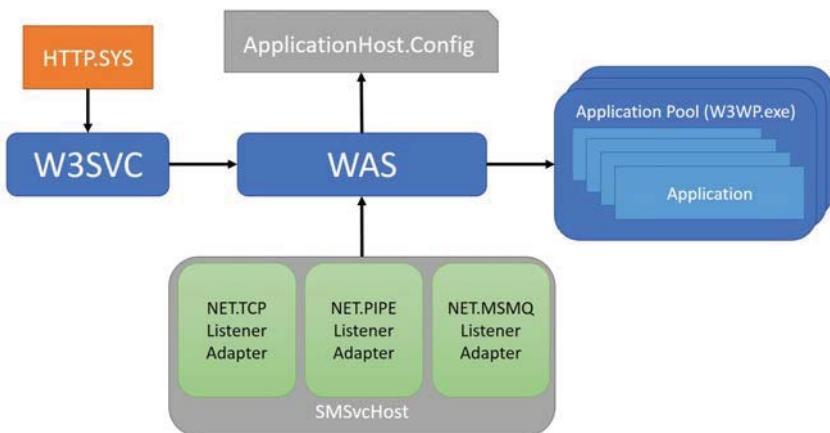


Figura 1.17: Fluxo de acesso http/https e não-http ao WAS

- **World Wide Web Publishing Service (W3SVC)** - É

um serviço hospedado pelo processo `svchost.exe` no modo usuário. Esse componente configura as filas do driver HTTP.sys, de acordo com as configurações descritas no `ApplicationHost.config`. Além disso, é responsável pelos contadores de performance.

- **Worker Process (w3wp)** - Esse processo é responsável por hospedar todos os códigos personalizados, como: ASP, ASP.NET, módulos de filtro (ISAPI), extensões etc.
- **IIS Application Pool** - Um pool de aplicativos define um grupo de um ou mais processos de trabalho (*Worker Process Service*) do IIS, visto com maior detalhe na sessão *Configurando o application pool para maximizar disponibilidade*, contida no capítulo 6. *Boas práticas*.
- **IIS Application Host Helper Service (AppHostSvc)** - É um serviço responsável por habilitar o histórico de configurações do IIS e mapear a identidade do usuário que inicia o *application pool*. Está hospedado no processo `svchost.exe`.
- **FTP Service** – Esse serviço está hospedado no processo `svchost.exe`. A partir da versão IIS 8.0 (antes no `inetinfo.exe`), permite que o servidor web forneça funções de FTP (*File Transfer Protocol*).
- **Inetinfo** - É um componente do modo de usuário que hospeda o arquivo de configuração (metabase) da versão IIS 6 e também serviços que não são web, como: serviço FTP (até versão IIS 7.0) e o serviço SMTP. O `inetinfo.exe` depende do serviço de administração do IIS para hospedar o metabase.
- **IISAdminService** - É serviço responsável pela configuração da contabilidade com versão IIS 6.
- **LSASS** - É um processo em modo usuário responsável

pela autenticação do Windows. Esse processo tem importância em cenários nos quais clientes não conseguem fazer a autenticação em modo kernel. É importante ressaltar que o driver `HTTP.SYS` é responsável pela implementação do mecanismo de autenticação integrada do Windows, a partir da versão IIS 7 e posteriores. Esta tarefa foi movida do processo `LSASS.EXE` no modo de usuário para melhorar o processo de autenticação e reduzir a sobrecarga.

- **Log e Trace** – O servidor Web (IIS) possui mecanismos para, em casos de problemas, o time de suporte conseguir rastrear e informar status da requisição, por exemplo, IIS logs e FREB.

Depois de conhecer a arquitetura do IIS e continuando com o cenário descrito no início da seção, o time de suporte em casos de problemas com aplicações web (através de conexões `http` ou `https`) deve seguir algumas dicas importantes:

- Quando verificar as logs do IIS e não encontrar nenhuma evidência sobre a requisição ou status no momento do problema, o componente a ser investigado é outro (no caso, o driver `http.sys`).
- O servidor Web (IIS) possui logs com status das requisições denominados `http status code`. Além disso, um status entre 400-600 pode ser considerado um problema.
- Como boa prática, os logs do IIS devem ser configurados com o padrão W3C, por disponibilizar um maior número de campos e informações no momento do tratamento do problema.
- O driver `http.sys` grava em log as informações com o status sobre a situação de uma requisição quando não encaminhada ao modo usuário, processo `w3wp.exe`.

Para os cenários onde as conexões não são http , é importante verificar os responsáveis por receber as conexões, por exemplo, IIS Hosting.

REFERÊNCIAS

IIS Hosting (Code Magazine) -

<http://www.codemag.com/article/0701041>

Log Files for a Web Site -

<http://www.iis.net/configreference/system.applicationhost/sites/site/logfile>

The HTTP status code in IIS 7.0, IIS 7.5, and IIS 8.0 (Microsoft) - <http://support.microsoft.com/kb/943891/en-us>

WAS Activation Architecture -

<https://msdn.microsoft.com/en-us/library/ms789006.aspx>

1.5 CONCLUSÃO

Neste capítulo, foram apresentados tópicos vitais para os demais temas discutidos neste livro. Também foi apresentada a técnica base dos engenheiros de suporte para a resolução de problemas: perguntar, ouvir, entender e mensurar a situação à sua volta, antes de recomendar qualquer solução.

No próximo capítulo, são apresentados problemas recorrentes enfrentados por times de desenvolvimento. Com base nos tópicos introdutórios discutidos anteriormente, os problemas apresentados a seguir serão compreendidos mais facilmente, e sendo evitados em futuros cenários.

CAPÍTULO 2

FALHAS E PROBLEMAS RECORRENTES DA PRODUÇÃO DE SOFTWARE

Falhas e problemas na produção de software são fatores constantes na rotina de qualquer time de desenvolvimento. O desafio para qualquer integrante de um time é atuar na resolução dessas falhas e desses problemas, e obter um resultado positivo que solucione tais itens após sua atuação.

Este capítulo descreve problemas recorrentes de software que afetam diretamente desenvolvedores, seus respectivos times e as companhias onde atuam. E, além de descrever os problemas e as situações nas quais foram encontrados, cada tópico deste capítulo vai descrever como estes problemas foram resolvidos.

O conhecimento apresentado neste capítulo se deve ao trabalho executado ao longo de anos de desenvolvimento e suporte a tecnologias em diferentes clientes, pelo time de Suporte ao Desenvolvimento Microsoft.

2.1 O QUE ACONTECE SE SEU NÃO USAR O MÉTODO DISPOSE?

Por Iury Oliveira

“Vejo que o consumo de memória de minha aplicação é muito alto, e esse problema tem interferido em outras aplicações que executam no mesmo servidor. E não sei ao certo o que fazer para resolver essa questão ou identificar onde está o problema.”

A resposta ao questionamento do título deste tópico pode ser resumida em: *“Sua aplicação poderá apresentar um padrão de consumo de recursos inadequado”*. Por mais que a oferta de recursos computacionais tenha aumentado nos últimos anos (ou seja, máquinas com mais memórias, mais processadores), ainda existem limitações, isto é: recursos computacionais ainda são finitos.

Ao criar um programa, este, de uma forma ou de outra, consumirá algum tipo de recurso (isto é, arquivos, conexões com banco de dados, conexões de redes). Ao tentar fazer uso deste recurso, o primeiro passo é alocar em memória a sua representação. Um exemplo seria o código a seguir:

```
FileStream fs = new FileStream(@"C:\Temp\myFile.txt", FileMode.OpenOrCreate);
```

É possível observar no trecho de código anterior o uso de um tipo chamado `FileStream` que vai realizar a leitura do arquivo `myFile.txt`, e tem sua representação criada em memória usando o método `new`. Alguns tipos no .NET Framework representam apenas um invólucro para recursos nativos, como exemplo, o `FileStream`, utilizado para abrir um arquivo do sistema operacional.

Após alocar este recurso em memória e as condições necessárias para sua utilização forem satisfeitas, poderão ser executadas as operações desejadas, por exemplo, a leitura do arquivo. Após realizada a operação sobre o `FileStream`, os recursos deverão ser liberados, pois não estão mais em uso; do contrário, teremos um conjunto de recursos acumulados sendo consumidos, mesmo que não exista nenhuma operação sendo executada sobre eles, podendo

degradar as condições do ambiente.

Mas, neste ponto, temos uma questão: “*O .NET Framework não possui componentes responsáveis em gerenciar a memória automaticamente?*”. O .NET Framework possui recursos para gerenciar memória automaticamente. Porém, memória gerenciada é apenas um dos muitos tipos de recursos existentes. Objetos de tipos nativos precisam ser liberados explicitamente e são referenciados como recursos não gerenciados.

O .NET Framework provê a interface `System.IDisposable`, que deve ser utilizada para liberação de recursos manualmente, tão cedo quanto possível. Ao implementar a interface `System.IDisposable`, devemos implementar o método `Dispose`. Este método terá dois objetivos, sendo eles:

- Propagar a execução do método `Dispose` de outros objetos que implementem essa interface e que sejam atributos ou propriedades da classe. Por exemplo, se um objeto A possui propriedades e/ou atributos de um objeto B, e este objeto B implementa a interface `System.IDisposable`, então é preciso que o método `Dispose` da classe A execute os métodos `Dispose` das instâncias do objeto B.
- Forçar a liberação de recursos não gerenciados.

O Garbage Collector (GC) não foi construído para controlar recursos não gerenciados. Desta forma, fica sob responsabilidade dos desenvolvedores liberar recursos não gerenciados. A classe `System.Object` declara um método virtual chamado `Finalize` (também conhecido como finalizador, ou destrutor), que é um método que deve ser usado exclusivamente para liberação de recursos não gerenciados.

Para um melhor entendimento sobre o uso do método

`Dispose` e a necessidade de liberação de recursos que não estão mais em uso, é preciso entender o mecanismo de alocação e uso de recursos feito pelo .NET durante a execução de um programa. Quando se trata de alocação de memória, fala-se sobre o operador `new`. Para entender o que esse operador faz, é preciso analisar a linguagem intermediária gerada, detalhada na figura seguinte.

```
IL_0000: nop
IL_0001: ldstr "C:\\Temp\\myFile.txt"
IL_0006: ldc.i4.4
IL_0007: newobj instance void [mscorlib]System.IO.FileStream::ctor(string, valuetype [mscorlib]System.IO.FileMode)
IL_000c: stloc.0
IL_000d: ldc.i4.0
IL_000e: callvirt instance void [mscorlib]System.IO.Stream::Dispose()
IL_0013: nop
IL_0014: ret
} // end of method Program::Main
```

Figura 2.1: Linguagem intermediária gerada para o operador `new`

Esse trecho refere-se ao código gerado em linguagem intermediária para a alocação do `FileStream` anterior. A instrução `newobj` vai realizar a criação de uma nova instância da classe inicializando seus campos com os valores padrões.

De modo geral, o processo de alocação de um novo item em .NET é algo simples e rápido, pois se trata de movimentação de ponteiros para um novo endereço de memória em uma estrutura que é um bloco contínuo de memória.

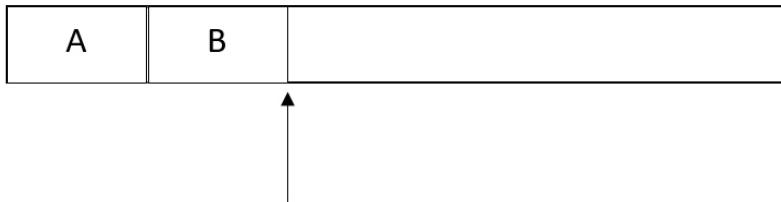


Figura 2.2: Ponteiro para o próximo objeto - `NextObjPtr`

Quando um novo objeto é criado utilizando o operador `new`, este terá como retorno a referência ao objeto recém-alocado, neste

caso B, e o ponteiro `NextObjPtr` aponta para o endereço onde o próximo objeto será colocado na `heap`.

Alocar um objeto em memória é uma operação extremamente otimizada em .NET. A maior parte dos problemas relacionados às aplicações desenvolvidas em .NET é o consumo inadequado de recursos, que muitas vezes está relacionado à não liberação destes.

A Common Language Runtime (CLR) fornece suporte para o gerenciamento automático de memória. Isto é, toda memória alocada usando o operador `new` não precisa ser explicitamente liberada.

Ao falar de liberação de recursos, temos sempre em mente o Garbage Collector. O GC é executado em alguns cenários, sendo eles:

- Quando não existe espaço suficiente para alocação de novos objetos.
- Quando explicitamente o GC é invocado por meio de `GC.Collect()`.
- Quando o Windows reporta que está com baixos níveis de memória.
- Quando um processo ou um *application domain* terminam sua execução.

REFERÊNCIAS

`System.IDisposable` - [https://msdn.microsoft.com/en-us/library/system.idisposable.dispose\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.idisposable.dispose(v=vs.110).aspx)

`Dispose` Pattern - [https://msdn.microsoft.com/en-us/library/b1yfkh5e\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.110).aspx)

2.2 PADRÃO DISPOSE

Por Iury Oliveira

Durante sua execução, aplicações desenvolvidas em .NET tendem a utilizar uma série de recursos, bem como qualquer outro framework ou linguagem o faria. A *Common Language Runtime* e o GC são os responsáveis em gerenciar recursos relacionados ao consumo de memória.

O .NET Framework possui dois tipos de finalização de objetos: determinístico e não determinístico. Objetos que necessitem de finalização determinística devem implementar a interface `IDisposable`.

O Padrão Dispose tem como principal objetivo a padronização de criação de métodos finalizadores (destrutores) e o uso da interface `System.IDisposable`. A motivação de uso deste padrão é reduzir a complexidade de implementação de finalizadores e do método `Dispose`, dados as diferentes situações como pode ser implementado.

A estrutura mais simples de implementação chama-se Padrão Básico de `Dispose` (*Basic Dispose Pattern*). Este modelo de implementação deve ser utilizando quando:

- A classe implementada contém propriedades ou atributos que sejam tipos de dados que implementem a interface `System.IDisposable` (também conhecidos como *disposable types*). Se um tipo de dados é responsável pelo ciclo de vida destes tipos de objetos, então é preciso criar meios de assegurar a liberação de seus recursos.
- A classe possui recursos que precisam ser liberados explicitamente e que não possuem finalizadores. Este

cenário também exige a implementação de um finalizador, para garantir a liberação de recursos quando o método `Dispose` não por executado.

- A classe construída não faz referência direta a recursos não gerenciados ou tipos de dados que implementem a interface `System.IDisposable`, mas possui subtipos que os fazem. O Padrão Básico de `Dispose` corresponde a implementação da interface `System.IDisposable` e da declaração de um método `Dispose(bool)`, que concentra toda a liberação de recursos e que pode ser executado por um possível finalizador.

O código a seguir apresenta a estrutura do Padrão Básico de `Dispose`, onde temos uma classe que implementa a interface `System.IDisposable` e um método `Dispose(bool)` que faz a liberação de recursos gerenciados. O parâmetro booleano do método `Dispose(bool)` deve indicar quando a chamada originou-se do método `Dispose`. Assim, limitamos o acesso a recursos que devem ser eliminados apenas na execução do método `Dispose`. Caso, o método `Dispose(bool)` seja executado a partir do método finalizador, então o valor passado como parâmetro deve ser falso.

```
public class TipoDados : IDisposable {  
  
    private TipoDisposable meuRecurso;  
  
    public TipoDados(){  
        this.meuRecurso = ... // aloca o recurso em memória  
    }  
  
    public void Dispose(){  
        Dispose(true);  
    }  
  
    protected virtual void Dispose(bool disposing){  
        if (disposing){  
            if (meuRecurso!= null)  
                meuRecurso.Dispose();  
        }  
    }  
}
```

```
    }  
}
```

Note que, nesse exemplo, não temos nenhum tipo não gerenciado ou um finalizador. Tipos finalizadores (também conhecidos como *finalizable types*) são tipos de dados que estendem o Padrão Básico de Dispose e implementam finalizadores.

É importante realçar que o uso de finalizadores é bastante específico, e que seu uso incorreto pode causar problemas de performance e elevação desnecessária da complexidade do código.

Finalizadores devem ser usados exclusivamente para a liberação de recursos não gerenciados utilizados pelo tipo de dados criados. Se o tipo de dados não faz referência a recursos não gerenciados, então não é preciso que seja implementado um finalizador para esse tipo de dados.

Se um tipo de dados possui um finalizador, o Garbage Collector o adiciona em uma fila de liberação de recursos exclusiva para tipos de dados com finalizadores, chamada Fila de Finalização (*finalization queue*). Essa fila possui a responsabilidade de assegurar que todos os métodos finalizadores, dos itens ali contidos, sejam executados. Caso um destes itens demore mais tempo para ser executado ou trave sua execução, todo o processo de liberação de recursos da aplicação será impactado.

O exemplo a seguir apresenta o Padrão Básico de Dispose implementado em uma classe que possui referência a tipos gerenciados e não gerenciados que exigem a liberação de recursos. Note neste exemplo que os empregos do método `Dispose(bool)` são executados tanto pelo método `Dispose` quanto pelo método finalizador.

```
public class TipoDados: IDisposable {  
  
    private IntPtr buffer; // tipo não gerenciado
```

```

private TipoDisposable meuRecurso;

public TipoDados(){
    this.buffer = ... // aloca o recurso em memória
    this.meuRecurso = ... // aloca o recurso em memória
}

protected virtual void Dispose(bool disposing){
    ReleaseBuffer(buffer); // libera memória não gerenciada
    if (disposing){
        if (meuRecurso != null)
            meuRecurso.Dispose();
    }
}

~ TipoDados(){
    Dispose(false);
}

public void Dispose(){
    Dispose(true);
    GC.SuppressFinalize(this);
}

private void ReleaseBuffer(IntPtr pointer){
    Marshal.Release(pointer);
}
}

```

Apesar de o método `Dispose` também ser usado para execução das rotinas de liberação de recursos não gerenciados, recomenda-se a implementação de um método finalizador para liberação de recursos não gerenciados quando o método `Dispose` não for executado. Entretanto, para evitar a dupla execução das rotinas de liberação de recursos não gerenciados (caso o método `Dispose` já fora executado), usa-se o método `GC.SuppressFinalize` para indicar ao GC que o trabalho de liberação de recursos já foi feito.

O Microsoft Visual Studio possui um recurso de análise estática de código denominado Code Analysis, que nos ajudar a evitar erros comuns de programação, como por exemplo, quando abrimos uma conexão com um banco de dados e não a fechamos, isto é, quando alocamos um recurso não gerenciado.

Em síntese, apesar de toda a facilidade provida pela CLR, bem como pelo GC, existem recursos que devem ser devidamente liberados pelo programador, visando não comprometer a estabilidade e performance de nossas aplicações. O Microsoft Visual Studio ajuda a endereçar algumas atividades de revisão de código, prestando grande auxílio na avaliação dos *assemblies* que compõe a aplicação. Entretanto, a melhor forma de garantir a qualidade final é entender como o .NET Framework funciona internamente e como podemos usar bons padrões de codificação, neste caso, o padrão *Dispose*.

REFERÊNCIAS

Finalizadores - <https://msdn.microsoft.com/en-us/library/system.object.finalize%28v=vs.110%29.aspx>

IDisposable.Dispose Method - [https://msdn.microsoft.com/en-us/library/system.idisposable.dispose\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.idisposable.dispose(v=vs.110).aspx)

2.3 O PORQUÊ DE UTILIZAR THREADS

Por Rafael Teixeira

“Quero melhorar a performance de meu código fazendo a paralelização de tarefas com threads. Qual a melhor forma para se trabalhar com threads no .NET? Utilizando o System.Threading.Thread ou o ThreadPool ?”

As *threads* são unidades mínimas de processamento usadas pelo sistema operacional para executar o código de programas e sistemas pelo processador. Ela existe dentro de um processo, isso quer dizer que um processo pode ter uma ou mais threads. Um processo sem

threads não executaria código algum.

Elas também permitem que códigos sejam executados em paralelo e, com isso, agilizem tarefas de um programa ou Sistema Operacional. Por isso, é possível escutarmos música enquanto falamos no chat e ainda navegamos na internet em nossos computadores.

Com a chegada dos processadores com mais de um núcleo, a execução das threads em paralelo ficou muito mais eficiente. Uma das grandes vantagens em utilizá-las no desenvolvimento de sistemas é a paralelização de tarefas no seu código, gerando o ganho de tempo na execução.

Os conceitos citados até agora são independentes da tecnologia usada no desenvolvimento de programas e sistemas, como por exemplo, C/C++, .NET (C#, VB.NET), Java etc.

A partir de agora, vamos entender as duas principais formas de trabalhar com threads no .NET Framework, suas facilidades, vantagens e desvantagens. Basicamente, o .NET Framework oferece duas maneiras de utilizar threads. Uma é através da classe `System.Threading.Thread`, e a outra através do .NET `ThreadPool`.

Vamos ao primeiro exemplo utilizando a classe `System.Threading.Thread`.

```
static void Main(string[] args)
{
    System.Threading.Thread threadAux = new System.Threading.Thread(ExecutarThreadAux);
    threadAux.Start();
}

static void ExecutarThreadAux()
{
    // Código executado pela threadAux
}
```

Quando executamos esse código, uma nova thread é criada, o método `ExecutarThreadAux()` é executado e, posteriormente, a thread é finalizada e retirada da memória. A cada execução deste código, os passos são repetidos.

Porém, como a criação de threads é considerada um processo custoso (por envolver chamadas ao sistema operacional), este método é indicado somente para quando o código a ser executado pela thread tiver uma duração longa e não necessitar recriar a thread a todo momento.

Para os casos de execução de código mais curtos, o .NET Framework disponibiliza o `ThreadPool`. Durante a inicialização de qualquer processo desenvolvido em .NET, um pool de threads é criado e usado pelo próprio .NET para diversas execuções do framework. Este pool também está disponível e pode ser utilizado por qualquer desenvolvedor na plataforma .NET, basta usar o objeto estático `ThreadPool`.

Vamos a outro exemplo de utilização de threads no .NET, mas agora utilizando o .NET `ThreadPool`.

```
static void Main(string[] args)
{
    ThreadPool.QueueUserWorkItem(new WaitCallback(ExecutarThreadAux));
}

static void ExecutarThreadAux(object state)
{
    // Código executado pela threadAux
}
```

Executando o código desse exemplo, o método `ExecutarThreadAux()` será colocado em uma fila e será executado pela próxima thread disponível no pool. Neste ponto, você pode estar pensando que colocar o método a ser executado em uma fila faça-o demorar mais a ser executado comparado a criação de thread

pelo `System.Threading.Thread`. Porém isso não é verdade, pois o `.NET ThreadPool` foi criado pensando na reutilização de threads. Assim, elas são criadas, utilizadas e, em vez de serem retiradas da memória quando acabam de executar, voltam ao `ThreadPool`, ficando disponíveis para um novo trabalho e evitando o custo futuro de criação de uma nova thread.

Além disso, o `ThreadPool` cria e executa as threads de uma forma controlada, levando em consideração a capacidade de hardware do computador (nº de processadores). Um exemplo simples de controle é quando o computador está com um consumo maior que 80% de CPU, e for solicitada a execução de um código pelo `ThreadPool`, conforme o exemplo no anterior. Em vez de disparar mais uma thread para concorrer por CPU e piorar o consumo de recursos da máquina, ele vai aguardar até o CPU cair abaixo de 80% para disparar os trabalhos na fila para as threads.

SYSTEM.THREADING.THREAD VERSUS THREADPOOL

O `.NET ThreadPool` é recomendado na maioria dos casos de utilização de threads no .NET, por fazer diversos controles e garantir a performance da máquina onde o código é executado. Porém, há situações onde o desenvolvedor precisa de um maior controle sobre a execução da thread e seu tempo de vida. Para esses casos de maior flexibilidade e controle, temos o `System.Threading.Thread`.

2.4 QUANDO DEVO SOBRESCREVER O MÉTODO FINALIZE

Por Felipe Fujiy Pessoto

“Não tenho certeza de quando devo utilizar o Finalize e se o faço da forma correta. Quais as melhores práticas?”

O Finalizer é um recurso muitas vezes mal compreendido e utilizado de forma incorreta, trazendo problemas de performance desnecessários. Seu uso é destinado a cenários específicos, e a correta implementação depende de cuidados especiais.

O Finalize é um método virtual declarado no System.Object . Caso este método seja sobreescrito, ele será chamado depois que um objeto é classificado como lixo e antes de sua memória ser liberada. Seu objetivo é liberar recursos não gerenciados como *handles* de arquivos, conexões de rede, entre outros, atuando como uma garantia caso o método Dispose não seja chamado.

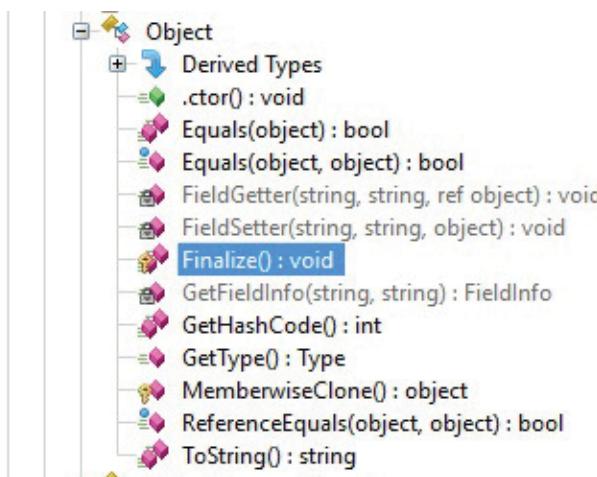


Figura 2.3: System.Object no ILSpy

No C#, não é possível sobreescriver este método da forma tradicional ou chamá-lo diretamente, uma vez que o método será chamado automaticamente. Para sobreescriver este método, deve-se utilizar a sintaxe de Destructor.

É importante nunca confundir o Finalizer do C# com o Destructor do C++, pois eles funcionam de forma diferente, sendo a mais notável o fato de os Desctructors C++ serem determinísticos. Esta sintaxe é o que se costuma chamar de *syntactic sugar*, de fato sobrescrevendo o método `Finalize` e garantindo que seu o `Finalize` na classe base seja chamado. Assim uma classe que declare um destructor:

```
public class Animal
{
    ~Animal()
    {

    }
}
```

Resulta em código IL semelhante a este:

```
.method family hidebysig virtual
    instance void Finalize () cil managed
{
    .override method instance void [mscorlib]System.Object::Finali
ze()

    .maxstack 1

    .try
    {
        IL_0000: leave.s IL_0009
    }
    finally
    {
        IL_0002: ldarg.0
        IL_0003: call instance void [mscorlib]System.Object::Final
ize()
        IL_0008: endfinally
    }

    IL_0009: ret
```

Sobrescrevendo o método `Finalize` e envolvendo o corpo do método em um `try/finally` onde o `Finalize` base é chamado no `finally`, garantindo sua execução.

Quando utilizar? E quando não utilizar?

O Finalizer tem um propósito específico: limpar recursos não gerenciados e, na grande maioria das classes, dificilmente será necessário.

Mesmo ao lidar com objetos como um `FileStream` ou `ConnectionString` que utilizam recursos não gerenciados, a responsabilidade de fazer a limpeza apropriada é dessas classes, que têm seu próprio Finalizer. Neste caso, você deve apenas se preocupar em implementar o `IDisposable`.

Com o .NET 2.0 e a classe `SafeHandler`, tornaram-se ainda mais raros os casos em que é necessário implementar seu próprio Finalizer, como discutiremos no tópico *SafeHandle*.

Não utilize o Finalizer como um padrão. É comum ver padrões internos que instruem seus desenvolvedores a sempre implementar o Finalizer ou mesmo `IDisposable`, em todas suas classes. Entretanto, isso gera um custo de performance e manutenção desnecessários para sua aplicação. O Finalizer deve ser tratado como uma exceção, apenas para casos excepcionais.

Utilizar um Finalizer vazio não trará nenhum benefício e fará com que uma referência ao objeto seja adicionada à *finalization queue*, mantendo o objeto por mais tempo em memória, pois não será possível removê-lo assim que o Garbage Collector detectar que ele não é mais referenciado.

Sempre que utilizar o `Finalize` em um tipo, garanta que ele também seja `IDisposable`, utilizando o *Dispose Pattern*, assim a limpeza pode ser feita de forma determinística e removendo a referência do objeto da *finalization queue*.

O que não fazer no Finalizer

O Finalizer é particularmente difícil de se escrever corretamente devido a algumas suposições em que normalmente podemos confiar e não são válidas nele. É comum se dizer que não se deve manipular outros objetos no `Finalize`, pois eles podem ter sido coletados. Mas este não é o real motivo, uma vez que os objetos referenciados por ele também são mantidos em memória quando o GC realiza a coleta.

Por sua natureza não determinística, não é possível saber em qual ordem o método `Finalize` será chamado entre os objetos, assim um objeto referenciado em sua classe pode já ter sido finalizado. O motivo de não se usar outros objetos no `Finalize` é por não ser possível garantir que ele já foi finalizado, por exemplo, um objeto privado da sua classe que não é “finalizable” pode ser usado com segurança.

Objetos em variáveis estáticas também estão sujeitos a ser finalizados durante o encerramento do processo ou do `AppDomain`. Por isso, se for necessário referenciar estas variáveis, tenha certeza antes de validar o `Environment.HasShutdownStarted` e o `AppDomain.IsFinalizingForUnload()`.

Se uma exceção for lançada no Finalizer de um objeto, o processo será terminado evitando que outros Finalizers sejam executados. Por exemplo, no código a seguir, em algumas vezes o Finalizer do `Animal` não será executado, caso o `FinalizeException` execute seu Finalizer antes, lembrando de que a ordem é indeterminada.

```
class Program
{
    static void Main(string[] args)
    {
        var pet = new Animal();
        var badFinalizer = new FinalizeException();

#if DEBUG
```

```

        badFinalizer = null;
        pet = null;
#endif
        GC.Collect(GC.MaxGeneration);

        Console.WriteLine("GC Collect chamado");

        Console.ReadLine();
    }
}

public class FinalizeException
{
    public object MyPrivateObject;

    ~FinalizeException()
    {
        Console.WriteLine("FinalizeException Finalize");
        MyPrivateObject.ToString();
    }
}

public class Animal
{
    ~Animal()
    {
        Console.WriteLine("Animal Finalize");
    }
}

```

Na implementação atual do .NET, existe apenas uma thread responsável pela finalização dos objetos (isto pode mudar no futuro, resolvendo alguns problemas e trazendo novos), ela remove o objeto da fila, executa seu método `Finalize` e passa para o seguinte. Portanto, é um recurso limitado, que com o crescente número de cores nos processadores e pelo uso de cada vez mais threads, essa conta fica mais desbalanceada. Assim, podemos ter um cenário com 15 threads em 15 cores criando objetos finalizáveis, e somente uma executando a finalização.

Um cenário ainda pior é por um Finalizer mal implementado e esta thread ser bloqueada indefinidamente, resultando em um vazamento de recursos. O exemplo adiante facilmente faz com que o

processo utilize quase 4GB de memória. Outros recursos como arquivos ou conexões também poderiam ser indefinidamente bloqueados.

```
class Program
{
    static void Main()
    {
        for (int i = 0; i < 1000000; i++)
        {
            new BlockingFinalizer();
        }

        Console.WriteLine("Fim");
        Console.ReadLine();
    }
}

public class BlockingFinalizer
{
    int[] values = new int[1000];

    ~BlockingFinalizer()
    {
        Console.WriteLine("Inicio BlockingFinalizer");
        Thread.Sleep(TimeSpan.FromDays(1));
        Console.WriteLine("Fim BlockingFinalizer");
    }
}
```

Outra particularidade do método `Finalize` é que ele pode ser chamado mesmo para um objeto parcialmente construído. Isto pode acontecer em alguns cenários em que uma exceção é lançada durante sua construção. Por isso, é importante que seu método `Finalize` permita este tipo de cenário, pois, mesmo que seu construtor não lance exceções, ele pode ser interrompido por um `AppDomainUnloadedException`, por exemplo.

```
class Program
{
    static void Main()
    {
        try
        {
```

```

        new ParcialmenteConstruido();
    }
    catch(Exception ex) { }

    GC.Collect(GC.MaxGeneration, GCCollectionMode.Forced, true
);

    Console.WriteLine("Fim");
    Console.ReadLine();
}
}

public class ParcialmenteConstruido
{
    private object resource;
    public ParcialmenteConstruido()
    {
        File.Open(@"c:\naoexiste.txt", FileMode.Open);
        resource = new object();
    }

    ~ParcialmenteConstruido()
    {
        resource.ToString(); //Erro
        //Limpa outros recursos não gerenciados
    }
}

```

Custos

Além da dificuldade de implementar um Finalizer corretamente, ele também tem custos para a aplicação. Sempre que um objeto com Finalizer é instanciado, uma referência precisa ser adicionada ao *finalization queue*. Este custo será sentido principalmente em objetos pequenos com muitas instâncias sendo criadas.

Um impacto ainda maior é devido ao fato de estes objetos não serem removidos da memória no momento em que o Garbage Collector detecta que ele não é mais utilizado, pois o objeto ainda precisa ser finalizado. Assim o objeto e toda sua cadeia de referências são mantidos em memória e promovidos para a próxima geração, se esta cadeia contiver muitos objetos em memória, poderá

se sentir impacto no uso de memória.

SafeHandle

Escrever um Finalizer é realmente difícil, por isso a partir do .NET 2 foi introduzida a classe `SafeHandle`, que encapsula um handle do sistema operacional.

A classe `SafeHandle` conta com diversas vantagens:

- Evita a corrupção de Handles por concorrência entre a thread da aplicação e a thread do Finalizer;
- Evita o vazamento de Handles devido a exceções como `ThreadAbortException`;
- Evita ataques de *Handle-Recycling*;
- Desencoraja a criação de grandes cadeias de referências no objeto `Finalizable`;
- Implementa o `CriticalFinalizerObject`, trazendo seus benefícios.

O exemplo a seguir demonstra como um objeto pode ser coletado e finalizado mesmo durante a execução de um método de instância.

```
class Program
{
    static void Main()
    {
        var teste = new TesteAlcanceGc();
        teste.Metodo();

        Console.ReadLine();
    }
}

public class TesteAlcanceGc
{
    IntPtr handle;

    public void Metodo()
```

```

{
    WeakReference refe = new WeakReference(this);
    UtilizaHandle(handle); //Última referência ao objeto é o this da chamada this.handle
    //Mesmo durante a execução de um método de instância o objeto já pode ser coletado.

    Console.WriteLine("this is alive: " + refe.IsAlive);

    //GC.KeepAlive(this);
}

public void UtilizaHandle(IntPtr handle)
{
    //Forcamos o GC para
    GC.Collect(GC.MaxGeneration, GCCollectionMode.Forced, true
);
    GC.WaitForPendingFinalizers();

    //Mesmo utilizando o Handle aqui, ele já foi fechado pelo
Finalize
    handle.ToString();
}

~TesteAlcanceGc()
{
    //Fecha o Handle
    Console.WriteLine("TesteAlcanceGc Finalize");
}
}

```

A `WeakReference` demonstra que o objeto já foi coletado e a mensagem do Finalizer é exibida antes de o método terminar. Para evitar este problema, é necessário o `GC.KeepAlive` no final do método.

Escrever uma classe que herda de `SafeHandle` também não é uma tarefa fácil. Por isso, o .NET traz um conjunto de classes que devem ser utilizadas:

- `SafeFileHandle`
 - `SafeMemoryMappedFileHandle`
 - `SafeMemoryMappedViewHandle`
 - `SafeNCryptHandle` , `SafeNCryptKeyHandle` ,
-

```
SafeNCryptProviderHandle          e
SafeNCryptSecretHandle
• SafeProcessHandle
• SafeRegistryHandle
• SafeWaitHandle
• SafeX509ChainHandle
```

Utilizando estas classes, não é necessário ter um Finalizer para limpar estes recursos, basta chamar o método `Dispose` delas ao implementar o `IDisposable`.

CriticalFinalizerObject

A classe `SafeHandle` e qualquer outra que herde de `CriticalFinalizerObject` tem seu Finalizer marcado como “critical”, executando em uma *Constrained Execution Region* (CER). Isso significa que todo o código do Finalizer terá a oportunidade de ser executado, prevenindo exceções assíncronas, por exemplo, que seja lançada uma `ThreadAbortException`.

Para isso, algumas regras devem ser seguidas. Seguem algumas operações não permitidas:

- Alocações;
- Adquirir lock;
- Boxing;
- Acesso a arrays multidimensionais;
- Chamar método por Reflection;
- Serialization.

Antes de um CER ser executada a CLR faz o preparo necessário para evitar uma possível falta de memória durante a compilação JIT (*Just-In-Time compilation*) ou carregar uma dependência. Além disso, a CLR ordena a execução do Finalizer dos objetos coletados durante o mesmo Garbage Collection. Os que não são marcados

como `critical` são executados antes que são, sendo possível que uma classe como `FileStream` realize o flush do seu buffer antes que o Finalizer do `SafeHandle`, usado por ela, seja finalizado.

2.5 EXCEPTION SHIELDING

Por Felipe Fujiy Pessoto

“O tratamento de erros dos meus serviços WCF não são consistentes e, em determinadas situações, expõe detalhes internos da aplicação, ou não são rastreáveis.”

Para evitar que detalhes da implementação do serviço sejam expostos, o WCF (Windows Communication Foundation) não envia para a aplicação cliente os detalhes de exceções desconhecidas, que poderiam ser usados para obter informações sobre o sistema, auxiliando o início um ataque. Assim, as exceções são recebidas como um `FaultException`, apenas alertando que ocorreu um erro interno.

É comum durante o desenvolvimento a necessidade de se ter detalhes dos erros para o *troubleshooting*. Para isso, existe o atributo `IncludeExceptionDetailInFaults` do elemento `ServiceDebug`, que deve ser usado de forma temporária para o debug. Porém, este atributo nunca deve ser utilizado com o valor `true` em produção, devido aos riscos de segurança. Desta forma, precisamos de uma solução em que os erros não exponham detalhes, e ainda assim seja possível rastreá-lo internamente.

O Enterprise Library é um conjunto de ferramentas e bibliotecas construídas em blocos que podem ser usados individualmente, ou de forma interligada, facilitando as boas práticas de programação e o gerenciamento de tarefas repetitivas, como Logging, Caching,

Data Access, entre outros. Nesta seção, serão utilizados dois de seus blocos, o *Exception Handling Application Block* e *Logging Application Block*.

Se a aplicação já utiliza o *Logging Block* do Enterprise Library, definindo como e onde o log é registrado, você pode apenas configurar a *exception policy* do *Exception Shielding* e reaproveitar as configurações previas.

O Enterprise Library é composto de blocos que podem ser interligados, isto é, podemos utilizar o *Exception Handling Block* que utiliza o *Logging Block* que, por sua vez, utiliza o *Data Access Block*. Se não for o seu caso, não se preocupe, vamos criar todas as configurações necessárias.

Instalando o Enterprise Library

O NuGet tem se tornado o principal canal de distribuição de componentes dos desenvolvedores que utilizam o Visual Studio. Todos os componentes do Enterprise Library estão disponíveis por ele.

Para utilizar o Exception Shielding, precisamos dos seguintes pacotes:

- `EnterpriseLibrary.ExceptionHandling.WCF`
- `EnterpriseLibrary.ExceptionHandling.Logging`

As demais dependências serão instaladas automaticamente.

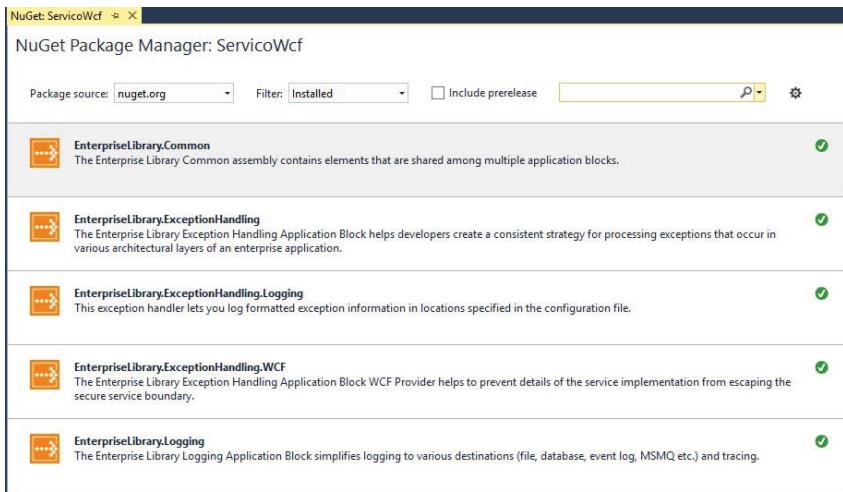


Figura 2.4: NuGet

Configurando o Logging e Exception Shielding

O Enterprise Library permite a configuração de seus blocos por meio do arquivo `config` da aplicação, ou da *Fluent API* via código.

A abordagem utilizando o arquivo `config` é a mais comum, portanto, a que será utilizada. Em suma, são feitas duas configurações: a primeira configurando o *Logging Application Block*, criando um *listener* para gravar no Event Log do Windows e, em seguida, o *Exception Handling Application Block*, que consome o Logging.

Como foi mencionado, os blocos são independentes. Desta forma, é possível alterar a configuração de Logging para gravar no banco de dados, arquivo XML, entre outros, sem alterar a configuração do Exception Handling.

Primeiramente, devemos adicionar as *Config Sections* dos respectivos blocos no arquivo de configuração:

```
<configSections>
    <section name="loggingConfiguration" type="Microsoft.Practices.E
```

```

nterpriseLibrary.Logging.Configuration.LoggingSettings, Microsoft.
Practices.EnterpriseLibrary.Logging, Version=6.0.0.0, Culture=neut
ral, PublicKeyToken=31bf3856ad364e35" requirePermission="true" />
    <section name="exceptionHandling" type="Microsoft.Practices.Entre
rpriseLibrary.ExceptionHandling.Configuration.ExceptionHandlingSet
tings, Microsoft.Practices.EnterpriseLibrary.ExceptionHandling, Ve
rsion=6.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" r
equirePermission="true" />
</configSections>

```

E então, as configurações dos blocos:

```

<loggingConfiguration name="" tracingEnabled="true" defaultCategor
y="General">
    <listeners>
        <add name="Event Log Listener" type="Microsoft.Practices.Enter
priseLibrary.Logging.TraceListeners.FormattedEventLogTraceListener
, Microsoft.Practices.EnterpriseLibrary.Logging, Version=6.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35"
            listenerDataType="Microsoft.Practices.EnterpriseLibrary.Logg
ing.Configuration.FormattedEventLogTraceListenerData, Microsoft.Pr
actices.EnterpriseLibrary.Logging, Version=6.0.0.0, Culture=neutra
l, PublicKeyToken=31bf3856ad364e35"
            source="Meu Serviço WCF" formatter="Text Formatter" log="App
lication"
            machineName="." traceOutputOptions="LogicalOperationStack, D
ateTime, Timestamp, ProcessId, ThreadId, Callstack" />
    </listeners>
    <formatters>
        <add type="Microsoft.Practices.EnterpriseLibrary.Logging.Forma
tters.TextFormatter, Microsoft.Practices.EnterpriseLibrary.Logging
, Version=6.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e3
5"
            template="Timestamp: {timestamp}{newline}&#xA;Message: {mess
age}{newline}&#xA;Category: {category}{newline}&#xA;Priority: {pri
ority}{newline}&#xA;EventId: {eventid}{newline}&#xA;Severity: {sev
erity}{newline}&#xA;Title:{title}{newline}&#xA;Machine: {localMach
ine}{newline}&#xA;App Domain: {localAppDomain}{newline}&#xA;Proces
sId: {localProcessId}{newline}&#xA;Process Name: {localProcessName
}{newline}&#xA;Thread Name: {threadName}{newline}&#xA;Win32 Thread
Id:{win32ThreadId}{newline}&#xA;Extended Properties: {dictionary({{
key} {value}}{newline})}"
            name="Text Formatter" />
    </formatters>
    <categorySources>
        <add switchValue="All" name="General">
            <listeners>
                <add name="Event Log Listener" />

```

```

        </listeners>
    </add>
</categorySources>
<specialSources>
    <allEvents switchValue="All" name="All Events" />
    <notProcessed switchValue="All" name="Unprocessed Category" />
    <errors switchValue="All" name="Logging Errors & Warnings" />
>
    <listeners>
        <add name="Event Log Listener" />
    </listeners>
    </errors>
</specialSources>
</loggingConfiguration>
<exceptionHandling>
    <exceptionPolicies>
        <add name="WCF Exception Shielding">
            <exceptionTypes>
                <add name="All Exceptions" type="System.Exception, mscorel
b, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e0
89">
                    postHandlingAction="ThrowNewException">
                    <exceptionHandlers>
                        <add name="Logging Exception Handler" type="Microsoft.
Practices.EnterpriseLibrary.ExceptionHandling.Logging.LoggingExcep
tionHandler, Microsoft.Practices.EnterpriseLibrary.ExceptionHandli
ng.Logging, Version=6.0.0.0, Culture=neutral, PublicKeyToken=31bf3
856ad364e35">
                            logCategory="General" eventId="100" severity="Error"
                            title="Enterprise Library Exception Handling"
                            formatterType="Microsoft.Practices.EnterpriseLibrary
.ExceptionHandling.TextExceptionFormatter, Microsoft.Practices.Ent
erpriseLibrary.ExceptionHandling"
                            priority="0" />
                        <add type="Microsoft.Practices.EnterpriseLibrary.Excep
tionHandling.WCF.FaultContractExceptionHandler, Microsoft.Practice
s.EnterpriseLibrary.ExceptionHandling.WCF, Version=6.0.0.0, Cultur
e=neutral, PublicKeyToken=31bf3856ad364e35">
                            exceptionMessage="Ocorreu um erro na transação. Por
                            favor entre em contato com o Suporte. Identificador do erro: {hand
                            lingInstanceId}"
                            faultContractType="ServicoWcf.ContratoFalha, Servico
                            Wcf" name="Fault Contract Exception Handler">
                                <mappings>
                                    <add source="{Guid}" name="HandlingInstanceId" />
                                </mappings>
                            </add>
                        </exceptionHandlers>

```

```
</add>
</exceptionTypes>
</add>
</exceptionPolicies>
</exceptionHandling>
```

Há algumas configurações que normalmente gostaríamos de alterar para cada projeto:

- Em `loggingConfiguration`, `listeners` e `add`, alteraremos o atributo `source` de acordo com o nome da aplicação. O `source` deve ser criado no sistema usando o comando PowerShell:

```
New EventLog LogName Application Source "Meu Serviço WCF"
```

- Dentro de `exceptionHandlers`, no segundo `add`, o atributo `faultContractType` deve conter o nome qualificado da classe que será usada como Fault Contract. O formato é **NomeDaClassComNamespace, NomeDoAssembly**, por exemplo, `My.Empresa.ContratoFalha, My.Empresa.`

```
public class ContratoFalha
{
    public Guid HandlingInstanceId { get; set; }
}
```

O Fault Contract geralmente contém informações úteis para a aplicação cliente, sem expor nenhuma informação sensível. Em nosso exemplo, a única propriedade será o `HandlingInstanceId`, garantindo a rastreabilidade dos erros. O mapeamento das informações que serão copiadas da Exception é encontrado no elemento `mapping` do `exceptionHandlers` no código IL do método `Finalize`, no começo desta seção.

Configurando o Exception Shielding

Após as configurações de Logging e Exception Handling, resta apenas definir o Exception Shielding nos serviços desejados por meio de atributos. Na interface do serviço, deve ser usado atributo `ExceptionShielding` e, nas operações, o `FaultContract` com o tipo definido na configuração anterior:

```
[ServiceContract]
[ExceptionShielding]
public interface IMeuServiço
{
    [OperationContract]
    [FaultContract(typeof(ContratoFalha))]
    string ObterValor(int value);
}
```

Rastreando os erros

Utilizando o `ExceptionShielding` e o arquivo de configuração, criamos uma `Policy` responsável por capturar os erros do serviço, gravar no `Event Log` e gerar uma nova `Exception` que inclui o `HandlingInstanceId`.

As aplicações clientes do serviço WCF receberão este código e podem também gravar a mensagem de erro contendo o `HandlingInstanceId`. No caso de aplicações web, este código pode ser informado ao usuário, que pode utilizar como referência ao solicitar o suporte. No servidor onde se encontra o serviço WCF, o resultado será um registro do Event Log como este:

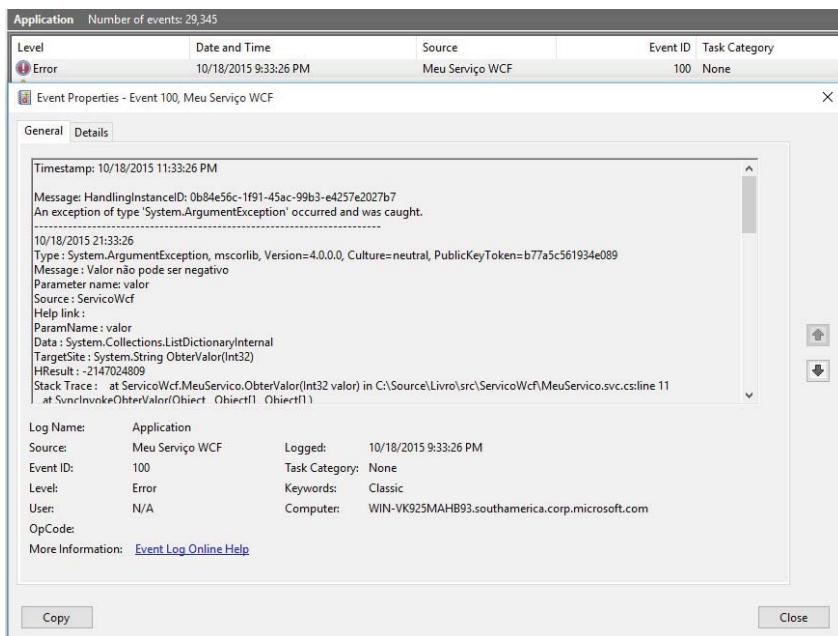


Figura 2.5: Event Log do WCF

A aplicação cliente pode, então, capturar o erro como um `FaultException<FaultContract>` e tratar as informações de forma apropriada, por exemplo, adicionando ao log da aplicação, ou mesmo exibindo a informação ao usuário.

```
using (MeuServicoClient cliente = new MeuServicoClient())
{
    try
    {
        ViewBag.Resultado = cliente.ObterValor(valor.Value);
    }
    catch (FaultException<ContratoFalha> ex)
    {
        ViewBag.MensagemErro = ex.Message;
        ViewBag.Guid = ex.Detail.HandlingInstanceId;
    }
}
```



Figura 2.6: Aplicação exibindo identificador de erro

O resultado é uma aplicação que não expõe detalhes sobre erros internos e, ainda assim, é possível rastreá-los de ponta a ponta utilizando o identificador único do erro.

2.6 PROPAGAÇÃO DE EXCEÇÕES

Por Fernando Henrique Inocêncio Borba Ferreira

“Tenho muitos blocos try/catch em meu projeto, isso torna o código deselegante e difícil de ler. E mesmo assim, as exceções apresentadas aos usuários não são relevantes e não me auxiliam na manutenção da aplicação e na resolução de erros.”

Trate as exceções apenas quando existir uma razão para fazê-lo. Não crie mecanismos de propagação de exceções entre suas chamadas de métodos. Permita que as exceções movam-se diretamente do bloco no qual foram lançadas para o bloco *catch* que vai tratá-las. Evite incluir em seu código blocos *try/catch* que não tratem as exceções e que apenas as lancem para a camada superior da pilha de execução.

O código da criação de nova thread utilizando `System.Threading.Thread` demonstra um bloco *try/catch* que

não realiza nenhum tratamento sobre a exceção, apenas faz o seu lançamento para a camada superior na pilha de execução. A figura mais à frente apresenta o fluxo da exceção entre as camadas da pilha de execução. Nota-se que neste código nenhuma ação é feita sobre a exceção capturada, apenas é feito o seu lançamento para uma camada superior.

```
public void MetodoB()
{
    try
    {
        // Código que pode gerar uma exceção
        MetodoC();
    }
    catch (SqlException se)
    {
        throw;
    }
}
```

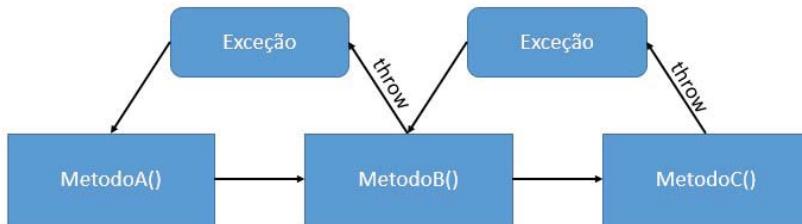


Figura 2.7: Propagação de exceções sem tratamento entre camadas

A existência de um bloco `try/catch` sugere que existe um tratamento adequado para a exceção. Se não é feito nenhum tratamento para essa exceção, então o bloco foi mal utilizado e deveria ser removido.

Existem três modelos de propagação de exceções:

- **Deixar que as exceções propaguem automaticamente:** neste modelo, a exceção é ignorada com o objetivo de que se mova diretamente para o bloco `try/catch` que

tenha um tratamento adequado. A figura a seguir exemplifica o fluxo de uma exceção entre uma hierarquia de chamadas de métodos quando temos apenas um bloco `try/catch` que provê o tratamento adequado para a exceção.

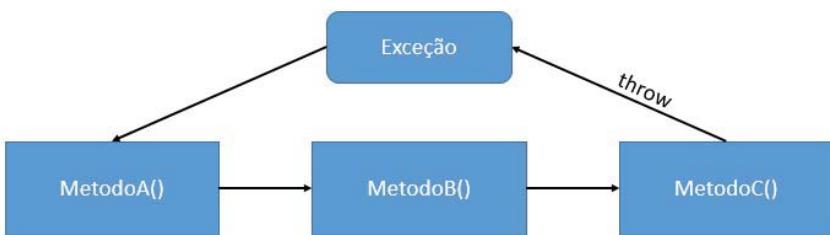


Figura 2.8: Propagação de uma exceção cujo único ponto de parada é o bloco `try/catch` que irá tratá-la adequadamente

- **Capturar a exceção, executar uma ação e lançar a exceção:** esta abordagem, ao contrário do problema citado, executa uma ação de contorno devido à existência de uma exceção e a lança para o bloco mais apto a tratá-la adequadamente. O código seguinte demonstra a propagação de uma exceção a partir de uma ação de contorno executada após uma exceção. Este uso é válido, pois o bloco `catch` executa uma ação de contorno a partir de uma exceção e lança-a para o bloco mais apto para fazer o seu tratamento.

```
public void MetodoC()
{
    try
    {
        // Código de persistência no banco de dados
    }
    catch (SqlException se)
    {
        // Código de envio de dados para uma fila de mensageria
        // para processamento posterior
    }
}
```

```
        throw;
    }
}
```

- **Capturar a exceção, tratá-la e lançar uma exceção mais específica:** este modelo preza que, ao propagar uma exceção pela pilha de execução, seu tipo de dados torna-se irrelevante. Assim, propõe-se que uma exceção mais relevante seja lançada ao chamador. A propriedade `InnerException` da classe `System.Exception` é usada com o intuito de manter a exceção original associada a uma nova e mais relevante exceção.

O código adiante descreve a situação prevista por esta abordagem, onde: uma `SqlException` é lançada e filtrada por um bloco `catch`, cujo tratamento é encapsulá-la na propriedade `InnerException` de um tipo de exceção mais específico e relevante ao chamador.

```
public void MetodoB()
{
    try
    {
        // Código que pode gerar uma exceção
        MetodoC();
    }
    catch (SqlException se)
    {
        throw new AcessoFonteDadosPrincipalException(se);
    }
}
```

A figura a seguir demonstra o fluxo de uma exceção por entre sua pilha de chamada, incluindo seu encapsulamento em um tipo de exceção mais relevante ao chamador.

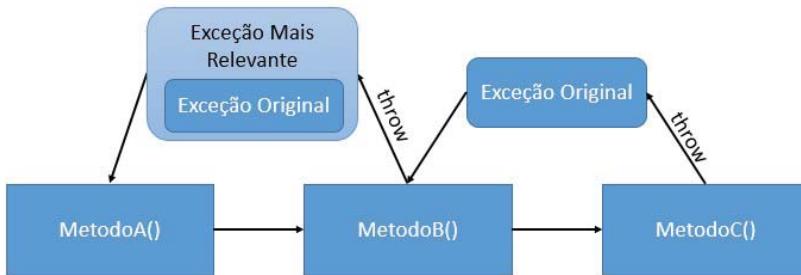


Figura 2.9: Propagação de uma exceção e o seu encapsulamento em uma exceção mais relevante ao chamador

Considere as implicações de performance geradas pelo lançamento de exceções. Em códigos bem escritos, blocos `try/finally` são mais comuns do que blocos `try/catch` (CWALINA; ABRAMS, 2009). Blocos `try/finally` devem ser utilizados para limpar e assegurar um estado consistente do sistema quando uma exceção é lançada.

REFERÊNCIAS

Guia de gerenciamento de exceções -
<http://www.microsoft.com/en-us/download/details.aspx?id=11748>

2.7 BUSCA EM MEMÓRIA

Por Fernando Henrique Inocêncio Borba Ferreira

“Tenho muitos dados em memória e minhas queries sobre esses dados estão lentas. Como posso fazer para minhas queries serem mais rápidas? ”

O surgimento da sintaxe LINQ, assim como a utilização de

query methods, facilitou a busca em memória. Com estes recursos, podemos facilmente executar *queries* em *arrays*, coleções e listas de tipos genéricos. O uso deste modelo de sintaxe agiliza o processo de desenvolvimento por tornar a busca em memória trivial e de simples codificação.

Mas, ao adotarmos esse modelo de busca linear (sequencial), estamos realmente escrevendo código performático? Seriam essas consultas o modelo mais rápido de busca em memória? Não perdemos poder computacional ou tempo de processamento ao adotar estes recursos em determinados cenários de pesquisa em memória? Dada a necessidade de executar consultas eficientes e com baixo custo computacional, devemos passar a evitar consultas que consumam muitos recursos computacionais e que sejam lentas.

Desta forma, passamos a adotar soluções alternativas de busca em memória que sejam mais performáticas para tornar nossos sistemas mais eficientes. Dentre tais opções, destacamos:

Busca binária

A busca binária é um algoritmo de busca que segue o paradigma da divisão e conquista. Partindo do pressuposto de que o conjunto de elementos está ordenado, são executadas diversas divisões do espaço de busca restringindo o possível local no qual o elemento buscado está posicionado. Partindo da comparação do elemento do meio do array, divide-se o array em segmentos, restringindo a superfície de busca e repetindo a comparação com o elemento no centro do segmento, até que o item procurado seja encontrado. A figura a seguir ilustra o processo de divisão do conjunto de elementos realizado pela busca de elementos.

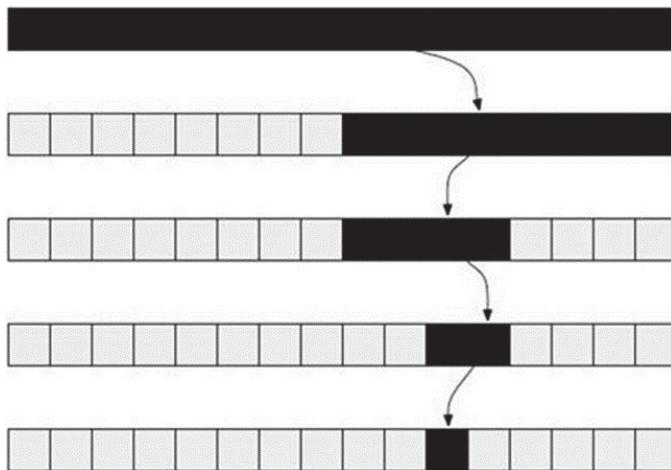


Figura 2.10: Representação da busca em memória

A busca binária possui complexidade algorítmica de $\Theta(\log n)$, enquanto que os algoritmos de busca linear possuem complexidade algorítmica de $\Theta(n)$ – dado que n é o tamanho do conjunto de elementos.

O algoritmo de busca binária é apresentado no código seguinte. Note que a sua execução depende da identificação de um ponto central no conjunto de elementos para que subconjuntos menores sejam criados. Vale ressaltar que este algoritmo é funcional apenas com elementos ordenados. Se os elementos não estiverem ordenados, então a busca binária não deve ser usada.

```
static bool ExecuteBinarySearch(IList<BasicStructure> targetList,
string searchParameter) {

    return ExecuteBinarySearch(targetList, searchParameter, 0, targetList.Count - 1);
}

static bool ExecuteBinarySearch(IList<BasicStructure> targetList,
string searchParameter, int start, int end) {

    if (start >= end)
        return false;
```

```

        int middlePoint = end - ((end - start) / 2);

        int compare = string.Compare(targetList[middlePoint].CodigoTipoRegistro, searchParameter);

        if (compare < 0) {
            return ExecuteBinarySearch(targetList, searchParameter, middlePoint, end);
        }
        else if (compare > 0) {
            return ExecuteBinarySearch(targetList, searchParameter, start, middlePoint - 1);
        }
        else {
            return true;
        }
    }
}

```

Busca linear

A busca linear nada mais é do que a varredura sequencial de um conjunto de elementos. Seja essa varredura iniciada na posição 0, na posição N ou com dupla validação de extremidades, ela é realizada item a item, sem utilizar o paradigma da divisão e conquista, ou qualquer outra abordagem. A sua complexidade algorítmica é de $\Theta(n)$.

O código a seguir apresenta diferentes métodos de busca linear, diferenciando-se apenas pelo operador de comparação usado, sendo eles:

- Método `ExecuteLinearSearchStringCompare` : executa a busca linear utilizando o método `String.Compare` para comparação de valores entre itens.
- Método `ExecuteLinearSearchStringEquals` : executa a busca linear usando o método `String.Equals` para comparação de valores entre

itens.

- Método

ExecuteLinearSearchStringEqualsCurrentCultureIgnoreCase : executa a busca linear utilizando o método `String.Equals`, ignorando diferenças de cultura e sendo *case insensitive*, para comparação de valores entre itens.

- Método

ExecuteLinearSearchStringCompareOrdinal : executa a busca linear usando o método `String.CompareOrdinal` para comparação de valores entre itens.

- Método ExecuteLinearSearchEqualityOperator : executa a busca linear utilizando o operador de igualdade (==) para comparação de valores entre itens.

```
static bool ExecuteLinearSearchStringCompare(IList<BasicStructure>
targetList, string searchParameter) {

    foreach (var item in targetList) {
        if (string.Compare(item.CodigoTipoRegistro, searchParameter
r) == 0)
            return true;
    }
    return false;
}

static bool ExecuteLinearSearchStringEquals(IList<BasicStructure>
targetList, string searchParameter) {

    foreach (var item in targetList) {
        if (string.Equals(item.CodigoTipoRegistro, searchParameter
))
            return true;
    }
    return false;
}
```

```

static bool ExecuteLinearSearchStringEqualsCurrentCultureIgnoreCase(IList<BasicStructure> targetList, string searchParameter) {

    foreach (var item in targetList) {
        if (string.Equals(item.CodigoTipoRegistro, searchParameter,
        StringComparison.CurrentCultureIgnoreCase))
            return true;
    }
    return false;
}

static bool ExecuteLinearSearchStringCompareOrdinal(IList<BasicStructure> targetList, string searchParameter) {

    foreach (var item in targetList) {
        if (string.CompareOrdinal(item.CodigoTipoRegistro, searchParameter) == 0)
            return true;
    }
    return false;
}

static bool ExecuteLinearSearchEqualityOperator(IList<BasicStructure> targetList, string searchParameter) {

    foreach (var item in targetList) {
        if (item.CodigoTipoRegistro == searchParameter)
            return true;
    }
    return false;
}

```

Hash tables e dicionários

A classe `Dictionary< TKey, TValue >` provê o mapeamento de uma coleção por meio da associação de um conjunto de chaves com um conjunto de valores. A busca de valores com dicionários é muito rápida, próxima ao estado da arte de $\Theta(1)$.

A velocidade dos dicionários se deve ao fato de serem implementados como *hash tables*. Hash tables (*hash map*) utilizam funções de hash para calcular um índice no array, dividindo-o em segmentos, nos quais os valores procurados são agrupados.

O código apresenta a busca em memória usando hash tables e dicionários.

```
static bool ExecuteDictionarySearch(IDictionary<string, BasicStructure> dictionary, string searchParameter) {  
  
    var searchValue = dictionary[searchParameter];  
  
    if (searchValue != null)  
        return true;  
  
    return false;  
}
```

Busca com LINQ

Estes últimos são os exemplos com queries LINQ. Eles são apresentados para que possamos identificar a diferença de tempo entre estas queries e as demais.

```
static bool ExecuteLinqAny(IList<BasicStructure> targetList, string searchParameter) {  
  
    var query = targetList.Any(e => e.CodigoTipoRegistro == searchParameter);  
    return query;  
}  
  
static bool ExecuteLinqFirstOrDefault(IList<BasicStructure> targetList, string searchParameter) {  
  
    var query = targetList.FirstOrDefault(e => e.CodigoTipoRegistro == searchParameter);  
    return (query != null);  
}
```

Medição dos tempos

Para identificar o tempo expendido na execução de cada query, foi criada uma classe chamada `TimeCollector`. Esta utiliza um `Stopwatch` internamente, coletando o início e o término da execução das queries.

```
public class TimeCollector : IDisposable {  
  
    private string _label;  
    private Stopwatch _watch;  
  
    public object Result { get; set; }  
  
    public TimeCollector(string label) {  
  
        this._label = label;  
        this._watch = Stopwatch.StartNew();  
    }  
  
    public void Dispose() {  
  
        this._watch.Stop();  
  
        Console.WriteLine(":: {0} Total: {1} / Result: {2}", thi  
s._label, this._watch.Elapsed.TotalMilliseconds, this.Result == nu  
ll ? string.Empty : this.Result.ToString());  
    }  
}
```

Testes de performance

Para validar a performance dos algoritmos de busca, utilizaremos uma lista com 100 mil itens em memória. Esses itens estão ordenados e a mesma lista será usada por todos os exemplos.

A seguir, temos o resultado de três execuções:

- Primeira execução:

```
file:///C:/FH/Internal/Livro/Source/ConsoleSearch/ConsoleSearch/bin/Debug/ConsoleSearch.EXE
Parâmetro buscado 075151
Busca com LINQ
:: Linq ANY - Total: 3.4566 / Result: True
:: Linq FIRSTORDEFAULT - Total: 2.0859 / Result: True
Buscas Lineares
:: Busca linear Básica - Total: 1.8621 / Result: True
:: Busca linear com String.Compare - Total: 18.1549 / Result: True
:: Busca linear com String.CompareOrdinal - Total: 1.8724 / Result: True
:: Busca linear com String.EqualsIgnoreCase - Total: 15.8612 / Result: True
:: Busca linear com String.Equals - Total: 2.2332 / Result: True
Busca binária
:: Busca binária - Total: 0.0135 / Result: True
Busca com dicionários
:: Dicionário - Total: 0.0045 / Result: True
```

- Segunda execução:

```
file:///C:/FH/Internal/Livro/Source/ConsoleSearch/ConsoleSearch/bin/Debug/ConsoleSearch.EXE
Parâmetro buscado 075790
Busca com LINQ
:: Linq ANY - Total: 2.4303 / Result: True
:: Linq FIRSTORDEFAULT - Total: 2.1023 / Result: True
Buscas Lineares
:: Busca linear Básica - Total: 1.9865 / Result: True
:: Busca linear com String.Compare - Total: 13.7129 / Result: True
:: Busca linear com String.CompareOrdinal - Total: 1.8449 / Result: True
:: Busca linear com String.EqualsIgnoreCase - Total: 15.2208 / Result: True
:: Busca linear com String.Equals - Total: 1.7312 / Result: True
Busca binária
:: Busca binária - Total: 0.0147 / Result: True
Busca com dicionários
:: Dicionário - Total: 0.0049 / Result: True
```

- Terceira execução:

```
file:///C:/FH/Internal/Livro/Source/ConsoleSearch/ConsoleSearch/bin/Debug/ConsoleSearch.EXE
Parâmetro buscado 069259
Busca com LINQ
:: Linq ANY - Total: 2.0621 / Result: True
:: Linq FIRSTORDEFAULT - Total: 1.9237 / Result: True
Buscas Lineares
:: Busca linear Básica - Total: 1.7344 / Result: True
:: Busca linear com String.Compare - Total: 12.5782 / Result: True
:: Busca linear com String.CompareTo - Total: 1.702 / Result: True
:: Busca linear com String.EqualsIgnoreCase - Total: 14.4433 / Result: True
:: Busca linear com String.Equals - Total: 1.5862 / Result: True
Busca binária
:: Busca binária - Total: 0.0114 / Result: True
Busca com dicionários
:: Dicionário - Total: 0.0045 / Result: True
```

Com estas execuções, podemos notar que:

- Os métodos de busca que usam LINQ possuem performance próxima a dos algoritmos de busca linear.
- O meio utilizado para comparação de strings pode ser tido como fator discriminante para uma execução mais rápida na busca linear.
- A busca binária mostrou mais rápida que as buscas lineares, chegando a ser 100 vezes mais rápida que o melhor caso da busca linear.
- A busca usando *hash tables* (por meio de dicionários) foi a mais rápida, sendo de 2 a 3 vezes mais rápida do que a busca binária.

Existem outros métodos de busca em memória tão performáticos quanto a busca binária (ou seja, busca em profundidade, busca em largura). É preciso conhecer tais algoritmos e saber quando os utilizar, pois são parte do nosso dia a dia e agregam mais performance a nossas aplicações.

Assim, neste tópico mostramos que sempre que for necessária a

pesquisa em memória, seja feita uma análise do cenário e da disposição dos dados, para que assim o melhor algoritmo seja usado.

O código seguinte apresenta na íntegra o código adotado deste tópico.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;

class Program {

    const int POPULATIONSIZE = 100000;
    const int RANDOM_SEED = 10;

    static void Main(string[] args) {

        IDictionary<string, BasicStructure> dictnry = new Dictionary<string, BasicStructure>();
        IList<BasicStructure> list = new List<BasicStructure>();

        PopulateList(list);
        PopulateDictionary(dictnry);

        Random rnd = new Random(RANDOM_SEED);
        do {

            Console.Clear();
            string SEARCHPARAMETER = rnd.Next(0, list.Count).ToString().PadLeft(POPULATIONSIZE.ToString().Length, '0');

            Console.WriteLine("Parâmetro buscado {0}", SEARCHPARAMETER);

            Console.WriteLine("Busca com LINQ");

            using (var timer = new TimeCollector("Linq ANY")) {
                timer.Result = ExecuteLinqAny(list, SEARCHPARAMETER);
            }

            using (var timer = new TimeCollector("Linq FIRSTORDEFALT")) {
                timer.Result = ExecuteLinqFirstOrDefault(list, SEA
```

```

RCHPARAMETER);
}

Console.WriteLine("Buscas Lineares");

using (var timer = new TimeCollector("Busca linear Básica")) {
    timer.Result = ExecuteLinearSearchEqualityOperator(list, SEARCHPARAMETER);
}

using (var timer = new TimeCollector("Busca linear com String.Compare")) {
    timer.Result = ExecuteLinearSearchStringCompare(list, SEARCHPARAMETER);
}

using (var timer = new TimeCollector("Busca linear com String.CompareOrdinal")) {
    timer.Result = ExecuteLinearSearchStringCompareOrdinal(list, SEARCHPARAMETER);
}

using (var timer = new TimeCollector("Busca linear com String.EqualsIgnoreCase")) {
    timer.Result = ExecuteLinearSearchStringEquals.CurrentCultureIgnoreCase(list, SEARCHPARAMETER);
}

using (var timer = new TimeCollector("Busca linear com String.Equals")) {
    timer.Result = ExecuteLinearSearchStringEquals(list, SEARCHPARAMETER);
}

Console.WriteLine("Busca binária");

using (var timer = new TimeCollector("Busca binária"))
{
    timer.Result = ExecuteBinarySearch(list, SEARCHPARAMETER);
}

Console.WriteLine("Busca com dicionários");

using (var timer = new TimeCollector("Dicionário")) {
    timer.Result = ExecuteDictionarySearch(dictnry, SEARCHPARAMETER);
}

```

```

    }

    Console.ReadLine();

    Console.WriteLine(
        ");
} while (true);
}

static void PopulateList(IList<BasicStructure> targetList) {

    for (int i = 0; i < POPULATIONSIZE; i++) {

        BasicStructure newNode = new BasicStructure();

        newNode.CodigoTipoRegistro = string.Format(i.ToString()
).PadLeft(POPULATIONSIZE.ToString().Length, '0'));

        targetList.Add(newNode);
    }
}

static void PopulateDictionary(IDictionary<string, BasicStructure> targetDictionary) {

    for (int i = 0; i < POPULATIONSIZE; i++) {

        BasicStructure newNode = new BasicStructure();

        newNode.CodigoTipoRegistro = string.Format(i.ToString()
).PadLeft(POPULATIONSIZE.ToString().Length, '0'));

        targetDictionary.Add(newNode.CodigoTipoRegistro, newNode);
    }
}

static bool ExecuteDictionarySearch(IDictionary<string, BasicStructure> dictionary, string searchParameter) {

    var searchValue = dictionary[searchParameter];

    if (searchValue != null)
        return true;

    return false;
}

```

```

        static bool ExecuteBinarySearch(IList<BasicStructure> targetLi
st, string searchParameter) {

            return ExecuteBinarySearch(targetList, searchParameter, 0,
targetList.Count - 1);
        }

        static bool ExecuteBinarySearch(IList<BasicStructure> targetLi
st, string searchParameter, int start, int end) {

            if (start >= end)
                return false;

            int middlePoint = end - ((end - start) / 2);
            int compare = string.Compare(targetList[middlePoint].CodigoTipoRegistro, searchParameter);

            if (compare < 0) {
                return ExecuteBinarySearch(targetList, searchParameter
, middlePoint, end);
            }
            else if (compare > 0) {
                return ExecuteBinarySearch(targetList, searchParameter
, start, middlePoint - 1);
            }
            else {
                return true;
            }
        }

        static bool ExecuteLinearSearchStringCompare(IList<BasicStruct
ure> targetList, string searchParameter) {
            foreach (var item in targetList) {
                if (string.Compare(item.CodigoTipoRegistro, searchPara
meter) == 0)
                    return true;
            }
            return false;
        }

        static bool ExecuteLinearSearchStringEquals(IList<BasicStructu
re> targetList, string searchParameter) {
            foreach (var item in targetList) {
                if (string.Equals(item.CodigoTipoRegistro, searchParam
eter))
                    return true;
            }
            return false;
        }
    }
}

```

```

    }

    static bool ExecuteLinearSearchStringEqualsCurrentCultureIgnoreCase(IList<BasicStructure> targetList, string searchParameter) {
        foreach (var item in targetList) {
            if (string.Equals(item.CodigoTipoRegistro, searchParameter, StringComparison.CurrentCultureIgnoreCase))
                return true;
        }
        return false;
    }

    static bool ExecuteLinearSearchStringCompareOrdinal(IList<BasicStructure> targetList, string searchParameter) {
        foreach (var item in targetList) {
            if (string.CompareOrdinal(item.CodigoTipoRegistro, searchParameter) == 0)
                return true;
        }
        return false;
    }

    static bool ExecuteLinearSearchEqualityOperator(IList<BasicStructure> targetList, string searchParameter) {
        foreach (var item in targetList) {
            if (item.CodigoTipoRegistro == searchParameter)
                return true;
        }
        return false;
    }

    static bool ExecuteLinqAny(IList<BasicStructure> targetList, string searchParameter) {
        var query = targetList.Any(e => e.CodigoTipoRegistro == searchParameter);
        return query;
    }

    static bool ExecuteLinqFirstOrDefault(IList<BasicStructure> targetList, string searchParameter) {
        var query = targetList.FirstOrDefault(e => e.CodigoTipoRegistro == searchParameter);
        return (query != null);
    }
}

public class TimeCollector : IDisposable {
    private string _label;

```

```

private Stopwatch _watch;

public object Result { get; set; }

public TimeCollector(string label) {
    this._label = label;
    this._watch = Stopwatch.StartNew();
}

public void Dispose() {
    this._watch.Stop();
    Console.WriteLine(":: {0} Total: {1} / Result: {2}", thi
s._label, this._watch.Elapsed.TotalMilliseconds, this.Result == nu
ll ? string.Empty : this.Result.ToString());
}
}

public class BasicStructure {
    public string CódigoTipoRegistro { get; set; }
    public object Value { get; set; }
}

```

2.8 MODELO PARA SERVIÇOS WINDOWS

Por Alexandre Teoi

Serviço Windows, ou *Windows service*, permite a criação de aplicações que ficam em execução por um longo período de tempo e, normalmente, sem interação com usuários. Essas aplicações são gerenciadas pelo Gerenciador de Controle de Serviços, ou *Services Control Manager* (SCM), que permite iniciar, terminar, pausar, retomar e configurar os serviços Windows.

Um exemplo de aplicação que implementa um serviço Windows é o Microsoft SQL Server. Uma das funcionalidades desse serviço é monitorar uma porta TCP específica para receber as consultas ao banco de dados provenientes de aplicações clientes.

O serviço Windows é uma funcionalidade que existe desde a primeira versão do Windows NT. Por isso, apesar de ter recebido atualizações e melhorias, hoje existem alternativas mais modernas e simples de desenvolver, implantar, escalar e manter - por exemplo, ASP.NET Web API ou serviços Windows Communication Foundation hospedados no Windows Process Activation Service.

Obviamente, existem casos onde serviços Windows se encaixam perfeitamente. Um exemplo são os serviços de processamento em lote. Nesses casos, a aplicação normalmente espera a chegada dos dados (por exemplo, via sistema de arquivos ou registros no banco de dados), para então iniciar o processamento dessas informações. Também é comum esse tipo de aplicação iniciar o processamento em horários pré-determinados, para evitar a concorrência com os sistemas online.

Nessa seção, vamos apresentar as práticas recomendadas para desenvolver um serviço Windows utilizando o .NET Framework. Apesar de o .NET Framework ter simplificado bastante o desenvolvimento de serviços Windows, existem alguns detalhes que farão o serviço interagir corretamente com o Gerenciador de Controle de Serviços, além de facilitar a monitoração e depuração da aplicação quando ela estiver rodando em um ambiente controlado de produção.

Criação do projeto de Serviço Windows

Para conseguir anexar o depurador a um processo do tipo serviço Windows, o Visual Studio precisa rodar em contexto de segurança elevado. Por isso, nesse módulo o Visual Studio deve ser executado como administrador.

O Visual Studio possui um modelo de projeto específico para serviços Windows. Para encontrá-lo, basta fazer uma busca por **windows service** na caixa de diálogo *New Project* do Visual Studio, como mostra a figura adiante. Após encontrar o modelo correto, crie um novo projeto com nome `ServicoExemplo`.

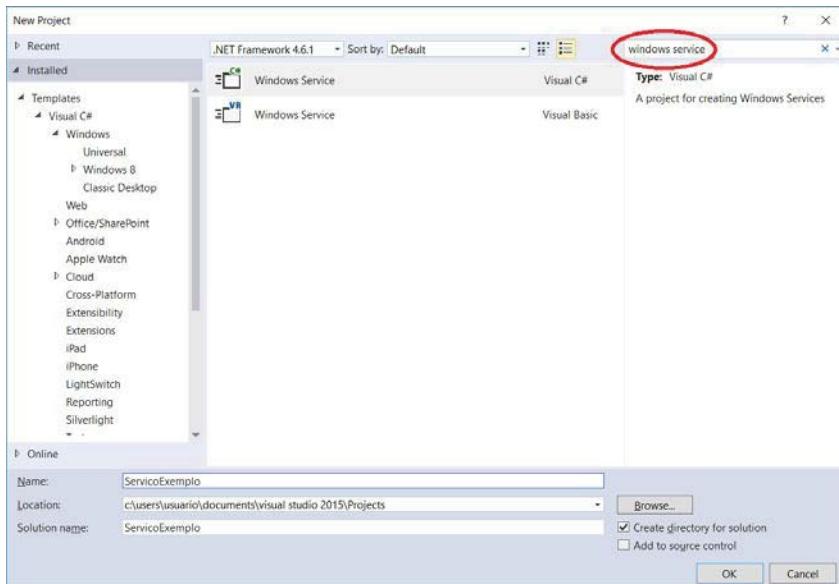


Figura 2.14: Criação de um novo projeto do tipo serviço Windows

O serviço Windows que vamos criar como exemplo vai receber requisições via protocolo HTTP e responder com um número da sequência de Fibonacci aleatório. Para que isso fique claro no nosso projeto, renomeie a classe `Service1` para `FibonacciAleatorio`,

como mostra a figura:

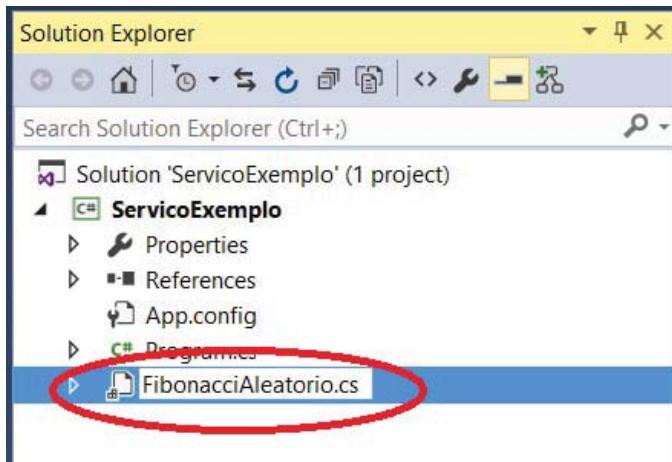


Figura 2.15: Renomeação da classe que implementa o serviço

Para facilitar o desenvolvimento do serviço, vamos alterar o método `Main` da classe `Program` para criar um executável que pode rodar tanto como console quanto serviço Windows, como mostra o código a seguir. A propriedade

`System.Environment.UserInteractive` nos informa se o processo está rodando em modo interativo. Vamos utilizá-lo para decidir entre abrir a janela do console ou rodar como serviço.

```
static void Main(params string[] args)
{
    if (Environment.UserInteractive)
    {
        FibonacciAleatorio servico = new FibonacciAleatorio();
        servico.ExecutarInterativo(args);
    }
    else
    {
        ServiceBase[] ServicesToRun;
        ServicesToRun = new ServiceBase[]
        {
            new FibonacciAleatorio()
        };
        ServiceBase.Run(ServicesToRun);
    }
}
```

```
    }  
}
```

Na classe `FibonacciAleatorio`, adicione o método `ExecutarInterativo`, como mostra o código a seguir. Esse método vai disparar os eventos de início e término da aplicação. Quando a aplicação roda em modo não interativo, o disparo desses eventos é feito pelo SCM.

```
public void ExecutarInterativo(string[] args)  
{  
    OnStart(args);  
  
    using (DebugConsole dc = new DebugConsole())  
    {  
        Console.WriteLine(  
            "Serviço em execução. Pressione ENTER para terminar.");  
    };  
    Console.ReadLine();  
}  
  
OnStop();  
}
```

Como o projeto desse programa não é do tipo *Console Application*, será preciso abrir a janela do console manualmente. A classe `DebugConsole` do código adiante encapsula a abertura dessa janela através de chamadas diretas para o sistema operacional Windows. Essa classe é usada no método `FibonacciAleatorio.ExecutarInterativo`.

```
using System;  
using System.ComponentModel;  
using System.Runtime.InteropServices;  
  
namespace ServicoExemplo  
{  
    internal static class NativeMethods  
    {  
        [DllImport("kernel32.dll", SetLastError = true)]  
        internal static extern bool AllocConsole();  
  
        [DllImport("kernel32.dll", SetLastError = true)]  
        internal static extern bool FreeConsole();
```

```

}

internal class DebugConsole : IDisposable
{
    bool disposed = false;
    bool consoleAlocado = false;

    public DebugConsole()
    {
        consoleAlocado = NativeMethods.AllocConsole();
        if (!consoleAlocado)
            throw new Win32Exception();
    }

    ~DebugConsole()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected void Dispose(bool disposing)
    {
        if (disposed)
            return;

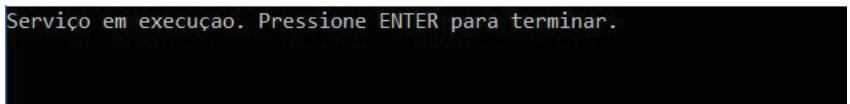
        if (disposing)
        {
            // Liberar outros recursos gerenciados
            //
        }

        // Liberar recursos nativos
        if (consoleAlocado)
        {
            NativeMethods.FreeConsole();
            consoleAlocado = false;
        }
        disposed = true;
    }
}

```

Com essas alterações, já é possível compilar e depurar o

exemplo. Ao executar o programa, a janela mostrada na figura seguinte deve aparecer.



```
Serviço em execução. Pressione ENTER para terminar.
```

Figura 2.16: Console do programa executado em modo interativo

Instalador do Serviço Windows

Para rodar a aplicação como serviço, ele precisa ser cadastrado no Gerenciador de Controle de Serviços. O .NET Framework disponibiliza duas classes para facilitar esse cadastramento:

- `System.ServiceProcess.ServiceProcessInstaller`
- `System.ServiceProcess.ServiceInstaller` .

Em teoria, é possível ter vários serviços hospedados em um único processo. A classe `ServiceProcessInstaller` é responsável por fazer a configuração do processo. Cada serviço que vai ser hospedado nesse processo deve ser configurado por uma instância da classe `ServiceInstaller` .

Para adicionar o instalador no nosso projeto com o Visual Studio, basta abrir o arquivo `FibonacciAleatorio.cs` em modo Design, abrir o menu de contexto através do botão direito do mouse e executar o comando *Add Installer*, como mostra a figura:

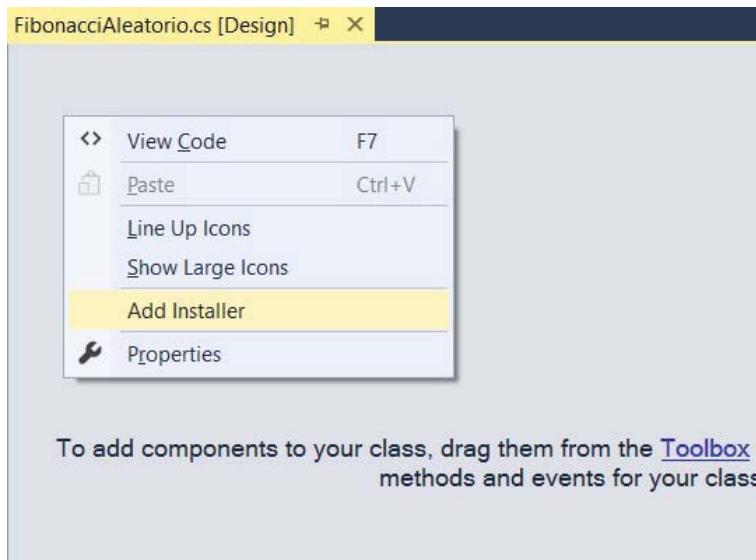


Figura 2.17: Adicionar instalador para o serviço

Esse comando vai incluir o arquivo `ProjectInstaller.cs` na solução. Para configurar os parâmetros de instalação do serviço, abra esse arquivo no modo Design, como mostra a figura:

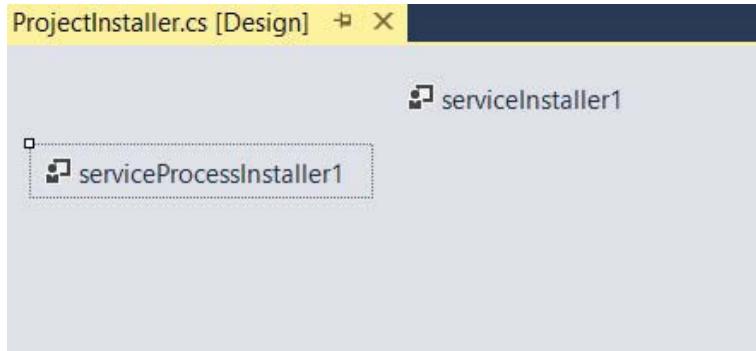


Figura 2.18: Modo Design do Arquivo ProjectInstaller.cs

Para configurar o contexto de segurança que o processo do serviço vai rodar, selecione o componente `serviceProcessInstaller1`, abra a janela *Properties* através da

tecla F4, e altere a propriedade Account para LocalSystem , como mostra a figura:

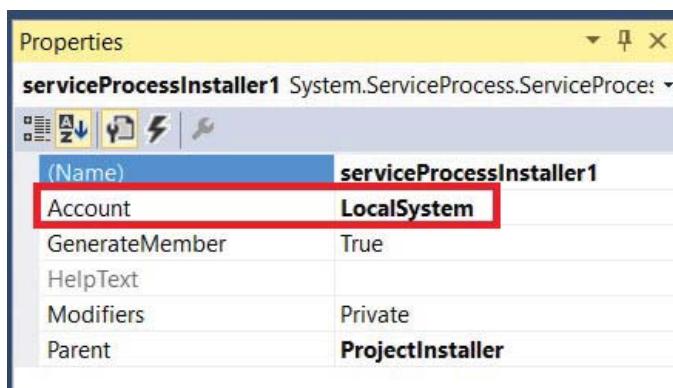


Figura 2.19: Configuração de instalação do processo

Por motivos de segurança, o recomendado é rodar serviços em contexto de contas com nível baixo de privilégio (por exemplo, a conta Network Service). Nesse exemplo, usaremos a conta Local System porque utilizaremos a classe `System.Net.HttpListener`. É possível utilizar essa classe com uma conta mais restrita, mas usaremos a conta Local System para não aumentar a complexidade do exemplo.

Para configurar as propriedades do serviço, selecione o componente `serviceInstaller1` e abra a janela *Properties* através da tecla F4. Altere os seguintes parâmetros desse componente:

- `Description` : calcula um número da sequência de Fibonacci aleatório
- `DisplayName` : Fibonacci Aleatório
- `ServiceName` : FibonacciAleatorio
- `StartType` : Manual

Os parâmetros `DisplayName` e `Description` são os textos apresentados na ferramenta de gerenciamento de serviços Windows (`services.msc`). O parâmetro `ServiceName` é utilizado pelo programa `SC.exe` para iniciar ou parar um serviço por meio de comando de linha. O parâmetro `StartType` indica se o serviço vai executar automaticamente após a inicialização do sistema operacional.

Após compilar a solução com essas alterações, é possível instalar o serviço no sistema operacional pela ferramenta `InstallUtil.exe`. A forma mais fácil de rodar essa ferramenta é pelo Developer Command Prompt do Visual Studio em modo administrativo. Se você estiver usando o Windows 10, utilize a `Cortana` para encontrar essa ferramenta, como mostra a figura:

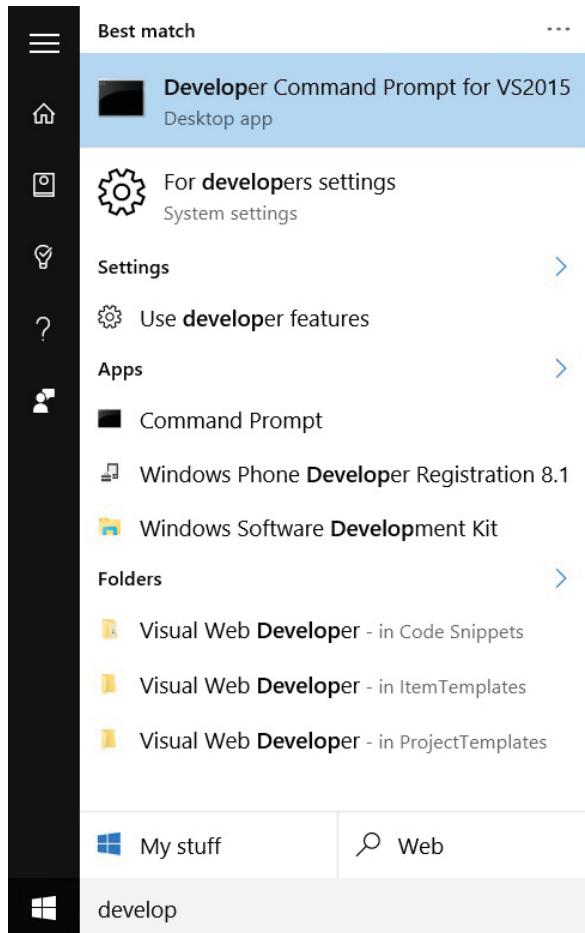


Figura 2.20: Developer Command Prompt do Visual Studio

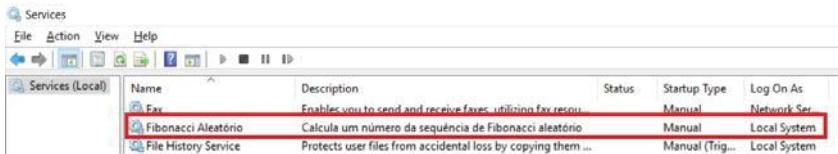
Depois de abrir o Developer Command Prompt como administrador, basta executar o comando `installutil.exe`, passando o caminho do executável gerado pelo nosso projeto, como mostra a figura seguinte. Para desinstalar o serviço, basta adicionar o parâmetro `/u` ao comando `installutil.exe`.

```
Administrator: Developer Command Prompt for VS2015
C:\Windows\system32>installutil C:\Users\usuario\Source\Repos\ServicoExemplo\ServicoExemplo\bin\Debug\ServicoExemplo.exe
```

A screenshot of a Windows Command Prompt window titled "Administrator: Developer Command Prompt for VS2015". The command `installutil C:\Users\usuario\Source\Repos\ServicoExemplo\ServicoExemplo\bin\Debug\ServicoExemplo.exe` is being typed into the prompt. The command has been partially entered, with the path to the executable file visible.

Figura 2.21: Comando de instalação do serviço

Com o serviço instalado, abra a ferramenta SERVICES.MSC (localizado na pasta SYSTEM32 do Windows) para confirmar que ele foi instalado corretamente, como mostra a figura:



Name	Description	Status	Startup Type	Log On As
Fax	Enables you to send and receive faxes, utilizing fax resources.	Manual	Manual	Network Service
Fibonacci Aleatório	Calcula um número da sequência de Fibonacci aleatório	Manual	Manual	Local System
File History Service	Protects user files from accidental loss by copying them ...	Manual (Trig...)	Manual (Trig...)	Local System

Figura 2.22: Serviço instalado corretamente

Para iniciar o serviço, selecione serviço na ferramenta e aperte o botão *Iniciar Serviço* localizado na barra de ferramentas.

Implementação do serviço

Agora que temos a infraestrutura pronta, podemos adicionar o código responsável por implementar as funcionalidades do nosso serviço. Nesse exemplo, o serviço receberá uma requisição por meio do protocolo HTTP, e responderá com um número aleatório da sequência de Fibonacci.

A classe que implementa o serviço precisa substituir os métodos `OnStart` e `OnEnd`, que são executados pelo Gerenciador de Controle de Serviços para iniciar e parar o serviço, respectivamente.

Por default, o serviço precisa iniciar em menos de 30 segundos. Se esse limite for atingido, o SCM cancelará o seu início. Por isso, a execução do método `OnStart` não pode demorar mais que esse tempo, ou o serviço não conseguirá subir.

Por esse motivo, recomenda-se criar threads nesse método para executar os códigos que vão fazer os processamentos das requisições. Nesse exemplo, usaremos a *Task Parallel Library*, ou TPL, para facilitar o gerenciamento dessas threads.

O código adiante contém os métodos que devem ser adicionados a classe `FibonacciAleatorio` para processar as requisições. O método `OuvirRequisicoes` prepara o serviço para receber as requisições HTTP. Ele utiliza a classe `System.Net.HttpListener`, que fornece toda a infraestrutura para implementar um servidor Web simples. Esse método recebe como parâmetro uma estrutura do tipo `System.Threading.CancellationToken`, que é usado para notificar o método que ele deve terminar sua execução.

O objeto `HttpListener` é configurado para responder no endereço http://*:5000/fibonacci. Para cada requisição recebida nesse endereço, o serviço cria uma nova tarefa (*Task*) para fazer o seu processamento. Essa tarefa executa o método `ProcessarRequisicao` que calcula um número aleatório da sequência de Fibonacci e retorna esse valor para o navegador por meio de uma página HTML. O método `ProcessarRequisicao` também recebe a estrutura `CancellationToken` para que possa fazer qualquer tratamento necessário antes de finalizar o serviço.

```
Random geradorAleatorio = new Random();

private void OuvirRequisicoes(CancellationToken ct)
{
    using (HttpListener listener = new HttpListener())
    {
        listener.Prefixes.Add("http://*:5000/fibonacci/");
        listener.Start();
        while (!ct.IsCancellationRequested)
        {
            var contextTask = listener.GetContextAsync();
            try
            {
                contextTask.Wait(ct);
                Task.Run(() =>
                {
                    ProcessarRequisicao(contextTask.Result, ct);
                });
            }
            catch (OperationCanceledException)
```

```

        {
        }
    }
}

private void ProcessarRequisicao(HttpContext context,
    CancellationToken ct)
{
    ct.ThrowIfCancellationRequested();

    HttpListenerResponse response = context.Response;
    int aleatorio = geradorAleatorio.Next(100000);
    ulong fibonacci = CalcularFibonacciCancelavel(aleatorio, ct);
    string output =
        $"<html><body>Fibonacci({aleatorio}) = {fibonacci}</body></html>";
    byte[] buffer = Encoding.UTF8.GetBytes(output);
    response.OutputStream.Write(buffer, 0, buffer.Length);
    response.OutputStream.Close();
}

private static ulong CalcularFibonacciCancelavel(int n,
    CancellationToken ct)
{
    ulong n0 = 0;
    ulong n1 = 1;
    for (int i = 0; i < n; i++)
    {
        ct.ThrowIfCancellationRequested();

        ulong tmp = n0 + n1;
        n0 = n1;
        n1 = tmp;
    }
    return n0;
}

```

O método `OnStart` precisa iniciar uma thread para executar o método `OuvirRequisicoes`. Isso pode ser feito de uma forma simples através do método `System.Threading.Tasks.Task.Run` da TPL. O método `OnStop` precisa sinalizar o fim da execução através da estrutura `CancellationToken`, e esperar o retorno do método `OuvirRequisicoes`. O código a seguir mostra a implementação dos métodos `OnStart` e `OnStop`.

```

CancellationTokenSource cts = new CancellationTokenSource();
Task principal = null;

protected override void OnStart(string[] args)
{
    principal = Task.Run(() => { OuvirRequisicoes(cts.Token); });
}

protected override void OnStop()
{
    cts.Cancel();
    principal.Wait();
}

```

Com essas alterações, o serviço já deve estar pronto para rodar. Execute o programa através do Visual Studio (lembre-se de rodar o Visual Studio como administrador), ou instale o serviço e o execute pela aplicação de gerenciamento de serviços (SERVICES.MSC). Com o seu navegador preferido, abra o endereço <http://localhost:5000/fibonacci>, e uma página similar ao da figura seguinte deve aparecer no seu navegador.

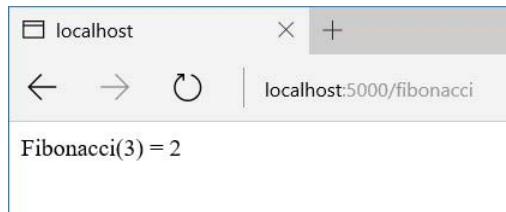


Figura 2.23: Serviço em execução

Registro de eventos

Apesar de já termos uma aplicação funcionando, se a implantarmos dessa forma no ambiente de produção, caso ocorra algum erro, será impossível descobrir sua causa. No nosso exemplo, se o método `ProcessarRequisicao` gerar uma exceção, não haverá nenhum registro para auxiliar em um futuro processo de resolução de problema. Na plataforma Windows, o melhor lugar para registrar esses eventos é no Log de Eventos.

Muitos preferem criar uma tabela no banco de dados para armazenar os registros de eventos. Mas, se o problema for com o banco de dados, não será possível fazer o registro. Por isso, recomenda-se também usar o Log de Eventos. Além disso, praticamente todas as ferramentas de monitoração de aplicação suportam o Log de Eventos. Isso permite que alertas sejam gerados por essas ferramentas caso o serviço tenha algum problema.

Para utilizar o Log de Eventos do Windows, você precisa criar uma origem no Log de Eventos na instalação do serviço. O .NET Framework disponibiliza a classe `System.Diagnostics.EventLogInstaller` para facilitar a criação dessa origem. Para utilizar essa classe, abra o código da classe `ProjectInstaller` e altere o construtor como mostra o código:

```
public ProjectInstaller()
{
    InitializeComponent();

    EventLogInstaller eli = new EventLogInstaller();
    eli.Source = FibonacciAleatorio.origemLogEvento;
    eli.Log = FibonacciAleatorio.nomeLogEvento;
    Installers.Add(eli);
}
```

Adicione os parâmetros do código seguinte à classe `FibonacciAleatorio` com os nomes do Log de Evento e a origem que usaremos para fazer os registros.

```
public partial class FibonacciAleatorio : ServiceBase
{
    public const string nomeLogEvento = "Application";
    public const string origemLogEvento = "Fibonacci Aleatorio";
    public enum IdEventos
    {
        ErroInicializacao,
        ErroProcessamento
    }
}
```

No nosso exemplo, geraremos eventos em dois métodos: `OuvirRequisicoes` e `ProcessarRequisicao`. No método

OuvirRequisicoes , registraremos um evento de erro caso ocorra algum erro durante a configuração inicial do nosso serviço e chamaremos o método ServiceBase.Stop para avisar o SCM que o serviço foi parado, como mostra o código:

```
private void OuvirRequisicoes(CancellationToken ct)
{
    try
    {
        using (HttpListener listener = new HttpListener())
        {
            listener.Prefixes.Add("http://*:5000/fibonacci/");
            listener.Start();
            while (!ct.IsCancellationRequested)
            {
                var contextTask = listener.GetContextAsync();
                try
                {
                    contextTask.Wait(ct);
                    Task.Run(() =>
                    {
                        ProcessarRequisicao(contextTask.Result, ct
);
                    });
                }
                catch (OperationCanceledException)
                {
                }
            }
        }
    }
    catch (Exception ex)
    {
        EventLog el = new EventLog(nomeLogEvento, ".", origemLogEvento);
        el.WriteEntry("Erro ao iniciar o serviço:\n" + ex.ToString
(),
            EventLogEntryType.Error,
            (int)IdEventos.ErroInicializacao);
        principal.ContinueWith((antecedent) => { Stop(); });
    }
}
```

No método ProcessarRequisicao , registraremos avisos caso ocorra algum erro durante o processamento de alguma requisição.

Para isso, vamos colocar um bloco `try/catch` para capturar qualquer exceção e gravar o seu conteúdo no Log de Eventos, como mostra o código:

```
private void ProcessarRequisicao(HttpContext context,
    CancellationToken ct)
{
    try
    {
        ct.ThrowIfCancellationRequested();

        HttpListenerResponse response = context.Response;
        int aleatorio = geradorAleatorio.Next(100000);
        ulong fibonacci = CalcularFibonacciCancelavel(aleatorio, c
t);
        string output =
            $"<html><body>Fibonacci({aleatorio}) = {fibonacci}</body
></html>";
        byte[] buffer = Encoding.UTF8.GetBytes(output);
        response.OutputStream.Write(buffer, 0, buffer.Length);
        response.OutputStream.Close();
    }
    catch (OperationCanceledException)
    {
        throw;
    }
    catch (Exception ex)
    {
        EventLog el = new EventLog(nomeLogEvento, ".", origemLogEv
ento);
        el.WriteEntry("Erro ao processar requisição:\n" + ex.ToStr
ing(),
            EventLogEntryType.Warning,
            (int)IdEventos.ErroProcessamento);
    }
}
```

Para utilizar a nova versão da aplicação, será necessário reinstalar o serviço para criar a origem do Log de Eventos. Para desinstalar a versão anterior, use o comando `INSTALLUTIL.EXE` com o parâmetro `-u` e passando o caminho do executável do serviço. Após desinstalar, reinstale com o comando `INSTALLUTIL.EXE` como mostra a figura *Comando de instalação do serviço*.

Contadores de desempenho

O log de eventos é muito útil para saber se houve algum problema durante a execução do nosso serviço, mas não fornece informações sobre o seu desempenho. Normalmente, em ambiente de produção, também é importante fornecer indicadores de desempenho para que ações sejam tomadas rapidamente caso a aplicação comece a mostrar algum sinal de degradação.

No ambiente Windows, a forma padrão de fornecer indicadores de desempenho é pelos contadores de desempenho. Esses contadores podem ser visualizados graficamente através da ferramenta Monitor de Desempenho (`PERFMON.EXE`), e todas ferramentas de monitoração de datacenters conseguem ler esses contadores e gerar alertas por meio de configuração de valores limites.

Para gerar contadores de desempenho, precisamos registrar a categoria e os contadores propriamente ditos no sistema operacional. O `.NET Framework` disponibiliza a classe

`System.Diagnostics.PerformanceCounterInstaller` para facilitar esse registro.

Nesse exemplo, utilizaremos uma categoria com nome **Fibonacci Aleatório**, e três contadores:

- **Taxa de requisições:** quantidade de requisições processadas por segundo;
- **Tempo de execução:** tempo gasto para processar uma requisição em milissegundos;
- **Taxa de erro:** quantidade de erros de processamento por segundo.

Os nomes da categoria e contadores de desempenho serão definidos como constantes na classe `FibonacciAleatorio`, como

mostra o código:

```
public partial class FibonacciAleatorio : ServiceBase
{
    public const string NomeCategoriaContadorDesempenho = "Fibonaci Aleatório";
    public const string NomeContadorTaxaRequisicoes = "Taxa de requisições";
    public const string NomeContadorTempoExecucao = "Tempo de execução";
    public const string NomeContadorTaxaErro = "Taxa de erro";
    PerformanceCounter taxaRequisicoes = new PerformanceCounter(
        NomeCategoriaContadorDesempenho, NomeContadorTaxaRequisicoes, false);
    PerformanceCounter tempoExecucao = new PerformanceCounter(
        NomeCategoriaContadorDesempenho, NomeContadorTempoExecucao, false);
    PerformanceCounter taxaErro = new PerformanceCounter(
        NomeCategoriaContadorDesempenho, NomeContadorTaxaErro, false);
```

Usaremos a classe `PerformanceCounterInstaller` no construtor da classe `ProjectInstaller`, como mostra o código seguinte. Repare como utilizamos dois tipos de contadores:

- `RateOfCountsPerSecond32` : contadores desse tipo são mostrados como taxa por segundo. Toda vez que completarmos uma operação, esse contador deve ser incrementado.
- `NumberOfItems32` : o valor desse tipo de contador deve ser gravado diretamente pela aplicação. Não é feita nenhuma operação para converter o valor de contadores desse tipo.

```
public ProjectInstaller()
{
    InitializeComponent();

    EventLogInstaller eli = new EventLogInstaller();
    eli.Source = FibonacciAleatorio.OrigemLogEvento;
    eli.Log = FibonacciAleatorio.NomeLogEvento;
    Installers.Add(eli);
```

```

    PerformanceCounterInstaller pci = new PerformanceCounterInstaller();
    pci.CategoryName = FibonacciAleatorio.NomeCategoriaContadorDesempenho;
    pci.CategoryHelp =
        "Informação de desempenho do serviço Fibonacci Aleatório.";
    pci.CategoryType = PerformanceCounterCategoryType.SingleInstance;

    CounterCreationDataCollection dados =
        new CounterCreationDataCollection();
    CounterCreationData dado = new CounterCreationData();
    dado.CounterName = FibonacciAleatorio.NomeContadorTaxaRequisicoes;
    dado.CounterHelp = "Quantidade de requisições processadas por segundo.";
    dado.CounterType = PerformanceCounterType.RateOfCountsPerSecond;
    dados.Add(dado);

    dado = new CounterCreationData();
    dado.CounterName = FibonacciAleatorio.NomeContadorTempoExecucao;
    dado.CounterHelp =
        "Tempo gasto para processar uma requisição em milissegundos.";
    dado.CounterType = PerformanceCounterType.NumberOfItems32;
    dados.Add(dado);

    dado = new CounterCreationData();
    dado.CounterName = FibonacciAleatorio.NomeContadorTaxaErro;
    dado.CounterHelp = "Quantidade de erros de processamento por segundo.";
    dado.CounterType = PerformanceCounterType.RateOfCountsPerSecond;
    dados.Add(dado);

    pci.Counters.AddRange(dados);
    Installers.Add(pci);
}

```

Com esses contadores, modificaremos o método `FibonacciAleatorio.ProcessarRequisicao` para atualizar os seus valores. A classe `System.Diagnostics.Stopwatch` será utilizada para calcular o tempo gasto no processamento. Se o

processamento terminar com sucesso, incrementaremos o contador de taxa de requisições, e atualizaremos o contador de tempo de requisição. Caso ocorra uma exceção, incrementaremos o contador taxa de erro.

O código a seguir mostra como atualizar esses contadores.

```
private void ProcessarRequisicao(HttpContext context, CancellationToken ct)
{
    try
    {
        ct.ThrowIfCancellationRequested();

        Stopwatch sw = new Stopwatch();
        sw.Start();

        HttpResponseMessage response = context.Response;
        int aleatorio = geradorAleatorio.Next(100000);
        ulong fibonacci = CalcularFibonacciCancelavel(aleatorio, c
t);
        string output =
        $"<html><body>Fibonacci({aleatorio}) = {fibonacci}</body></htm
l>";
        byte[] buffer = Encoding.UTF8.GetBytes(output);
        response.OutputStream.Write(buffer, 0, buffer.Length);
        response.OutputStream.Close();

        sw.Stop();
        taxaRequisicoes.Increment();
        tempoExecucao.RawValue = sw.ElapsedMilliseconds;
    }
    catch (OperationCanceledException)
    {
        throw;
    }
    catch (Exception ex)
    {
        EventLog el = new EventLog(NomeLogEvento, ".", OrigemLogEv
ento);
        el.WriteEntry("Erro ao processar requisição:\n" + ex.ToStr
ing(),
            EventLogEntryType.Warning,
            (int)IdEventos.ErroProcessamento);
        taxaErro.Increment();
    }
}
```

}

Com essas alterações, podemos compilar a solução e reinstalar o serviço através do comando `INSTALLUTIL.EXE` para cadastrar os contadores de desempenho. Para visualizar esses contadores, utilize a ferramenta Monitor de Desempenho (`PERFMON.EXE`). Com o seu navegador preferido, tente enviar algumas requisições para o serviço pelo endereço <http://localhost:5000/fibonacci>. Se tudo estiver certo, os valores dos contadores devem ser atualizados como mostra a figura a seguir.

Se o serviço rodar em um contexto de um usuário que não seja administrador, ele deve ser incluído no grupo **USUÁRIOS DE MONITOR DE DESEMPENHO** (*Performance Monitor Users*) para poder atualizar os contadores de desempenho.

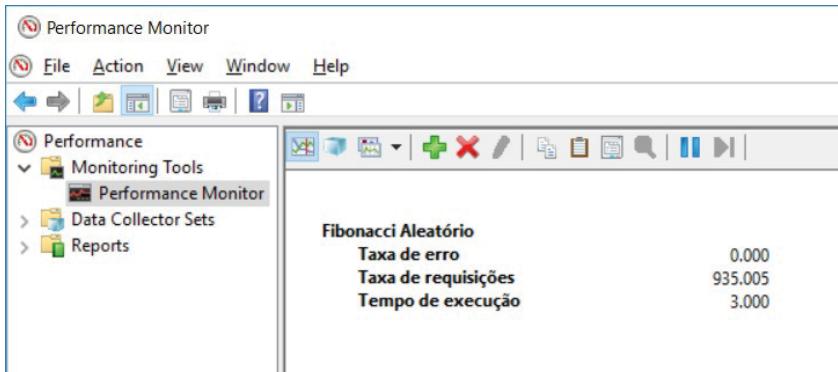


Figura 2.24: Contadores de desempenho do serviço

Resumindo

Neste capítulo, vimos como criar um serviço Windows que interage corretamente com o Gerenciador de Controle de Serviços. Ele está preparado para configurar os recursos do sistema

operacional necessários para a sua execução (cadastramento do serviço, origem de Log de Eventos e contadores de desempenho). Além disso, o serviço está instrumentando, e pode ser monitorado facilmente quando estiver execução em um ambiente de produção controlado.

REFERÊNCIAS

Windows services - <https://msdn.microsoft.com/en-us/library/windows/desktop/ms685141.aspx>

Services Control Manager (SCM) -
<https://msdn.microsoft.com/en-us/library/windows/desktop/ms685150.aspx>

ASP.NET Web API - <http://www.asp.net/web-api>

Windows Communication Foundation -
<https://msdn.microsoft.com/en-us/library/dd456779.aspx>

Windows Process Activation Service -
<https://msdn.microsoft.com/en-us/library/ms734677.aspx>

SC.exe - <https://technet.microsoft.com/en-us/library/807191da-4a33-4149-addf-c11ded938b5f>

InstallUtil.exe - <https://msdn.microsoft.com/en-us/library/50614e95.aspx>

Cortana - <http://windows.microsoft.com/en-us/windows-10/getstarted-what-is-cortana>

Límite de tempo do SCN - <https://support.microsoft.com/en-us/kb/922918>

Task Parallel Library (TPL) - <https://msdn.microsoft.com/en-us/library/hh194605.aspx>

[us/library/dd537609.aspx](https://msdn.microsoft.com/en-us/library/dd537609.aspx)

System.Diagnostics.PerformanceCounterInstaller
<https://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounterinstaller.aspx>

System.Diagnostics.Stopwatch
<https://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch.aspx>

2.9 UTILIZANDO SERVER NAME INDICATIONS

Por Leandro Almeida

“Não consigo configurar mais de um WebSite HTTPS sem que haja a necessidade de um IP adicional para a ligação (binding) HTTP.”

Durante o início do processo de criptografia de requisições, o Host Header não está disponível para o servidor. Sendo as únicas informações disponíveis os IPs de destino e a porta de destino. Devido a essa restrição do protocolo, quando configurado mais de um website HTTPS no servidor web, não é possível encaminhar a requisição HTTPS para o website correto, caso não haja um IP dedicado e configurado na ligação (*binding*) do website.

Server Name Indications são uma extensão do protocolo SSL/TLS (*Secure Sockets Layer/Transport Layer Security*), na qual o *server name* (nome do servidor) será fornecido, possibilitando assim que o IIS (*Internet Information Services*) identifique qual é o website correto. O cliente envia a mensagem “*clienthello*”, que contém a

extensão Server Name Indication, no início do processo de negociação para estabelecer uma conexão segura (processo conhecido como “*handshake*”).

O IIS 8.0 (disponível no Windows Server 2012) suporta Server Name Indications. Desta forma, torna-se possível configurar a propriedade host name no binding de websites HTTPS. Isso possibilita o IIS encaminhar a requisição para o website correto, sem a necessidade de um IP dedicado para o website.

Habilitando Server Name Indication

Para habilitar o Server Name Indication no IIS 8.0, será utilizada a interface gráfica de administração do IIS. O Server Name Indication é habilitado durante a configuração de binding do website. Para isto, é necessário configurar o host name com a URL da aplicação que será exposta, e selecionar a opção “*Require Server Name Indication*”, conforme exibido na figura a seguir:

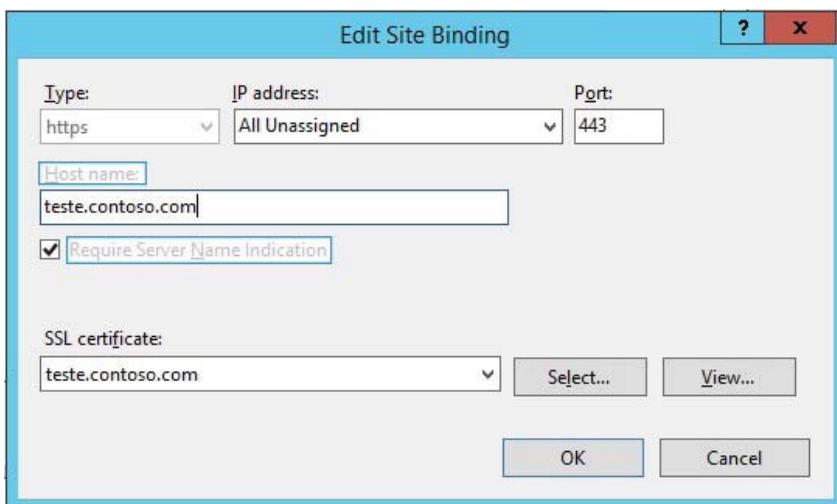


Figura 2.25: Habilitando Server Name Indication

Dentro do arquivo de configuração do IIS, chamado

`ApplicationHost.Config` (arquivo onde é armazenado as configurações comuns do IIS), podemos visualizar a configuração do Server Name Indication pela propriedade `sslFlags` nas configurações dos bindings dos websites. Essa propriedade indica que o website que o IIS exposto utiliza Server Name Indications.

```
<bindings>
  <binding protocol="http" bindingInformation="*:80;" />
  <binding protocol="https" bindingInformation="*:443:teste.contoso.com" sslFlags="1" />
</bindings>
```

A tabela a seguir descreve os possíveis valores suportados para a propriedade `sslFlags`:

SslFlags	Descrição
0	SNI não configurado
1	SNI configurado
2	SNI não configurado usando Certificado Centralizado (permite o administrador do servidor armazenar e acessar os certificados através de um compartilhamento)
3	SNI configurado usando Certificado Centralizado

Como funciona os Server Name Indications?

Durante o processo de *handshake*, o cliente envia uma mensagem *clienthello* para o servidor e aguarda a sua resposta. No conteúdo da mensagem *clienthello*, é enviada a extensão Server Name (figura a seguir), contendo o website que será responsável por receber a requisição.

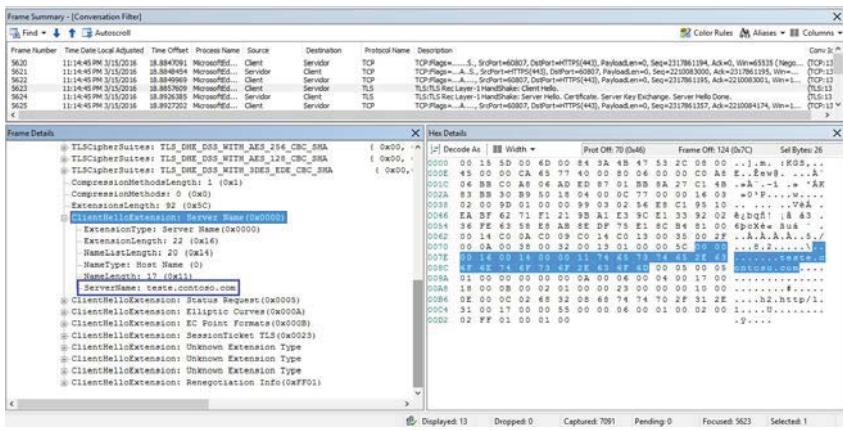


Figura 2.27: Server Name Extension

Quando a requisição chega ao servidor web, o IIS identifica que o cliente suporta Server Name Indications por meio de um módulo chamado `HTTP.sys`. Este faz parte do kernel do Windows, e sua função é receber requisições HTTP e transferi-las ao IIS. Assim, após a identificação e o encaminhamento da requisição para o website, o seu processamento é iniciado.

Resumindo

A utilização de Server Name Indications proporciona dois benefícios:

1. Aumento da escalabilidade e da segurança, pois permite adicionar quantos sites seguros forem necessários a um mesmo servidor;
2. Redução no número de IPs em uso, pois não se faz mais necessária a utilização de um IP exclusivo a um único site seguro.

O uso de Server Name Indications possui uma única ressalva: o browser do cliente precisa suportar a extensão. Durante a escrita deste livro, a maioria dos browsers suporta o Server Name

Indications. O link https://en.wikipedia.org/wiki/Server_Name_Indication apresenta informações relacionadas aos browsers suportados.

REFERÊNCIAS

Introduction to ApplicationHost.Config – <http://www.iis.net/learn/get-started/planning-your-iis-architecture/introduction-to-applicationhostconfig>

IIS 8.0 Centralized SSL Certificate Support - SSL Scalability and Manageability – <http://www.iis.net/learn/get-started/whats-new-in-iis-8/iis-80-centralized-ssl-certificate-support-ssl-scalability-and-manageability>

2.10 CONCLUSÃO

Durante este capítulo, discutiram-se problemas comuns aos times de desenvolvimento. Nota-se que, para resolução da maioria destes problemas, não nos basta entender o que cada elemento do .NET Framework faz, é preciso compreender como estes elementos devem ser combinados para criação de soluções robustas.

Planejar novas demandas e gerenciar atividades definitivamente não são tarefas fáceis. Responder a feedbacks e aumentar a qualidade no ciclo de vida dos projetos podem ser as soluções. No próximo capítulo, serão apresentadas técnicas ágeis para planejar requisitos, gerir demandas, controlar o código-fonte de forma distribuída, trabalhar em conjunto com fábricas de software para uma demanda em comum, e facilitar a forma com que os trabalhos são geridos.

CAPÍTULO 3

PLANEJAMENTO E GESTÃO DE DEMANDAS

Um dos grandes desafios de quem desenvolve e entrega software é planejar e gerir suas demandas. Com um bom planejamento de projetos e uma gestão de demandas inteligente, é possível tirar maior proveito dos recursos do projeto para trazer maior valor ao negócio, principalmente por ter consciência do que vem pela frente.

Conseguir organizar, construir e priorizar um *backlog* consistente de acordo com a necessidade dos interessados é uma arte nada fácil de dominar. A seguir, serão abordadas algumas formas de ter sucesso nos próximos projetos, desde novas formas de planejar seus requisitos com práticas ágeis até formas de guiar o projeto baseado em feedback.

3.1 PROBLEMAS NA GESTÃO DE REQUISITOS E SUAS PRINCIPAIS CAUSAS

Por Luiz Macedo

“Os requisitos não estão refletindo as reais necessidades dos usuários. Eles estão incompletos e inconsistentes. Isso aconteceu porque passaram por muitas mudanças após a aprovação. Isso está gerando retrabalho, atrasos no cronograma e claro, mais custo para o projeto.”

Em média, 1/3 dos defeitos dos sistemas são produto de erros nos requisitos. Muitas funcionalidades entregues não são e nunca serão utilizadas. Levando em consideração que o desenvolvimento de software é crítico para o negócio e normalmente atrasos em entregas têm um impacto real.

Parte dessas afirmações faz muito sentido, mas por que os requisitos não estão refletindo as necessidades do usuário? Esses mesmos requisitos são detalhados no início do desenvolvimento? Quando isso acontece, normalmente é porque estão sendo definidos quando ainda não existe conhecimento suficiente sobre o software.

Os problemas dessa abordagem é que a maior parte das decisões é tomada com base em suposições, gerando problemas como o grande número de defeitos causados por erro de requisitos incompletos, ou até mesmo esforço gasto em funcionalidades que não são relevantes ou não trazem valor para o produto, aumentando assim o prazo e custo dos projetos.

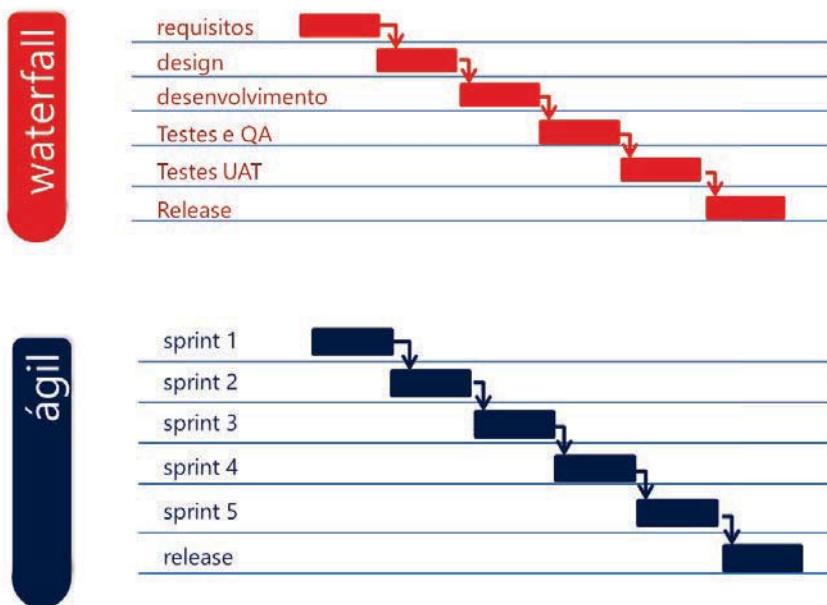
Todas essas questões levam à consequência da mudança, o time investe grande esforço no início do ciclo de desenvolvimento com grandes especificações para evitar mudanças. Mas ao não conseguir evitá-las, acabam gerando retrabalho, atrasos nas entregas e maiores custos ao projeto.

A última e uma das principais causas é a comunicação ineficiente e a falta de envolvimento dos interessados no projeto (stakeholders). Mas para isso veja o tópico sobre “Planejamento de projeto guiado a feedback”.

Requisitos ágeis

Fazendo um comparativo ao modelo tradicional e o modelo ágil, temos requisitos que eram desenvolvidos totalmente no início do processo de desenvolvimento e todo o restante do processo era

baseado nesses requisitos. Normalmente, eram feitas baselines sobre esses requisitos, os quais não poderiam ser alterados sem uma solicitação de mudança, a qual passaria por um comitê de aprovação. Novamente, esses requisitos não eram flexíveis.



Já no modelo ágil, a cada incremento ou iteração se espera que tenha uma versão do software funcionando e pronto para ser utilizado. Esse modelo permite uma resposta mais rápida para mudanças e suporta o modelo Moderno de ALM, chamado de *adaptive planning*, com foco em entregar o máximo valor possível ao negócio. Além disso, os requisitos são desenvolvidos com o feedback contínuo dos *stakeholders*, tornando-os assim muito mais consistentes em cada iteração.

Com grande foco em colaboração, usuários (*stakeholders*), desenvolvedores, testers, operações etc., todo o time trabalha em conjunto para definir os requisitos e gerenciar as demandas. Nem todo requisito será inteiramente definido no início do projeto, assim

o time trabalha em requisitos menores e com mais valor imediato ao usuário, e continua definindo melhor os requisitos para as próximas iterações de acordo com os feedbacks.



- **Pontos importantes:**

- Os requisitos devem ser modelados, detalhados, analisados e revisados durante todo o projeto.
- Os requisitos podem ser alterados a qualquer momento para que o usuário possa tirar o maior proveito do desenvolvimento.
- É extremamente necessário gerenciá-los de acordo com as prioridades definidas com o stakeholder.
- Os requisitos possuem níveis de detalhes diferentes (refinamento progressivo).
- Criar modelagens que facilitam o entendimento e colaboração de todos os stakeholders (User Stories, Storyboarding etc.)

User Stories

Uma das formas de trabalhar com planejamento ágil usando o TFS (Team Foundation Server) é utilizar *User Stories*. O objetivo principal é criar uma simples e breve descrição do que é necessário para o cliente ter no produto, representando uma necessidade do

usuário ou descrição de uma funcionalidade do software.

Para modelar os papéis de usuários, algumas premissas devem ser levadas em consideração:

- Evitar usuário genérico;
- Levar em consideração diferentes perspectivas e objetivos;
- Descobrir novas User Stories.

Contribuinte	
Descrição	Expectativas
Contribuinte é o sujeito passivo de uma obrigação tributária. Em outros termos, é aquele que se sujeita, por previsão legal, ao pagamento de tributos ao fisco.	- Preencher e submeter a declaração anual de ajuste pela internet, através de um sistema com boa usabilidade. - Agilidade para recebimento da restituição.

Um exemplo de padrão para ser usado na descrição de User Stories é:

Formato:

Descrição:

Como um <X> eu posso/gostaria/devo <Y> para <Z>

X = Usuário, Sistema (Persona)

Y = Necessidade de cliente/usuário

Z = Valor ao negócio

Critério de aceite:

Dado <A> quando então <C>

A = Contexto ou condição

B= Evento ou ação do usuário

C = Objetivo

Utilização:

Exemplo:

Título: Contribuinte verifica as pendências na declaração de IR

Descrição:

▪ **Como** um Contribuinte **eu gostaria** de verificar se há incorreções no preenchimento da minha declaração **para** poder ajustar e fazer a gravação dos dados corretamente

▪ Critério de Aceitação:

▪ **Dado** que a declaração está sendo preenchida **quando** o contribuinte escolher a opção Verificar Pendências **então** deverá aparecer a opção com as incorreções da declaração

▪ **Dado** que o Contribuinte está visualizando a lista de incorreções **quando** clicar em uma incorreção **então** será exibido o campo a ser corrigido

Algumas precauções devem ser tomadas para não criar User Stories de forma incorreta:

- Não devem ser muito pequenas ou muito grandes;
- User Stories não devem conter interdependências;
- Muitos detalhes;
- Não possuir valor de negócio associado;
- Possuir especificação técnica ou interface de usuário muito cedo;

- Não orientada a um objetivo.

Definição de Done e definição de Ready

Em comum acordo com o time, a User Story pode ser considerada concluída quando estiver codificada, testada, integrada, aprovada etc. Não somente no nível de User Story mas a “definição de pronto” pode ser aplicada em outros níveis como Sprint, Release, Feature, Epics. Com isso ao finalizar cada uma das fases ou entregas do projeto, os critérios de Definition of Done são validados para concluir cada etapa, e se algum critério não foi implementado ou executado, a entrega falhou.

Além da Definition of Done, outra forma de dar continuidade ao andamento do projeto é a Definition of Ready. Diferente da primeira prática, aqui é um acordo entre os envolvidos para definir algo que está pronto para ser iniciado e pronto para a próxima fase.

3.2 A FÁBRICA ESTÁ ATRASANDO TODAS AS ENTREGAS, O QUE POSSO FAZER PARA MELHORAR ISSO?

Por Ricardo de Almeida

“Contratei uma fábrica de software para agilizar o meu processo de desenvolvimento de software, porém, os cronogramas não têm sido cumpridos e eu não tenho visibilidade do que está sendo feito pela fábrica.”

A contratação de uma fábrica de software tradicionalmente ocorre quando não se possui um time de software interno, ou quando não é possível dar vazão ao desenvolvimento de todas as necessidades do negócio com o time interno. Ao se contratar uma fábrica de software, usualmente tem-se dois modelos de trabalho

estabelecidos:

1. Time da fábrica fica alocado no espaço físico da contratante, usando seus recursos de rede e infraestrutura.
2. Time da fábrica fica alocado em espaço próprio, utilizando pouco ou nenhum recurso de rede ou infraestrutura da contratante.

Para ambos os modelos, algumas preocupações relacionadas ao gerenciamento de propriedade intelectual e ao gerenciamento dos projetos são recorrentes: manter o time da fábrica de software em seu espaço físico é bastante custoso além de abrir o acesso de terceiros para informações do seu negócio como um todo. Em contrapartida, ter o time da fábrica em um ambiente não controlado também pode gerar problemas, pois, além de diminuir o monitoramento em relação ao que está sendo feito no projeto, é também mais difícil proteger a propriedade intelectual de quem contratou o serviço de desenvolvimento de software.

No final das contas, não existe uma receita mágica para um modelo de contratação de fábricas de software; esta decisão sempre vai depender das necessidades de cada contratante. Quando falamos de propriedade intelectual, a única maneira de se resguardar é por meio de contratos muito bem formulados pela área jurídica. Já em relação à supervisão de atividades e andamento do projeto, existem ferramentas que podem ajudar fortemente a centralizar todas as suas atividades.

Com uso do Team Foundation Server ou Visual Studio Online, por exemplo, é possível integrar o controle de Portfólio e Projetos no Project Server, o controle e compartilhamento de arquivos e artefatos do projeto com o SharePoint Server, o controle de códigos fontes com GitHub ou TFSSCM, a compilação e armazenagem de binários e símbolos com o TFS Build, a automação de Release e

Deploy com o Release Management, o provisionamento de estações de trabalho para desenvolvimento, testes, homologação e produção com Azure ou System Center, além do controle de requisitos e atividades no próprio TFS. Tudo isso é feito de forma totalmente integrada e segmentada garantindo a avaliação dos projetos de software, tanto com fábricas trabalhando na infraestrutura da contratante quanto de forma remota on-premises ou na nuvem, porém, ainda utilizando uma infraestrutura de desenvolvimento de software controlada pela contratante.

A maior parte dos atrasos em projetos de software se dá por entendimento incompleto dos requisitos por parte dos desenvolvedores, requisitos subestimados durante a fase de planejamento ou mudanças de escopo durante os ciclos de desenvolvimento.

As seguintes práticas podem ajudar na resolução desses problemas:

- Criação de requisitos menores, focando individualmente cada funcionalidade.
- No processo de escrita dos requisitos, utilizar exemplos práticos para o fluxo principal e fluxos alternativos de cada funcionalidade.
- Criar uma definição de pronto para cada requisito, ou seja, definir o que deve estar funcionando no software para que o requisito seja considerado finalizado.
- Criação de protótipos para facilitar o entendimento do time de desenvolvimento e garantir o alinhamento com o usuário que está solicitando a funcionalidade.
- Atividades de testes integradas às atividades de desenvolvimento para garantir que os requisitos estão sendo implementados corretamente durante o ciclo de desenvolvimento do software, não somente ao seu final.

- Planejamento de testes alinhados aos requisitos e à definição de pronto.
- Ciclos de desenvolvimento de software menores com entrega de software funcionando ao final de cada ciclo. Incluir também a apresentação e validação do software funcionando com o usuário final ou responsável pelo software.
- Manutenção de backlog de requisitos “vivo”, ou seja, antes do final de cada ciclo de desenvolvimento, analisar, refinar e priorizar os requisitos que deverão ser trabalhados no próximo ciclo.

Estas práticas remetem ao desenvolvimento de software ágil:

1. **Indivíduos e interação entre eles** mais que processos e ferramentas
2. **SOFTWARE EM FUNCIONAMENTO** mais que documentação abrangente
3. **Colaboração com o cliente** mais que negociação de contratos
4. **Responder a mudanças** mais que seguir um plano

Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.

Equilibrar questões de custo e velocidade no desenvolvimento de software quando se trabalha com times terceirizados é uma das grandes dores de cabeça de quem contrata times externos para desenvolver uma nova aplicação ou customizar aplicações existentes. Questões contratuais para resguardar o que deve ser entregue pela fábrica, considerando o que foi planejado na fase de

levantamento de requisitos, resguardam a contratante por um lado e a prende em relação a mudanças e melhorias por outro.

O ideal nestes casos é possuir um acordo de entrega contínua de software, pois tendo isto negociado com o time terceirizado, quando mudanças se fizerem necessárias, a contratante terá recebido apenas parte do que precisava ser feito, e não um software inteiro onde muitas vezes será necessário recomeçar o desenvolvimento do zero para se melhorar ou mudar o que é exigido pelo negócio.

O desenvolvimento de software essencialmente conta com três pilares: Pessoas, Processos e Ferramentas. Quando terceirizamos a construção ou manutenção de uma aplicação, também terceirizamos o mais importante dentre estes três pilares, ou seja, as pessoas. E com o gerenciamento direto de um time de desenvolvimento de software, é possível adquirir feedbacks mais consistentes e em tempos menores. Isto ajuda os gestores a terem uma melhor ideia da evolução dos projetos de software.

Porém, a única maneira verdadeiramente consistente de se avaliar o progresso de desenvolvimento de uma aplicação é inspecionando regularmente partes do software, ou seja, com entregas contínuas de software funcionando. Para se atingir este estágio, podemos contar com ferramentas e processos que estabeleçam a entrega contínua de software funcionando.

Processos

Estabelecer um processo de desenvolvimento ágil com a fábrica de software ajuda a contratante do serviço de desenvolvimento a criar uma cultura de entrega contínua das aplicações. Desta maneira, ao final de um ciclo de desenvolvimento (fase de entrega de um módulo do sistema ou final do contrato), a contratante já terá avaliado e testado partes do software, e terá o controle em tempo real do que ainda falta para que a aplicação seja totalmente

finalizada.

Quando se adota um processo de desenvolvimento ágil, a contratante pode identificar qual parte do que foi levantado na fase de planejamento do projeto já atenderá as necessidades do negócio e focar os esforços em outras áreas ou recursos para o mesmo software. Outra vantagem é a possibilidade de disponibilizar partes do software em produção antes mesmo que ele esteja completamente desenvolvido.

Vamos a um exemplo. Vamos supor que contratamos uma fábrica de software para o desenvolvimento de um sistema que venderá ingressos online para um evento. Este sistema terá os seguintes módulos:

1. Módulo de cadastro para pré-venda dos ingressos;
2. Módulo de venda de ingressos;
3. Módulo de relatórios financeiros.

Também vamos supor que temos as seguintes datas fixadas pela área de negócios:

1. Período de inscrição para pré-venda de 01/01/2020 até 31/01/2020;
2. Venda de ingressos de 15/02/2020 até 15/03/2020;
3. Entrega do balanço financeiro até 31/03/2020.

Para atender esta demanda, é possível fixar com a fábrica entregas contínuas do módulo de inscrição para pré-venda para que se validem as funcionalidades, façam-se os devidos ajustes de funcionalidade, testes de carga para criação de uma infraestrutura que suporte a demanda esperada e fixar uma data alvo para que este módulo esteja disponível para o público em 01/01/2020.

Desta maneira, a contratante do software poderá gerenciar possíveis ajustes que se façam necessários conforme as partes

entregues do sistema forem testadas em vez de receber um módulo inteiro a uma semana da data da pré-venda e, somente aí, descobrir que existem funcionalidades desnecessárias ou que a navegação para o usuário será muito complicada.

Outro ganho será a possibilidade de medir questões relacionas à performance da aplicação muito antes de ela ser disponibilizada em produção. Isso ajudará no provisionamento de uma infraestrutura para atender a demanda e diminuir o risco de uma experiência ruim para os usuários finais ou gastos desnecessários com infraestrutura. Para os demais módulos, segue-se o mesmo processo.

Com entregas contínuas, a contratante dos serviços de software pode, muito mais que acompanhar o progresso do software desenvolvido pela fábrica, priorizar o que deverá ser feito no próximo ciclo de desenvolvimento e testar cada pequena funcionalidade do software antes que ele seja liberado em produção. Imagine que a fábrica de software não tenha finalizado o desenvolvimento da validação de CEP até a data exigida para que o módulo de inscrição para pré-venda esteja aplicado em produção. Isto não impedirá que o módulo seja colocado em produção se todo o restante já tiver sido testado pela contratante e estiver funcional. Ainda neste cenário, com entregas constantes de software, é possível acrescentar a validação de CEP ao módulo de pré-venda dias ou semanas após o mesmo ser liberado em produção, sem afetar o período de pré-venda.

Ferramentas

O uso correto de ferramentas para gerenciar o ciclo de vida das aplicações ajuda a evitar dependências com terceiros, e a otimizar todas as fases relacionadas à construção de um software. Com o uso de ferramentas, podemos controlar onde os códigos-fonte são armazenados, controlar o andamento dos requisitos dos projetos,

gerenciar a compilação e geração de pacotes das aplicações, definir como seus softwares serão liberados e instalados e, por fim, como as aplicações serão monitoradas.

- **Repositório de código-fonte:**

O artefato mais importante de um software tanto para fins de deploy quanto para fins de *troubleshooting* é o código-fonte. Definir um tipo de repositório para código é o primeiro passo para melhorar a integração com as fábricas de software e melhorar a liberação das aplicações. Existem repositórios centralizados, onde o controle dos códigos-fontes fica em uma infraestrutura totalmente controlado pela contratante, e os repositórios distribuídos onde a contratada pode ter um repositório em sua própria infraestrutura integrando com o repositório da contratante - neste cenário, o controle do repositório fica distribuído. O TFS a partir da versão 2013 possibilita o uso do TFS-GIT (Repositório distribuído) e do TFSSCV (Repositório centralizado) dentro da mesma solução de ALM.

- **Planejamento de projetos e requisitos:**

É possível gerenciar projetos e requisitos de forma integrada com o desenvolvimento do software. Utilizando o TFS, por exemplo, é possível associar todos os códigos-fontes aos requisitos e atividades. Isso ajuda os contratantes de fábricas de software a acompanharem a evolução das aplicações e terem rastreabilidade dos artefatos que são gerados a cada ciclo do desenvolvimento. As contratantes dos serviços de desenvolvimento de software também podem contar com a integração do TFS com o Project Server para gerenciar os projetos em alto nível de forma totalmente

integrada com a evolução dos requisitos e atividades de desenvolvimento

- **Build:**

A compilação e empacotamento das aplicações são essenciais para garantir a liberação das aplicações nos ambientes de testes, homologação e produção e isso se dá através do processo de Build. Possuir um servidor de Build ajuda as contratantes de fábricas de software a manter um repositório com todas as versões do software facilitando as publicações e mudanças de versões, automação da compilação e criação de pacotes das aplicações, gerenciamento de dependência tirando da máquina de desenvolvedores específicos todos os artefatos necessários para geração dos pacotes das aplicações e centralizando em ambientes gerenciados pelas contratantes da fábrica. Também é possível medir automaticamente a qualidade dos códigos-fontes gerados pela fábrica de software a cada ciclo de desenvolvimento através do TFS Build com medição da cobertura de testes e execução do Code Analysis por exemplo.

- **Release e Deploy:**

Automatizar a liberação e a troca de versões das aplicações é essencial para garantir a continuidade dos negócios dependentes de software. Integrar o Release e Deploy com uma ferramenta de Build garante maior agilidade nesse processo e protege as contratantes de fabricadas de software da dependência de terceiros nesse processo que costuma ser um dos mais críticos para qualquer negócio. Com o uso da ferramenta Microsoft Release Management, por exemplo, é

possível integrar o processo de Release e Deploy com o TFS Build e automatizar as políticas de gestão de mudanças e instalação das aplicações em ambientes de teste, homologação e produção.

Ter um conjunto de processos e ferramentas integrado e flexível que possibilita a entrega contínua de software, gerenciamento de artefatos dos softwares, controle integrado das atividades dos projetos, gerenciamento dos processos técnicos e de liberação das aplicações é fundamental para integrar os processos e necessidades de quem contratou o desenvolvimento de software com quem desenvolveu o software externa ou internamente.

Mudanças sempre vão ocorrer no processo de construção de aplicações, porém, centralizar o processo de desenvolvimento e integrar todas as fases do projeto em ferramentas e processos que permitam agilidade nesse processo é a melhor saída para que se tenha software funcionando em ciclos menores e como resultado menos dependência para o negócio de quem contrata fábricas de software, maior qualidade nas aplicações, maior visibilidade do que está sendo feito pela fábrica, maior controle em relação ao que deve ser priorizado durante cada ciclo de desenvolvimento e o mais importante: Software em funcionamento.

REFERÊNCIAS

Ferramenta para gestão de ciclo de vida de aplicações - <https://msdn.microsoft.com/en-us/library/vs/alm/tfs/overview>

Manifesto Ágil - <http://www.manifestoagil.com.br>

3.3 PLANEJAMENTO DE PROJETO GUIADO A FEEDBACK

FEEDBACK

Por Ricardo de Almeida

“Sempre que entrego um módulo do meu software para o usuário final passo o dobro do tempo fazendo ajustes e alterações que caracterizam mudanças no escopo do projeto. Os meus usuários nunca estão satisfeitos com as entregas finais, nem aceitam o que foi acordado na fase de levantamento de requisitos.”

Frequentemente, os times de software guiam suas atividades baseando-se em cronogramas, requisitos, backlogs, entre outros. Porém, esquecem-se do guia mais efetivo: o *feedback dos clientes*, ou seja, as reais necessidades daqueles que de fato vão utilizar o software.

“Eu sei, eu sei... O usuário nunca sabe o que quer, ele toda hora pede para alterar algo” Mas, e você desenvolvedor? Você gostaria que os softwares que você utiliza refletissem as suas reais necessidades? Ou quando você veste o “chapeuzinho” do cliente, você sempre sabe o quer e nunca pede para melhorar algo?

Sendo assim, no seu time, o que acontece com os feedbacks enviados aos desenvolvedores?

O planejamento de projetos guiados a feedbacks é uma quebra de paradigma que envolve fortemente as duas partes que compõe um projeto de software: os times de desenvolvimento de software e os **contratantes do projeto**.

- **Times de Desenvolvimento de software:** o modelo tradicional de desenvolvimento de software não permite uma participação mais efetiva do cliente e usuário final durante todo o ciclo de desenvolvimento do software. Feedbacks colhidos e tratados em espaços de tempo menores e baseados em software funcionando

permitem que o software nasça e se desenvolva de forma muito mais assertiva e eficiente, gerando resultados que se alinhem ao negócio independente das mudanças que poderão e ocorrerão no curso de todo e qualquer projeto de software.

- **Contratantes de desenvolvimento de software:** quem contrata o desenvolvimento de um software precisa estar disposto a aceitar que a grande vantagem do uso de um software próprio está ligada à possibilidade de adequá-lo totalmente as necessidades do negócio. Da mesma maneira que pessoas e empresas compartilham necessidades comuns como: leis, frameworks de mercado e estilos de vida, elas possuem também necessidades específicas como: segurança, estratégias de negócios e receitas secretas. Visto isso, é possível balancear na contratação de um time de desenvolvimento de software o que pode ser flexível ou não e, dessa maneira, abrir espaço ao empirismo e evolução do software baseado na experiência de uso do usuário final. O ganho de eficiência operacional proveniente do uso de um software será totalmente definido em seu processo de criação, assim, prover e permitir que os feedbacks sejam tratados durante o ciclo de desenvolvimento do software custará muito menos do que uma operação prejudicada por um software difícil de ser utilizado ou com falhas que o desalinhem do negócio.

É preciso entender que as necessidades de qualquer negócio, seja ele profissional ou não, são voláteis. A globalização e o acesso em massa à internet trazem mais informação e a necessidade constante de mudanças e adaptações que vão desde indivíduos comuns até governos e multinacionais. A partir do momento que os times de

software aceitam que mudanças são bem-vindas, a transição para este novo mundo passa a ser muito mais fácil.

Para que o desenvolvimento de software se adapte a essa realidade é preciso ter um processo de desenvolvimento ágil que considere os seguintes pontos:

MANIFESTO ÁGIL

1. **Indivíduos e interação entre eles** mais que processos e ferramentas.
2. **Software em funcionamento** mais que documentação abrangente.
3. **COLABORAÇÃO COM O CLIENTE** mais que negociação de contratos.
4. **Responder a mudanças** mais que seguir um plano.

Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.

Um passo importante para a adoção de planejamento guiado a feedback é a construção ou adesão de um modelo de trabalho com ciclos pequenos de entrega de software funcionando. Dessa maneira, a produção de feedbacks será constante e o alinhamento com o usuário estará sempre afiado.

Existem ferramentas que facilitam a coleta de feedbacks, porém estar sempre em contato com o cliente, fazer apresentações frequentes das novas funcionalidades diretamente aos usuários finais e contratantes dos serviços de desenvolvimento de software são as maneiras mais eficientes de coletar feedbacks. Acordar com o cliente um backlog de requisitos flexível desde o início de cada

projeto também é fundamental para que os feedbacks guiem o desenvolvimento do software.

É muito comum, mesmo em projetos de times que adotaram modelos de trabalho ágeis, que a entrega de um novo software ou funcionalidade possua prazo com início e fim definidos. Para facilitar as discussões com o cliente em relação a prazo e alterações no projeto, é muito importante separar o resultado dos feedbacks colhidos em quatro categorias:

- **Problemas** - Quanto menores os ciclos de desenvolvimento, menores são as chances de descobrirmos problemas no software em produção, pois o usuário final passará a ser parte integrante do time testando e apontando erros de funcionalidade ou incoerências com os requisitos. Sempre que isso acontecer, o feedback deve ser tratado como um problema ou bug que deve ser arrumado e não deve ser acrescido ao tempo do projeto.
- **Melhorias** - Este é o cenário em que foi entregue o que foi previamente acordado com o usuário, porém, durante a utilização, o usuário encontrou melhorias que podem ser incorporadas ao software e gerar maior valor. Feedbacks visam justamente melhorar o software, mas neste caso a melhoria vai incorrer em acréscimo de tempo para o projeto e, por isso, a inclusão desse tipo de atividade no projeto deve ser tratada com o cliente.
- **Alterações** - É comum que, com o uso do software, o usuário perceba que o que foi solicitado na fase de levantamento não é funcional ou prático no uso diário do software. Nestes casos, é comum que o feedback produzido seja a solicitação de alteração de algo previamente acordado. Este é mais um cenário que

acrescentará maior tempo de desenvolvimento ao projeto e deve ser acordado com o cliente.

- **Novas funcionalidades** - Com a entrega contínua de software em funcionamento, é comum que o usuário sinta que, na fase de levantamento de requisitos, uma ou outra funcionalidade foi esquecida ou não foi considerada. Nesses casos, o feedback vai produzir a solicitação de novas funcionalidades. A inclusão de novas funcionalidades em projetos com prazos definidos deve ser muito bem alinhada entre o time de desenvolvimento e o cliente, visto que isso necessitará de mais tempo de desenvolvimento.

O planejamento de projetos guiado a feedbacks também possibilita uma visão preliminar do comportamento da plataforma onde o software será executado. Normalmente, o usuário final avalia requisitos funcionais do software e, somente após ele ser implantando em produção, é que se percebe questões relacionadas à performance que podem impactar ou inviabilizar a disponibilização do software em produção. Colher feedbacks relacionados a requisitos não funcionais como performance, tempo de resposta e visualização das aplicações em dispositivos mais antigos é determinante para que a experiência do usuário final seja completa e o sucesso do projeto garantido.

REFERÊNCIAS

Microsoft	Feedback	Client	-
https://msdn.microsoft.com/library/vs/alm/work/connect/give-feedback			

3.4 COMO GERENCIAR A ENTREGA DE SOFTWARE POR FÁBRICAS DE SOFTWARE UTILIZANDO O TFS GIT?

Por Alexandre Campos Silva

Git é um sistema de controle de versão que foi criado para controlar o código-fonte do Linux, rapidamente ganhou força nas comunidades de desenvolvimento de software "open source", se tornando um padrão. Em outubro de 2012, a Microsoft anunciou o suporte à Git no Team Foundation Server e no Visual Studio Online, o que foi adicionado já na versão 2012 do Visual Studio e vem se aperfeiçoando desde então.

Esta seção se destina principalmente ao desenvolvedor de software corporativo, que não desenvolve softwares no dia a dia em projetos, mas no ambiente corporativo. Mais especificamente vamos explorar um cenário onde o desenvolvimento, ou parte dele, é realizado por um ente externo: a fábrica de software.

Terceirização de desenvolvimento

Ao longo das próximas páginas, será demonstrado como utilizar as práticas e conceitos do mundo de software de código aberto para preencher uma lacuna muito importante em muitos ambientes de desenvolvimento corporativo: a gestão de um ativo importantíssimo, o código-fonte.

É muito comum no meio corporativo o desenvolvimento de software ser total ou parcialmente terceirizado. Esta terceirização pode ocorrer de diversas formas, existem alguns nomes na indústria para estas formas, como *bodyshopping*, *outsourcing*, fábrica de software, *staff augmentation*. Não é objetivo desta seção descrever ou equalizar estes modelos comerciais, mas, para fins didáticos, será

necessário categorizá-los quanto a alguns critérios:

1. Local do desenvolvimento

- Desenvolvimento interno: o desenvolvimento é realizado nas dependências físicas/virtuais do contratante. Neste modelo de desenvolvimento terceirizado, o código-fonte não deixa o controle de versão, e o desenvolvedor terceirizado trabalha conectado diretamente no ambiente do contratante, seja fisicamente presente, seja pelo uso de soluções de conectividade, como VPNs.
- Desenvolvimento externo: nesta modalidade, o fornecedor detém o controle do ambiente de desenvolvimento, pode retirar o código-fonte no início do projeto ou engajamento e, a cada entrega ou ao final do projeto ou do engajamento, o código-fonte é devolvido ao contratante com as devidas implementações.

2. Proprietário do software

- Software licenciado: muitos fornecedores possuem soluções ou produtos, como um ERP, e licenciam este produto. Na maioria dos cenários corporativos, estes produtos não atendem 100% das necessidades do contratante, sendo necessário realizar customizações e adaptações às necessidades de negócio específicas. Porém, em alguns destes cenários, estas adaptações são realizadas no próprio produto que está sendo licenciado, e o direito de propriedade do software não é transferido ao contratante.
- Software proprietário: existe a possibilidade da contratação de um fornecedor externo para o desenvolvimento de software cuja propriedade será do contratante. Neste cenário, o software que está sendo

desenvolvido é um ativo do contratante.

As técnicas apontadas aqui se concentram no cenário de Desenvolvimento Externo de Software Proprietário.

Controle de versão distribuído

Antes de se aprofundar no uso do Git em ambientes corporativos, é necessário entender o que o faz ser diferente dos outros sistemas de controle do código populares, como Team Foundation Version Control (TFVC), SVN, RTC, ou até mesmo o antigo Source Safe.

Tradicionalmente, os gerenciadores de código-fonte foram usados para centralizar o código-fonte criado pelos diversos desenvolvedores que nele trabalham ao longo do tempo. A abordagem de todos eles consistia em ter um repositório centralizado na rede corporativa, onde todos os desenvolvedores tivessem acesso e trabalhariam conectados compartilhando suas alterações com os outros colaboradores.

Esta abordagem é conhecida como um sistema de controle de versão centralizado (CVCS), e funciona muito bem para a maioria dos cenários de desenvolvimento de software corporativo, onde todos os desenvolvedores estão *in loco* e trabalhando na mesma *code base*.

Porém, com a popularização dos sistemas de código aberto, a necessidade por uma abordagem diferente ganhou força. Os sistemas de código aberto têm duas características importantes para este contexto:

1. Os desenvolvedores estão geograficamente dispersos;
2. Cada desenvolvedor tem a possibilidade de fazer um *fork* do repositório original e desenvolver novas funcionalidades

criando um novo projeto, e eventualmente estas novas funcionalidades podem ser reintegradas ao projeto original.

Para atender a estas demandas, foi criada uma nova categoria de gerenciadores de código-fonte: os sistemas de controle de código distribuídos (DVCS), dos quais o Git é o principal representante, abrigando dois dos principais softwares de código aberto, o kernel do Linux e o Android.

Para entender melhor como um DVCS funciona, é importante compará-lo com um sistema centralizado. Para efeito didático, a comparação será realizada entre o Git e o TFVC, mas muitas destas questões funcionam de forma equivalente nos sistemas representados por eles.

A primeira diferença fundamental é em relação ao que um desenvolvedor obtém do servidor para começar a trabalhar. Enquanto no TFVC o desenvolvedor faz um "Get Latest Version", no Git o desenvolvedor clona o repositório. Ou seja, no TFVC o desenvolvedor traz para sua máquina apenas a última versão de cada um dos arquivos que vai trabalhar e, no Git há uma cópia completa de tudo o que existe no servidor.

Desta forma, no Git, todas as operações são realizadas localmente (histórico, commit, branch, merge etc.), enquanto no TFVC estas operações são realizadas sempre no servidor. Isto faz com que, ao usar o Git, o desenvolvedor não tenha absolutamente nenhuma dependência de conectividade com o servidor de código-fonte. Outra diferença importante é em relação à forma como as alterações são gravadas no servidor. No TFVC, as alterações são sequenciais, a cada *changeset* é gerado um novo identificador sequencial. Ao fazer um merge, se a sequência não for respeitada, haverá conflito.

Já no Git, a ordem não é importante para a realização dos

commits e dos merges, o que importa é apenas o conteúdo das alterações. Cada commit é identificado por um hash que representa o conteúdo do repositório ao realizar aquele commit. Desta forma, as operações de merge podem ser realizadas independente da ordem, e até mesmo da origem de onde ela foi feita originalmente. Um conflito só será deflagrado caso ocorra alterações conflitantes no mesmo arquivo.

Um terceiro ponto que é importante destacar para entender por que o Git pode ser mais interessante para ser utilizado em um ambiente de terceirização do desenvolvimento de software é a forma como a integração entre dois repositórios pode ser feita. Utilizando TFVC, não existe uma forma simples e eficaz de manter dois repositórios em sincronia. Existem algumas ferramentas que visam fazer isto, mas, via de regra, são operações trabalhosas e de difícil manutenção.

Já o Git foi construído sob a premissa de que é essencial possibilitar a integração entre repositório, e possui diversas funcionalidades dedicadas a fazer esta integração de forma natural e controlada, como: *pull request*, *rebase* e *push*. Desta forma, o que faz o Git mais apropriado que o TFVC no contexto da terceirização de software é:

- Não existe dependência de conectividade durante o desenvolvimento;
- Suporta alterações não sequenciais;
- Funcionalidades para integração de repositórios.

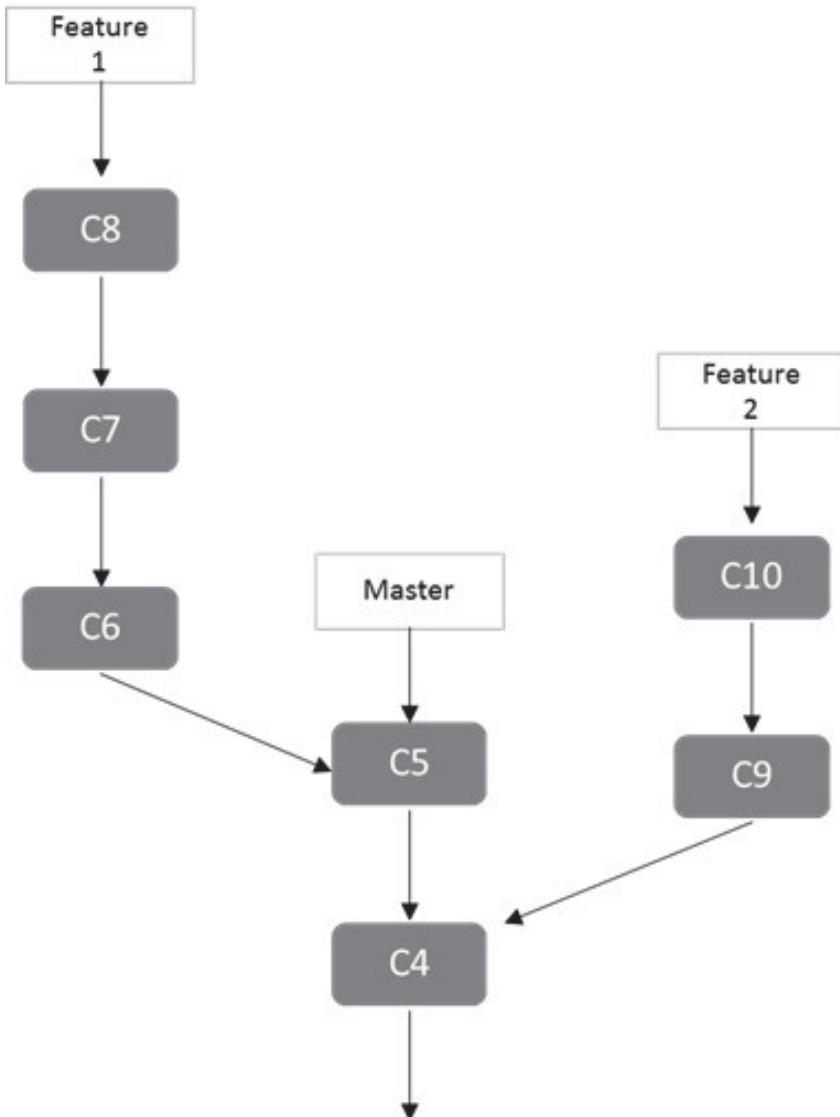
Git embaixo do capô

Para facilitar o entendimento das notações do workflow sugerido no próximo capítulo, é importante fazer uma revisão rápida de como o Git funciona. Cada commit contém, ao final, um *snapshot* de todo o conteúdo do repositório. Sobre este conteúdo, é

feito um hash utilizando o método SHA1, que gera uma sequência de 40 caracteres. Este hash é o identificador do commit e, normalmente, é identificado apenas pelos primeiros caracteres.

Adicionalmente, um commit contém também um ponteiro para o commit anterior a ele no histórico, e em alguns casos, ele pode conter dois ponteiros para os dois commits anteriores. Isto acontece quando o commit é resultado de um conflito de merge, ou merge de três pontas.

Desta forma, o histórico de um repositório Git normalmente é representado desta forma:

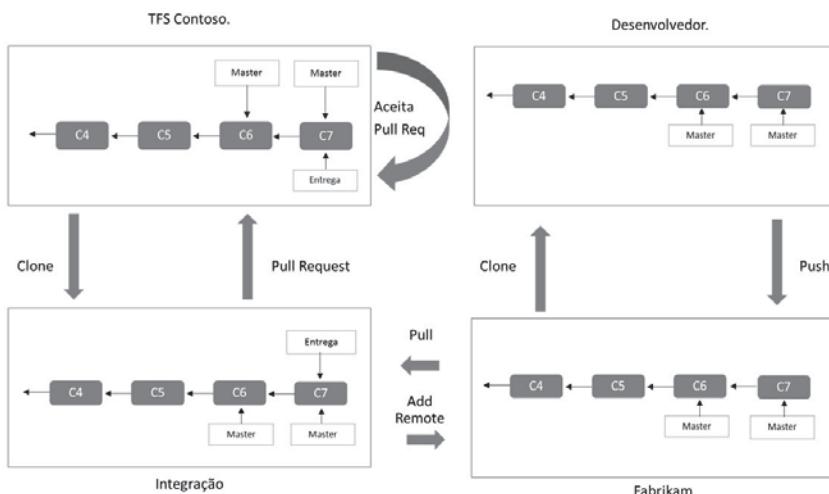


Workflow

Um workflow possível para o controle das entregas de fábrica de software no contexto definido é o seguinte:

1. Fábrica clona o repositório;
2. Fábrica cria um branch para a funcionalidade ou a entrega que será realizada;
3. Desenvolvedores clona o repositório em sua máquina;
4. Desenvolvedores fazem o commit local e o push para a origem;
5. Quando a entrega estiver pronta, a fábrica faz o rebase do branch;
6. Fábrica faz o push do branch para o fornecedor;
7. Fábrica solicita um pull request;
8. Contratante revisa as alterações realizadas e aceita o *pull request*.

Desta forma, teríamos o seguinte fluxo entre repositórios:



A seguir, será descrito cada um dos passos para se completar o workflow proposto. Para efeitos de demonstração, o contratante será chamado de Contoso, e a fábrica de Fabrikam. Neste cenário, Contoso utiliza o Visual Studio Online como seu repositório, enquanto a Fabrikam o GitHub.

Os exemplos serão feitos utilizando a linha de comando. Como a interface de usuário do Git dentro do Visual Studio está em constante evolução, e as ferramentas de linha de comando são relativamente estáveis, é mais provável que os exemplos a seguir continuem funcionando anos após a publicação deste livro.

1) Fábrica clona o repositório.

Para o exemplo a seguir, Contoso disponibiliza acesso ao seu repositório, e a Fabrikam criou um repositório no GitHub para guardar os fontes do Contoso.

```
git clone https://contoso.visualstudio.com/DefaultCollection/_git/  
Project  
git remote add github https://github.com/fabrikam/contoso.git  
git push u github master
```

2) Fábrica cria um branch para a funcionalidade ou a entrega que será realizada.

```
git branch entrega1  
git push github entrega1
```

3) Desenvolvedores cloram o repositório em sua máquina.

```
git clone https://github.com/arcs001/fork.git  
git checkout entrega1
```

4) Desenvolvedores fazem o commit no branch local e o push para o GitHub.

```
git commit  
git push origin entrega1
```

5) Quando a entrega estiver pronta, a fábrica faz o rebase do branch.

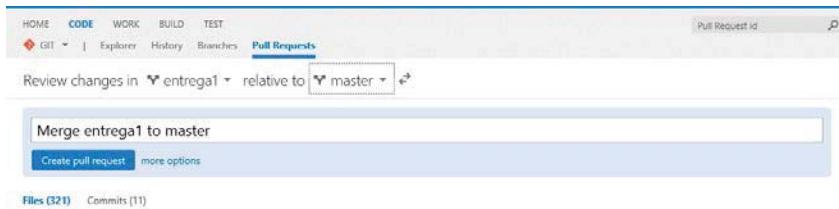
```
git checkout entrega1  
git rebase master
```

6) Fábrica faz o push do branch para o fornecedor.

```
git push origin entrega1
```

7) Fábrica solicita um pull request.

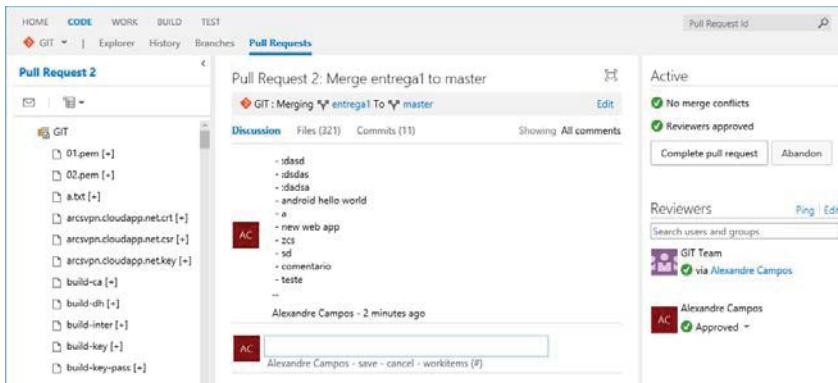
Para solicitar um pull request a partir do Visual Studio Online, acesso *Code / Pull Requests / Create Pull Request*. Selecione os branchs de origem e destino, e clique em *Create pull request*.



8) Contratante revisa as alterações realizadas e aceita o pull request.

Após a criação do pull request, a revisão da entrega se inicia por parte da Contoso. Neste momento, a Contoso pode revisar as alterações, fazer um build do novo código, realizar testes e interagir com a Fabrikam por meio de comentários na própria ferramenta.

Caso seja necessário a Fabrikam pode submeter novas alterações para resolver eventuais problemas apontados durante o processo de revisão. Após a Contoso estar satisfeita com a entrega, ela pode integrar o código em seu repositório fazendo o merge no *branch master*.



REFERÊNCIAS

Método SHA1 - [https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha1\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha1(v=vs.110).aspx)

3.5 CONCLUSÃO

Todas as técnicas apresentadas durante o capítulo podem e devem ser usadas no dia a dia para aumentar a produtividade e andamento dos projetos. São técnicas eficientes para garantir o controle de demandas em tempo de projeto ou sustentação. Aprender com os problemas encontrados demonstra a maturidade para ter sucesso.

No capítulo a seguir, são abordadas técnicas recomendadas para o dia a dia dos times de desenvolvimento – tanto para o processo de desenvolvimento de software como para a construção de novas soluções.

CAPÍTULO 4

PADRÕES DE DESENVOLVIMENTO

Aplicações consistentes que seguem padrões de desenvolvimento ampliam a qualidade do software, reduzem o *time-to-market*, promovem o trabalho colaborativo, eliminam a perda de tempo em problemas, simplificam a manutenção, e fazem uso de práticas conhecidas e já adotadas pelo mercado. Comumente, enfrentamos em campos problemas de desenvolvimento cujas soluções podem ser padronizadas e aplicadas em diferentes projetos.

Este capítulo detalha problemas encontrados mais de uma vez em diferentes projetos e de diferentes times de desenvolvimento, e as soluções adotadas para suas resoluções foram as mesmas. Os padrões de desenvolvimento introduzidos neste tópico são uma coleção de práticas para resolução de problemas de desenvolvimento e para produção de software.

4.1 POR QUE CRIAR EXCEÇÕES CUSTOMIZADAS

Por Rafael Teixeira

“Possuo alguns componentes que são utilizados por diversas aplicações. Em quais situação devo criar exceções customizadas para minhas classes do componente?”

Este é um tópico polêmico e talvez a melhor resposta para esta pergunta seja: raramente. Embora o .NET Framework nos permita criar exceções personalizadas, seguindo as boas práticas para exceções, devemos sempre utilizar as exceções já existentes no .NET antes de criar exceções personalizadas.

Para a grande maioria dos casos, as exceções no .NET nos atendem e devemos usá-las, como por exemplo, `TimeoutException`, `ArgumentException`, `InvalidOperationException` etc.

Mas vocês devem estar se perguntando: *e no caso de usar exceções para retornar informações de negócio para o método chamado, poderia então criar exceção personalizada?*

Partindo para um exemplo prático, suponha uma rotina de busca de clientes por nome, onde a exceção `ClientNotFoundException` é lançada quando nenhum cliente se enquadra nos filtros de pesquisa.

```
public List<Clientes> BuscarClientesPorNome(string nome)
{
    ExceptionDemoEntities db = new ExceptionDemoEntities();
    // Busca Clientes no repositório de dados.
    var clientes = db.Clientes.Where(p=> p.Nome.Equals(nome)).ToList();
    if (clientes.Count == 0)
        throw new ClientNotFoundException(Nome);
    return clientes;
}
```

Respondendo a esta pergunta, a classe de exceção `ClientNotFoundException` não deveria ter sido criada, e muito menos um método deve usar exceções para retornar informação de negócio. As exceções têm implicações diretas na performance da aplicação, devido à sua estrutura de gerenciamento e propagação. Então, quanto menor o número de exceções que ocorrem no seu código, maior a performance. Neste exemplo, o método deve

retornar à informação sem lançar a exceção `ClientNotFoundException`.

"E no caso de precisarmos adicionar informações na exceção? Precisamos criar uma exceção customizada?" Novamente, não. Todas as exceções do .NET derivam da classe `Exception`, que possui uma propriedade chamada `Data` do tipo `Dictionary`, que pode ser usada exatamente para estes casos, onde é necessário agregar informação para a exceção.

A seguir, temos um exemplo do método que, ao capturar a exceção, adiciona o horário antes de lançar para o método chamador.

```
public static void Connect(string hostname, int port)
{
    try
    {
        TcpClient tcp = new TcpClient();
        tcp.Connect(hostname, port);
        // ...
    }
    catch (SocketException sEx)
    {
        sEx.Data.Add("Exception Time", DateTime.Now.ToString());
        throw;
    }
}
```

"E quando devemos criar as exceções customizadas?" As exceções customizadas são recomendadas nos seguintes casos:

- Não encontramos exceção adequada no .NET;
- Temos de adicionar informações estruturadas na exceção;
- Permitir que o desenvolvedor capture a exceção para tomar uma ação.

Tomando o último exemplo anterior, vamos modificá-lo para

um caso onde será possível a criação de exceção customizada.

```
/// <summary>
///
/// </summary>
/// <param name="hostname"></param>
/// <param name="port"></param>
/// <exception cref="ServerTooBusyException" remarks="Quando reber
// está exceção, tente efetuar a conexão novamente"
public static void Connect(string hostname, int port)
{
    try
    {
        TcpClient tcp = new TcpClient();
        tcp.Connect(hostname, port);
        // ...
    }
    catch (SocketException sEx)
    {
        if (sEx.NativeErrorCode == 50000)           // Valor somen
te para exemplo
            throw new ServerTooBusyException("Servidor atingiu lim
ite máximo de conexões", sEx);
        throw;
    }
}
```

Classe da exceção customizada `ServerTooBusyException()`:

```
[Serializable]
public class ServerTooBusyException : Exception
{
    public ServerTooBusyException()
    {

        public ServerTooBusyException(string message) : base(message)
        {

            public ServerTooBusyException(string message, Exception innerE
xception) : base(message, innerException)
            {

                protected ServerTooBusyException(SerializationInfo info, Strea
mingContext context) : base(info, context)
                {
```

```
    }  
}
```

Neste caso, criamos a exceção customizada `ServerTooBusyException`, seguindo as boas práticas de exceções, sendo elas:

- Derive as exceções de `System.Exception` ou de outro tipo base de exceções;
- Faça com que o nome das exceções termine com o sufixo `Exception`;
- Faça com que as exceções sejam serializáveis;
- Implemente ao menos os quatro construtores base esperados nas exceções.

A seguir, temos o trecho de código que chama o método `Connect()` e faz o devido tratamento para a exceção de exemplo `ServerTooBusyException()`.

```
private static void ConnectToService(string hostname, int port, int retries)  
{  
    try  
    {  
        Connect("localhost", 1234);  
    }  
    catch (ServerTooBusyException)  
    {  
        if (retries == 3)  
            throw;  
        retries++;  
        ConnectToService(hostname, port, retries);  
    }  
}
```

Mas qual o melhor modo de tratar exceções? Veja o tópico a seguir e tire suas dúvidas.

REFERÊNCIAS

Best Practices for Exceptions - [https://msdn.microsoft.com/en-us/library/sehkszts\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/sehkszts(v=vs.110).aspx)

4.2 COMO TRATAR AS EXCEÇÕES

Por Rafael Teixeira

Antes de falar em *como* tratar as exceções, vamos falar de *quando* tratá-las. As exceções devem ser tratadas somente se você tem uma ação a tomar sobre ela. Na maioria dos casos, essas ações são:

- **Log** – Gravar informações do acontecimento da exceção para caso de depuração de problemas;
- **Contingência** – Se com a ocorrência da exceção existe um outro caminho a ser seguido, que possa contornar o problema ocorrido. Como no exemplo anterior da exceção customizada, a captura da exceção `ServerTooBusyException` é feita para fazer uma nova tentativa de conexão.

Agora, para falar em *como* tratar as exceções, devemos sempre considerar as seguintes regras:

- Sempre da mais específica para a menos específica. `ServerTooBusyException` é mais específica do que `Exception`, devido à sua especificidade.

```
private static void ConnectToService(string hostname, int port, int retries)
{
    try
```

```

    {
        Connect("localhost", 1234);
    }
    catch (ServerTooBusyException sEx)
    {
        DB.LogException(sEx);

        if (retries == 3)
            throw;

        retries++;
        ConnectToService(hostname, port, retries);
    }
    catch (Exception ex)
    {
        DB.LogException(ex);
    }
}

```

- Relançar a exceção utilizando o `throw`. Dessa forma o `StackTrace` é preservado, e a análise da exceção a direciona para o local correto no código onde foi lançada.

```

public static void Connect(string hostname, int port)
{
    try
    {
        TcpClient tcp = new TcpClient();
        tcp.Connect(hostname, port);
        // ...
    }
    catch (SocketException sEx)
    {
        sEx.Data.Add("Exception Time", DateTime.Now.ToUniversalTime
e().ToString());
        throw;
    }
}

```

- Caso ocorra o relançamento de uma exceção, deve-se sempre adicionar a exceção original como `InnerException` do seu contexto de execução. Desta forma, ao tratar a exceção, ou registrá-la em alguma rotina de log, teremos todo o histórico de exceções

relacionadas à execução.

```
public static void Connect(string hostname, int port)
{
    try
    {
        TcpClient tcp = new TcpClient();
        tcp.Connect(hostname, port);
        // ...
    }
    catch (SocketException sEx)
    {
        if (sEx.NativeErrorCode == 50000) // Valor somente para exemplo
            throw new ServerTooBusyException("Servidor atingiu limite máximo de conexões", sEx);
        throw;
    }
}
```

4.3 VALIDAÇÃO DE PARÂMETROS

Por Luís Henrique Demetrio

“Preciso me preocupar com a validação dos parâmetros? Quais são os benefícios dessa validação?”

No contexto da informática, os parâmetros, também muitas vezes usados como sinônimo de argumentos, são utilizados para atribuir os valores de entrada para as funções ou métodos.

Em 1979, já se sabia que, quanto antes uma falha fosse identificada, menor seria o custo de desenvolvimento do projeto (MYERS, 2015). Mesmo assim, ainda é muito comum encontrarmos aplicações que carecem de uma implementação apropriada de parâmetros.

A ausência dessa implementação provavelmente ocorre devido ao curto prazo disponível para a codificação de um projeto. Em alguns casos, esse tipo de descuido é decorrente de processos de

desenvolvimentos imaturos que não consideram o uso de testes e de ferramentas de validação no ciclo de desenvolvimento.

A validação inadequada dos parâmetros pode incapacitar o uso de uma determinada funcionalidade, ou até mesmo no encerramento indesejado da aplicação, ocasionando perdas financeiras e de credibilidade da aplicação, decorrentes da indisponibilidade dela. O problema é agravado quando o valor de um parâmetro, que não foi validado e que possui valor impróprio, é considerado correto pela aplicação. Esse tipo de erro pode demorar para ser identificado e ocasionar no aumento do prejuízo financeiro.

A falta de validação pode resultar em falhas de segurança catastróficas. Em 2013, 822 milhões de registros foram violados. Observe que esses são apenas os números conhecidos, não se sabe quantos não foram relatados, ou até mesmo os que nunca foram descobertos.

A ausência da validação somada à má implementação do tratamento de exceções pode ser desastrosa, abrindo espaço para vulnerabilidades e aumentando o esforço e tempo para descobrir onde existe a falha. Para maiores detalhes sobre o tratamento de exceções, revisite a seção *Como tratar as exceções*.

Validar os parâmetros que não aceitam valores nulos

Uma boa prática de desenvolvimento, definida como uma das regras do Microsoft Code Analysis (regra CA1062), é verificar se o valor do parâmetro de referência é nulo antes de acessar qualquer propriedade do parâmetro. Se apropriado, é necessário gerar uma exceção do tipo `System.ArgumentNullException` quando o argumento for nulo.

O fragmento de código a seguir exemplifica um método que viola a regra e outro que a satisfaz. O método “`DoNotValidate`” não

valida o conteúdo passado como parâmetro, assim torna-se propenso a possíveis erros em tempo de execução. O método “Validate” valida corretamente seu parâmetro de entrada, lançando uma exceção do tipo `System.ArgumentNullException` caso um valor nulo seja passado como parâmetro. Ao utilizar um tipo de exceção que corresponde de fato à falha identificada, direcionamos futuras análises aos possíveis erros associados com o tipo de exceção recebida.

```
public class Test
{
    // Esse método viola a regra
    public void DoNotValidate(string input)
    {
        if (input.Length != 0)
        {
            Console.WriteLine(input);
        }
    }
    // Esse método satisfaz a regra
    public void Validate(string input)
    {
        if (input == null)
            throw new ArgumentNullException(nameof(input)); //C# 6.0

        //antes do C# 6.0 era necessário executar           //th
        throw new ArgumentNullException("other");

        if (input.Length != 0)
        {
            Console.WriteLine(input);
        }
    }
}
```

Observe que, a partir da versão 6.0 da linguagem C#, não é mais necessário especificar o nome do parâmetro inválido por meio de strings ao gerar uma exceção do tipo `ArgumentNullException`. A expressão `nameof`, introduzida no C# 6.0, permite obter o nome de uma variável, um tipo ou um método. O uso de `nameof` ajuda a manter o código consistente ao renomear definições, uma vez que o compilador alterava apenas o valor da variável e mantinha

inalterado o valor do texto passado como parâmetro para a exceção `ArgumentNullException`.

Observe pelo fragmento de código a seguir que o construtor `Person` não verifica se o parâmetro de entrada `other` é nulo antes de acessar as propriedades desse objeto.

```
public class Person
{
    public string Name { get; private set; }
    public int Age { get; private set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    //O construtor viola a regra CA1062 ao acessar as propriedades
    do
        //parâmetro other sem verificar se o mesmo é nulo
    public Person(Person other)
        : this(other.Name, other.Age)
    {
    }
}
```

No exemplo de código a seguir, o objeto `other` é validado antes de ser chamado pelo construtor por meio do método `PassThroughNotNull`.

```
public class Person
{
    public string Name { get; private set; }
    public int Age { get; private set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public Person(Person other)
        : this(PassThroughNotNull(other).Name, PassThroughNotNull(
            other).Age)
    {
    }
    // Método utilizado para verificar se o parâmetro person é nulo
```

```

private static Person PassThroughNotNull(Person person)
{
    if (person == null)
        throw new ArgumentNullException("person");
    return person;
}

```

O Microsoft Visual Studio 2015 oferece a opção de habilitar e configurar a execução do Microsoft Code Analysis durante a compilação das aplicações. Pela regra CA1062, a ausência da implementação da validação de parâmetros pode ser identificada durante a compilação da aplicação.

Para habilitar e configurar a regra CA1062, é necessário realizar os seguintes passos:

- 1) Acessar as propriedades do projeto pelas teclas de atalho **ALT + ENTER**, ou pelo botão direito do mouse no projeto, e selecionar o item *propriedades*.
- 2) Clicar na aba *Code Analysis*, selecionar o item *Enable Code Analysis on Build* e clicar em *Open*.

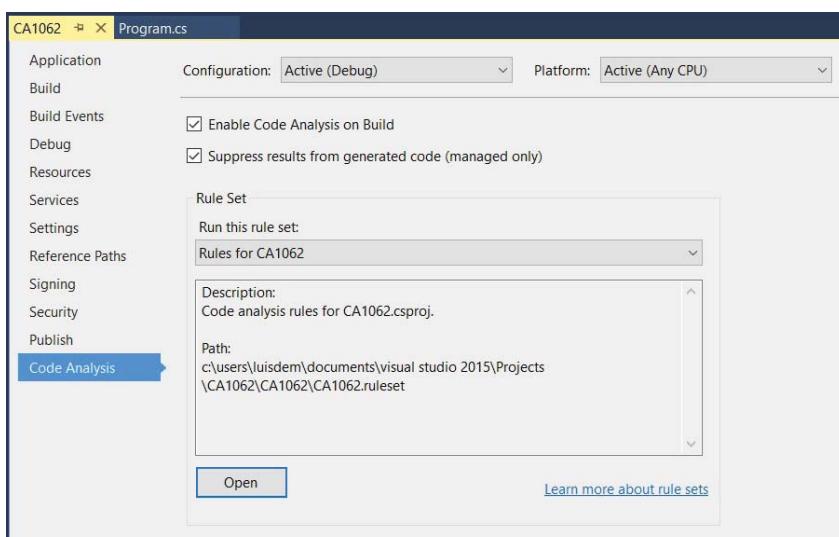


Figura 4.1: Habilitar o Code Analysis na compilação da aplicação

3) Expandir o nó *Managed Binary Analysis* e selecionar o item CA1062, conforme:

ID	Name	Action
<input type="checkbox"/> CA1058	Types should not extend certain base types	None
<input type="checkbox"/> CA1059	Members should not expose certain concrete types	None
<input checked="" type="checkbox"/> CA1060	Move P/Invokes to NativeMethods class	Warning
<input checked="" type="checkbox"/> CA1061	Do not hide base class methods	Warning
<input checked="" type="checkbox"/> CA1062	Validate arguments of public methods	Warning
<input checked="" type="checkbox"/> CA1063	Implement IDisposable correctly	Warning
<input type="checkbox"/> CA1064	Exceptions should be public	None
<input checked="" type="checkbox"/> CA1065	Do not raise exceptions in unexpected locations	Warning
<input type="checkbox"/> CA1300	Specify MessageBoxOptions	None

Figura 4.2: Habilitar a regra CA1062

Observe que, por padrão, essa regra está configurada para exibir um alerta. Caso alguma violação for encontrada, a compilação não será interrompida.

Code	Description	Project	File	Line
CA1062	In externally visible method 'Person.Person(Person)', validate parameter 'other' before using it.	CA1062	Program.cs	31

Figura 4.3: Retorno da execução do Code Analysis

É possível alterar esse comportamento ao clicar em cima do valor da ação e selecionar o valor desejado. Caso o valor for alterado para **Error** e existir alguma violação, a aplicação não será compilada decorrente ao erro gerado.

Validações de segurança

A falta de uma validação eficiente nos parâmetros de entrada pode contribuir para ataques de injeção de SQL nas aplicações (*SQL Injection*). Basicamente, esse tipo de ataque permite a execução de comandos no banco de dados decorrente de uma validação

ineficiente dos parâmetros de entrada.

Esse tipo de vulnerabilidade é muito explorado em ataques de tentativa de acesso não autorizado. A figura a seguir ilustra a atribuição de um valor no campo correspondente ao usuário, com o objetivo de modificar o comando SQL para que o resultado seja verdadeiro.

Use a local account to log in.

The screenshot shows a login interface with three fields: 'User' (containing "' or 1=1--"), 'Password' (empty), and a 'Login' button. The 'User' field has a red border, indicating an error or validation failure.

Figura 4.4: Tentativa de acesso via SQL Injection

Basicamente, essa vulnerabilidade é decorrente de uma validação de parâmetros ineficiente em conjunto com a construção dinâmica de instruções SQL sem a utilização de parâmetros com segurança de tipos.

```
string cmdText =  
    string.Format(  
        "SELECT COUNT(1) FROM [USER] WHERE Username = '{0}' AND Password='{1}'",  
        model.Username,  
        model.Password);  
  
bool userIsValid;  
  
try  
{  
    using (SqlConnection conn = new SqlConnection(ConfigurationManager.AppSettings["ConnectionString"]))  
    {  
        using (SqlCommand cmd = new SqlCommand(cmdText, conn))  
        {  
            conn.Open();  
            userIsValid = ((int)cmd.ExecuteScalar()) > 0;  
        }  
    }  
}
```

The code constructs a SQL query string by concatenating user input directly into the WHERE clause. Two sections of the code are circled in red: the WHERE clause containing the user input and the final part of the query where the user input is concatenated with the rest of the command.

Figura 4.5: Construção dinâmica de instruções SQL sem a utilização de parâmetros com segurança de tipos

Observe que o valor informado acrescenta a condição que será sempre verdadeira ($1=1$) e comenta o restante do comando SQL.

Nesse caso, não é necessário informar o usuário ou senha válidos.

Esse tipo de vulnerabilidade pode ocasionar perdas financeiras decorrentes da exposição das informações, da perda de dados, ou até mesmo da disponibilidade do próprio servidor. A figura a seguir exibe a execução de um comando que tem como finalidade liberar o endereço de IP do servidor de banco de dados com o objetivo de torná-lo inacessível.

Veja o comando usado:

```
'; exec master..xp_cmdshell 'ipconfig/release'
```

Para que esse tipo de ataque seja evitado, é necessário validar os parâmetros de entrada, evitar a concatenação de valores utilizando parâmetros para execução dos comandos, e evitar o uso de contas de banco de dados com privilégios elevados. Tomando como exemplo o código a seguir, note que deixamos de concatenar os valores diretamente na query. Além disso, passamos a utilizar os recursos provados pelo .NET Framework para adição de parâmetros aos comandos SQL. Esta prática é bastante segura e reduz a superfície de ataques por meio de injecções de SQL.

Use a local account to log in.

The screenshot shows a Windows login dialog box. The 'User' field contains the command: '; exec master..xp_cmdshell 'ipconfig/release''. The 'Password' field is empty. Below the fields is a 'Login' button.

Figura 4.6: Execução de comandos no servidor de banco de dados

Para que esse tipo de ataque seja evitado, é necessário validar os parâmetros de entrada, utilizar parâmetros para executar os comandos, e evitar o uso de contas de banco de dados com

privilégios elevados.

```
string cmdText = "SELECT COUNT(1) FROM [USER] WHERE Username = @username AND Password= @password";  
  
try  
{  
    using (SqlConnection conn = new SqlConnection(ConfigurationManager.AppSettings["ConnectionString"]))  
    {  
        using (SqlCommand cmd = new SqlCommand(cmdText, conn))  
        {  
            cmd.Parameters.AddWithValue("@username", model.User );  
            cmd.Parameters.AddWithValue("@password", model.Password);  
  
            conn.Open();  
            userIsValid = ((int)cmd.ExecuteScalar()) > 0;  
        }  
    }  
}
```

Figura 4.7: Utilização de parâmetros para executar comandos SQL

É importante registrar que a validação de parâmetros deve ser realizada em todas as camadas, inclusive nas *stored procedures*. A figura a seguir exemplifica uma stored procedure que apresenta uma vulnerabilidade de injeção de SQL por não validar os parâmetros de entrada.

```
CREATE PROCEDURE dbo.doQuery(@id varchar(128) )  
AS  
    DECLARE @query nchar(256)  
    SELECT @query = 'select ccnum from cust where id=' + @id + ''''  
  
EXEC @query
```

Figura 4.8: Stored Procedure que não valida os parâmetros de entrada

A figura seguinte exemplifica o uso correto da validação dos parâmetros:

```

CREATE PROCEDURE [dbo].[USP_SQL_INJECTION]
@FirstName VARCHAR(200),
@Lastname VARCHAR(200)
AS

BEGIN

DECLARE @SQL AS NVARCHAR (MAX)
DECLARE @PARAMLIST AS NVARCHAR (MAX)

SET @PARAMLIST =N'@FirstName VARCHAR(200),
@Lastname VARCHAR(200)'

SET @SQL = N'SELECT *
FROM Students WHERE Student_First_Name = @FirstName
AND Student_Last_Name = @Lastname'

EXEC sp_executesql @SQL, @PARAMLIST, @FirstName, @Lastname;

```

Figura 4.9: Uso correto dos parâmetros de entrada em uma stored procedure

O Microsoft Code Analysis fornece a regra CA2100, responsável por identificar os comandos SQL que apresentam vulnerabilidades.

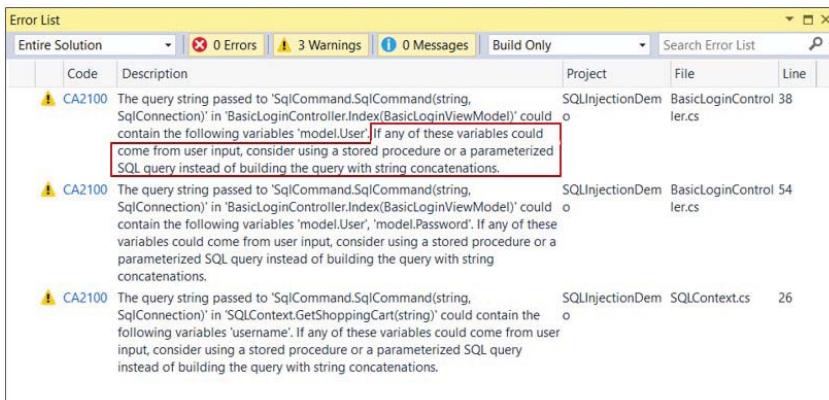


Figura 4.10: Exemplo do uso da regra CA2100 para identificar injeção de SQL 4.3.3 Validar

Transcrevendo o texto em destaque pela regra, temos claramente as ações que devem ser tomadas para resolução da

vulnerabilidade: “se alguma dessas variáveis contiver dados de entrada do usuário, considere a utilização de stored procedures ou comandos parametrizados, ao invés de construir a consulta por meio da concatenação de valores”.

Validar os valores

É muito importante validar se os valores passados para os parâmetros estão de acordo com a especificação de cada método. É importante realizar esse tipo de validação para evitar que valores inválidos comprometam o algoritmo e resultem em falhas, ou até mesmo prejuízos financeiros.

Existem diversas maneiras de validar os valores dos parâmetros de entrada. O fragmento de código a seguir exemplifica o uso de *Regular Expressions* para validar se o valor do parâmetro informado é um inteiro válido.

```
private bool IsCategoryIdValid(string categoryID)
{
    var positiveInts = new System.Text.RegularExpressions.Regex(@"^0*[1-9][0-9]*$");
    return positiveInts.IsMatch(categoryID);
}
```

O namespace `System.ComponentModel.DataAnnotations` fornece as classes de atributo que são usadas para definir os metadados para o ASP.NET MVC e controles de dados ASP.NET. O fragmento de código seguinte exemplifica o uso desses atributos.

```
public class Product
{
    public int Id { get; set; }
    [Required]
    [StringLength(10)]
    public string Name { get; set; }

    [Required]
    [Range(1, 1000)]
    public int CategoryID { get; set; }
```

```
[Required]
[RegularExpression(@"^$?\d+(\.(?\d{2}))?")]
public decimal UnitPrice { get; set; }
}
```

Observe que o atributo `Required` é usado para informar que a propriedade `Name` é obrigatória e que seu tamanho máximo se limita a 10 caracteres. Por meio do atributo `Range`, é possível observar que a faixa de valores válidos para a propriedade `CategoryID` varia de 1 a 1000. Também é utilizado *regular expressions* para restringir que a propriedade `UnitPrice` aceite apenas dois dígitos decimais. O atributo `EmailAddress` é usado para informar que o valor válido é um endereço de e-mail.

```
public class ForgotViewModel
{
    [Required]
    [Display(Name = "Email")]
    [EmailAddress]
    public string Email { get; set; }
}
```

O framework do ASP.NET MVC automaticamente gera as regras de validação que verificam se os valores informados pelos usuários estão de acordo com os definidos para o modelo. Essa validação é realizada nas páginas por meio de JavaScripts executados do lado do cliente (*client side*).

Log in.

Use a local account to log in.

The screenshot shows a login form with two required fields: 'Email' and 'Password'. The 'Email' field is empty and has a red error message: 'The Email field is required.' The 'Password' field is also empty and has a red error message: 'The Password field is required.' Below the fields is a 'Remember me?' checkbox and a 'Log in' button. At the bottom left is a link to 'Register as a new user'.

Email |
The Email field is required.

Password
The Password field is required.

Remember me?

Log in

[Register as a new user](#)

Figura 4.11: Validação do lado do cliente

É importante lembrar de que as validações devem ocorrer também no servidor para evitar que um usuário mal-intencionado possa burlar o mecanismo do lado do cliente, como por exemplo, desativando a execução de JavaScript no navegador.

O fragmento de código a seguir exemplifica a validação realizada no método executado do lado do servidor:

```
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
}
```

O objetivo desta seção foi demonstrar a importância da validação dos parâmetros. É importante lembrar de que essa validação deve ser validada em todas as camadas e métodos, e nunca assumir que os valores já foram validados anteriormente em outras camadas. Uma validação ineficiente pode gerar falhas e

indisponibilidades do sistema como também prejuízos financeiros decorrentes de valores inválidos ou de ações mal-intencionadas.

4.4 NÃO EXPONHA LISTAS EM SEU MODELO DE DADOS

Por Bruno Oliveira

*“Meu time de desenvolvimento utiliza Code Analysis para garantir a qualidade do código criado, porém constantemente é exibida a violação CA1002: **Do not expose generic lists**. Já tentamos buscar uma justificativa para adoção, mas não consigo entender que diferença faz a não utilização da classe System.Collections.Generic.List<T> ”.*

Para o entendimento desta regra, é preciso retomar alguns princípios da Orientação a Objetos, mais precisamente o *encapsulamento*. Este princípio diz que não devemos expor detalhes de implementação dos objetos, restringindo seus usuários de realizar alterações que possam deixar o objeto em um estado inválido.

Levando este princípio em conta, comprehende-se que classes não devem expor formas de alterar seu estado sem que sejam notificadas. É exatamente neste ponto que esta regra de Análise de Código (CA1002) é baseada. As violações desta regra, em geral, ocorrem em códigos similares ao apresentado no código:

```
public class Pedidos
{
    public List<Pedido> ListaPedidos { get; }
    public decimal Total { get; private set; }
}
```

Neste exemplo, usou-se a classe System.Collections.Generic.List<T> que representa uma lista

indexada de um tipo específico de objeto, que pode ser definida como uma representação genérica da classe `System.Collections.ArrayList`.

Ao expor estas listas, a classe do exemplo permitirá que seus usuários as modifiquem de forma inesperada (por exemplo, adicionar um item na lista de pedidos, sem atualizar o seu total).

Para resolver este problema, na declaração de propriedade ou retorno de métodos, o .NET Framework fornece o namespace `System.Collections.ObjectModel` que contém classes que podem ser usadas como coleções em modelo de objetos. Dentre estas classes, destacam-se:

- `Collection<T>` : classe base para coleções genéricas.
- `KeyedCollection< TKey, TItem >` : classe abstrata que representa uma lista de chaves e valores.
- `ReadOnlyCollection<T>` : classe base para coleções somente leitura.

Com a utilização destas classes, pode-se resolver o problema de exposição de listas, simplesmente alterando a utilização da classe `List<T>` para a classe `Collection<T>`, conforme representado no trecho do código a seguir.

```
public class Pedidos
{
    public Collection<Pedido> ListaPedidos { get; }
    public decimal Total { get; private set; }
}
```

Compreende-se que o código apresentado anteriormente atende as normas definidas pela regra CA1002 e ao princípio de encapsulamento. Entretanto, uma das grandes vantagens da utilização da classe `Collection<T>` está na sua capacidade de extensão. Para tal, esta classe define os seguintes membros virtuais:

- `ClearItems` : remove todos os elementos da coleção.
- `InsertItem` : adiciona um item em uma determinada posição da coleção.
- `RemoveItem` : remove um item específico da coleção.
- `SetItem` : altera um determinado item da coleção.

Estes métodos são invocados sempre que a coleção sofrer alguma alteração. O código seguinte, extraído do código-fonte do .NET Framework e publicado pela Microsoft, mostra o caso onde é alterado o valor de um item na coleção.

```
public class Collection<T>: IList<T>, IList, IReadOnlyList<T>
{
    ...
    public T this[int index] {
        get { return items[index]; }
        set {
            ...
            SetItem(index, value);
        }
    }
    ...
}
```

Com isso, recomenda-se que sempre seja considerada a utilização de subclasses que herdem destas classes bases em vez de usá-las diretamente (CWALINA; ABRAMS, 2009). Os trechos de código a seguir apresentam as alterações necessárias para que o código exemplificado neste capítulo atenda a esta recomendação.

```
public class PedidosCollection : Collection<Pedido>
{
    public event EventHandler<Pedido> AdicionarPedido;

    protected override void InsertItem(int index, Pedido item)
    {
        AdicionarPedido?.Invoke(this, item);

        base.InsertItem(index, item);
    }
}
public class Pedido { public decimal Valor { get; set; } }
```

```

}

public class Pedidos
{
    private PedidosCollection listaPedidos = new PedidosCollection
();
    public Collection<Pedido> ListaPedidos => listaPedidos;
    public decimal Total { get; private set; }

    public Pedidos()
    {
        listaPedidos.AdicionarPedido +=
            (sender, item) => Total = Total + item.Valor;
    }
}

```

Neste exemplo, foi criada a classe `PedidosCollection`, que realiza a implementação dos membros virtuais de forma que atenda aos requisitos da aplicação e altere a classe `Cliente` para o uso desta nova classe.

Recomenda-se também que a definição destas classes siga as seguintes regras:

- **Nomenclatura:**
 - Utilize o sufixo `Collection` para o nome da classe que representa uma coleção.
 - Considere a utilização do prefixo `ReadOnly` para coleções somente leitura.
- **Design:**
 - Defina propriedades que representam coleções como somente `Leitura`.
 - Utilize a classe `ReadOnlyCollection<T>` para representação de lista somente leitura.
 - Retorne sempre uma lista vazia em vez de valores nulos.

Além de fazer o uso da capacidade de extensão das classes definidas no namespace `System.Collections.ObjectModel`, a

implementação de subclasse permite a criação de método utilitários para interação com os itens da lista. No trecho de código a seguir é apresentado um exemplo onde são incluídas duas operações customizadas que permitem a execução de uma operação em lote para toda a coleção (método `FlushAll`) e o encapsulamento da adição de novos itens na lista (método `AddSource`).

```
public class TraceSourceCollection : Collection<TraceSource> {
    //optional helper method
    public void FlushAll(){
        foreach (TraceSource source in this) {
            source.FlushAll();
        }
    }
    //Another common helper
    public void AddSource(string sourceName) {
        AddSource(new TraceSource(sourceName));
    }
}
```

Este tipo de código foi amplamente aplicado pela Microsoft durante o desenvolvimento do .NET Framework e pode ser encontrado em inúmeras classes criadas pela empresa, como por exemplo:

- `System.Web.Routing.RouteCollection`
- `System.Web.ModelBinding.ModelErrorCollection`
- `System.Net.Mail.AttachmentCollection`
- `System.Activities.ActivityPropertyCollection`
- `System.IdentityModel.SecurityTokenHandlerCollection`

REFERÊNCIAS

Data Breach	QuickView	-
https://www.riskbasedsecurity.com/reports/2013-DataBreachQuickView.pdf		

Regra	CA1062	-	https://www.microsoft.com/en-us/download/details.aspx?id=19968	e
			https://msdn.microsoft.com/pt-br/library/ms182182(v=vs.140).aspx	
Expressão	<i>nameof</i>	-	https://msdn.microsoft.com/pt-br/library/dn986596.aspx	SQL Injection
				https://www.owasp.org/index.php/Top_10_2013-T10
Regular Expressions		-	https://msdn.microsoft.com/en-us/library/hs600312(v=vs.110).aspx	
System.ComponentModel.DataAnnotations		-	https://msdn.microsoft.com/pt-br/library/system.componentmodel.dataannotations(v=vs.110).aspx	
Regra		CA1002		-
			https://msdn.microsoft.com/library/ms182142.aspx	
System.Collections.Generic.List				-
			https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx	
System.Collections.ArrayList	-		https://msdn.microsoft.com/en-us/library/system.collections.arraylist(v=vs.110).aspx	
Código-fonte	do	.NET	Framework	-
			http://referencesource.microsoft.com	

4.5 PASSAGEM DE PARÂMETROS

Por Tiago Soczek

“Durante a passagem de parâmetros, qual a diferença dos

modificadores ref e out? O modificador ref se aplica para tipos por referência? Como faço para retornar mais de um valor no método?”

Passagem de parâmetros é um tópico que gera bastante confusão. É muito comum encontrar em campo desenvolvedores, mesmo os mais experientes, com dúvidas sobre o assunto. Contudo, antes de falar sobre passagem de parâmetros, é preciso entender alguns conceitos importantes do sistema de tipos do .NET: Common Type System.

Common Type System (CTS)

O Common Type System é uma das bases do .NET Framework. Ele define como os tipos das variáveis devem ser declarados, utilizados e gerenciados, além de possuir um papel importante para a integração entre linguagens do .NET.

Todos os tipos no .NET são classificados em dois grandes grupos:

- **Tipos por valor (Value Types):** constam as estruturas (`struct`) e enumerações (`enum`), que pode ser tipos nativos (`Boolean`, `Byte`, `Char`, `DateTime`, `Decimal`, `Double`, `Int16`, `Int32`, `Int64`, `SByte`, `Single`, `UInt16`, `UInt32` e `UInt64`) ou definidos pelo usuário.
- **Tipos por referência (Reference Types):** nesse grupo, constam as classes (`class`), arrays, delegates e interfaces. O principal exemplo de um tipo por referência é a classe `String`.

Há uma diferença de comportamento entre eles, e isso é fundamental para o entendimento da passagem de parâmetros. Os tipos por valor armazenam diretamente os seus dados, ou seja, ao

realizar uma cópia de uma variável, os dados serão duplicados em memória, tornando-se independentes. Assim, qualquer modificação em uma cópia não afetará a outra.

Já os tipos por referência armazenam apenas uma referência para os dados, então ao realizar uma cópia, apenas a referência será copiada. Ou seja, terão duas referências apontando para os mesmos dados em memória.

No livro *C# in Depth*, o autor Jon Skeet (2013) faz uma excelente analogia do comportamento dos tipos por valor e tipos por referência. Ele descreve da seguinte forma: imagine que você está lendo um excelente artigo e deseja compartilhar com seu amigo. Como o artigo está em papel, você deve tirar uma fotocópia e entregar a cópia a ele, assim os dois terão uma cópia do artigo, e qualquer alteração não será compartilhada. Esse é o comportamento dos tipos por valor.

Agora, imagine que o artigo é uma web page. Para compartilhar com seu amigo, você apenas enviará a URL da web page para ele, assim os dois terão acesso ao artigo. Porém, qualquer alteração nesse artigo (imagine um wiki, onde você adicionou as suas anotações) tanto você como seu amigo terão acesso às mudanças realizadas. Esse é o comportamento dos tipos por referência.

Armazenamento em memória

Em relação ao armazenamento em memória desses tipos, é muito comum encontrar a seguinte afirmação: “*Tipos por referência são armazenados na memória heap e tipos por valor na memória stack*”.

Somente a primeira parte está correta. Sobre a segunda, existem diversos cenários nos quais isso não acontece, como no exemplo a seguir:

```
public class Foo
{
    private int bar;
}
```

No caso, o `int` (tipo por valor) é um *field* de uma classe (tipo por referência) e será armazenado na memória *heap*, e não na memória *stack*. Esse assunto é muito bem detalhado nos seguintes artigos do Eric Lippert e do Jon Skeet:

- *The Truth About Value Types* (LIPPERT, 2016);
- *Memory in .NET - What goes where* (SKEET, 2014).

A forma de armazenamento dos tipos, na maioria das vezes, é irrelevante, pois se trata de um detalhe de implementação tornando o comportamento transparente para o desenvolvedor.

Passagem de parâmetros

O C# possui quatro tipos de passagem de parâmetros: por valor (padrão), por referência (modificador `ref`), parâmetros de saída (modificador `out`) e parâmetros com número variável de valores (modificador `params`).

Por valor:

Esse é o tipo padrão de passar parâmetros em C#. Ele consiste na cópia da variável para o escopo do método, assim qualquer alteração dentro do método não afetará a variável externa.

É importante não confundir a passagem de parâmetros por valor com os tipos por valor, esses são conceitos distintos. Mesmo os tipos por referência são passados por valor, por padrão.

Segue um exemplo com um parâmetro `int` (tipo por valor):

```
public void SomarUm(int valor)
{
    valor++; // Adiciona 1
```

```
}

int numeroQualquer = 0;

SomarUm(numeroQualquer);

Debug.Assert(numeroQualquer == 0); // true
```

Neste caso, a variável `numeroQualquer` não é afetada pela adição realizada no método `SomarUm` pois , ele alterou apenas a cópia. Segue exemplo com tipos por referência:

```
public class Pessoa
{
    public int Idade { get; set; }
}

public void IncrementarIdade(Pessoa pessoa)
{
    pessoa.Idade++; // Adiciona 1
    pessoa = null;
}

Pessoa pessoa = new Pessoa();

pessoa.Idade = 28;

IncrementarIdade(pessoa);

Debug.Assert(pessoa != null); // true

Debug.Assert(pessoa.Idade == 29); // true
```

Com tipos por referência, foi realizada apenas uma cópia da referência apontando para os mesmos dados. Com isso, a adição na propriedade `Idade` foi efetiva, porém, quando atribuímos `null` para a variável, isso não é refletido externamente.

Por referência:

Quando se utiliza o modificador `ref` , é usada a passagem de parâmetros por referência. Isso indica que não é feita uma cópia da

variável, mas a passagem da própria variável para o escopo do método.

O modificador `ref` deve ser informado tanto na declaração do método como nas chamadas:

```
public void SomarUm(ref int valor)
{
    valor++; // Adiciona 1
}

int numeroQualquer = 0;

SomarUm(ref numeroQualquer);

Debug.Assert(numeroQualquer == 1); // true
```

A adição realizada no método `SomarUm` afeta a variável `numeroQualquer`, pois, ao utilizar o modificador `ref`, será passada a própria variável para o método, e não mais uma cópia.

```
public void IncrementarIdade(ref Pessoa pessoa)
{
    pessoa.Idade++;
    pessoa = null;
}

Pessoa pessoa = new Pessoa();

pessoa.Idade = 28;

IncrementarIdade(ref pessoa);

Debug.Assert(pessoa == null); // true
```

Apenas foi passada a referência da variável `pessoa`, assim será possível acessar e alterar os seus valores e também atribuir um novo valor, como no caso a variável foi atribuída para `null`.

Existem alguns casos em que não é possível utilizar o modificador `ref`:

- Métodos assíncronos (modificador `async`);
- Métodos de iteração que retornam usando `yield return` e `yield break`;
- Utilizados como parâmetros opcionais.

Parâmetros de saída:

Os parâmetros por referência têm funcionalidade de entrada e saída de valores, já os parâmetros de saída, possuem apenas a funcionalidade de retornar valores. Isso é caracterizado pelas seguintes diferenças:

- Os métodos devem sempre atribuir um valor para esses parâmetros;
- Caso a variável passada já tenha sido inicializada, seu valor será ignorado e substituído.

Assim como nos parâmetros por referência, o modificador `out` também deve ser informado tanto na declaração do método como nas chamadas:

```
public void AtribuirDez(out int valor)
{
    valor = 10;
}

int numeroQualquer;

AtribuirDez(out numeroQualquer);

Debug.Assert(numeroQualquer == 10); // true
```

Um bom exemplo da utilização de parâmetros de saída é o método `int.TryParse`:

```
Console.Write("Insira um número (int): ");

string entrada = Console.ReadLine();

int numero;
```

```
if (int.TryParse(entrada, out numero))
{
    Console.WriteLine("Você digitou o número: {0}", numero);
}
else
{
    Console.WriteLine("O valor inserido ({0}) não é um número (int ) válido", entrada);
}
```

O parâmetro de saída (int – resultado da conversão) é usado em conjunto com o retorno do método (bool – se a conversão foi bem-sucedida), assim será possível retornar 2 valores em apenas uma chamada. Apesar de ser um recurso interessante, são raros os casos em que isso será realmente necessário. Mais detalhes sobre o assunto podem ser encontrados no tópico de boas práticas.

As mesmas restrições de uso que são aplicadas aos parâmetros por referência, também são aplicadas aos parâmetros de saída.

Parâmetro com número variável de valores:

Por meio do uso do modificador params , é possível permitir que um parâmetro receba uma quantidade variável de valores. Esses podem ser passados separados por vírgula e, nesse caso, será criado um array temporário com os valores, ou ainda, passar um array já existente.

Para o método, esses detalhes serão transparentes, ele sempre receberá um array. Só é permitido um parâmetro desse tipo por método, e ele deve ser o último e não pode ser opcional.

```
public int Somar(params int[] valores)
{
    return valores.Sum(v => v); // Linq é demais!
}

Somar(); // Resultado 0
```

```
Somar(1, 2, 3); // Resultado 6  
  
int[] numeros = new []{ 4, 6 };  
  
Somar(numeros); // Resultado 10
```

Vale a pena salientar que é possível chamar o método sem passar nenhum parâmetro, então é importante que o método contenha validações que prevejam a entrada de valores nulos.

Uma ótima aplicação dos parâmetros com número variável de valores, é o método `string.Format`:

```
string formato = "{0}, Preço: {1:c}, Estoque: {2}";  
  
string descricao = string.Format(formato, "Produto #10", 99.99, 100);  
  
Debug.Assert(descricao == "Produto #10, Preço: R$ 99,99, Estoque: 100");
```

Boas práticas

É importante conhecer o funcionamento completo da passagem de parâmetros no C#, pois, muitas vezes, é necessário manter código legado que foi desenvolvido abusando desses recursos. Mas são raras exceções em que será justificável a utilização dos modificadores: `ref`, `out` e `params`.

"Preciso retornar mais de um valor no método, como fazer?". A utilização dos modificadores `ref` e `out` gera complexidade, ainda mais quando vários parâmetros utilizam esses modificadores. Em poucos casos, essa complexidade é justificada, como no método `int.TryParse`.

Provavelmente, a melhor opção seja criar uma classe ou estrutura contendo os valores que serão retornados no método. Isso tornará o método mais simples e expressivo, facilitando o entendimento e a manutenção.

"Quando vale a pena utilizar o modificador params?". Somente quando for possível aproveitar a facilidade de passar os valores separados por vírgula. Se na maioria das vezes você já tiver um array com os valores, não compensa a utilização.

"Como devo validar os parâmetros?". É recomendada a validação de todos os parâmetros de um método, ainda mais se o método for público, cuja audiência é desconhecida.

Normalmente, caso algum parâmetro não atenda aos requisitos mínimos, é disparada uma exceção do tipo `ArgumentException` ou de suas derivadas, como a `ArgumentNullException`. É preferível disparar uma exceção indicando explicitamente o motivo da falha, do que enfrentar uma exceção do tipo `NullReferenceException`, que traz poucas informações e torna o troubleshooting bem árduo.

Uma boa alternativa a esse modelo de validação é utilizar Code Contracts. Code Contracts são restrições baseadas em pré-condições, pós-condições e restrições sobre o código. Todas essas restrições são validadas em tempo de execução e compilação, para verificar se o objeto têm os requisitos mínimos esperados para assumir um tipo de dados.

"Qual a quantidade máxima de parâmetros recomendada?". Não existe um número máximo, mas se um método estiver recebendo muitos parâmetros, provavelmente ele está fazendo mais do que deveria – possui muitas responsabilidades. Uma opção seria criar classes ou estruturas agrupando negocialmente esses parâmetros, e alterar o método para receber esses tipos recém-criados.

4.6 QUAL A MELHOR ESTRATÉGIA DE BRANCH PARA O MEU SISTEMA?

Por Luiz Macedo

Esta seção é uma introdução sobre estratégias de Branching e Merging para alguns dos cenários comumente utilizados. Normalmente branches são necessárias para suportar releases ou desenvolvimento paralelo. Quando um desenvolvedor cria uma branch no controle de versão, é criada uma cópia do código-fonte em um local separado de sua origem. Tudo o que é feito em uma determinada branch não afeta as outras até que o código seja reintegrado à origem, no caso, realizando o merge de código.

Para cenários mais simples, geralmente não são necessárias branches, somente a criação de builds e labels são suficientes. Uma label é como tirar uma foto do estado atual do código-fonte. No momento em que uma label é criada, é possível identificar como o código ou arquivos no controle de versão estavam. Uma label não deve ser utilizada como auditoria de código, devido ao fato de que pode ser alterada. Em um exemplo prático, ao utilizar uma label é possível recriar um build anteriormente executado a qualquer momento no futuro ou até mesmo encontrar alguma versão do código-fonte utilizado em outro build em particular. O problema mais comum ao utilizar branches é que, quanto mais branches forem criadas, maior é o custo em fazer merge entre elas e manter a velocidade no desenvolvimento paralelo.

Eu preciso realmente de branches?

“Certo, a utilização de branch deve ser considerada para o isolamento de trabalho em times paralelos e controle de funcionalidades e releases. Mas por onde eu devo começar?”.

A primeira consideração é: não crie branches a menos que elas realmente sejam necessárias para o desenvolvimento de seu produto. Inicie somente com uma branch, isso mesmo, uma! Somente utilizar a Main é uma estratégia que funciona bem,

inclusive trabalhando em conjunto com o *Release Management* para a gestão de releases.



Figura 4.12: Estratégia de Branch única – Main Only (<http://aka.ms/treasure18>)

Essa figura representa o formato de branch única utilizando label para administrar as baselines dos releases entregues. A label funciona como uma foto do momento em que o código se encontra, guardando a informação de como ele estava até o momento da criação daquela label. Como a label pode ser alterada, temos um risco com relação à auditoria daquele momento do código, mas é bem interessante para o caso de necessidade de voltar com versões anteriores do produto.

Dependendo da ferramenta usada para o controle de versão, como no caso do Team Foundation Server, é possível alterar o permissionamento do controle de versão para a administração de labels. Caso uma única branch não atenda completamente as necessidades de isolamento do time de desenvolvimento, outras estratégias podem ser adotadas.

Quais são as estratégias mais utilizadas?

O objetivo aqui é mostrar algumas das estratégias de branches mais utilizadas no dia a dia em campo.

- **Isolamento do desenvolvimento:**

Normalmente, após iniciar com uma única branch (*Main*) e a necessidade de paralelizar o desenvolvimento de uma Release futura ou para algum outro tipo de necessidade, esse modelo é a evolução

natural.

Usando esse modelo, é possível desenvolver em paralelo com a *Main* sem que as alterações realizadas sejam enviadas para a branch estável, antes de garantir que não será afetada por algum problema de integração. Ao escalar o número de branches, é de extrema importância, antes de fazer o merge com a *Main*, baixar a última versão da *Main* na branch de desenvolvimento para garantir que as alterações de desenvolvimento não vão afetar a *Main* antes de serem testadas.

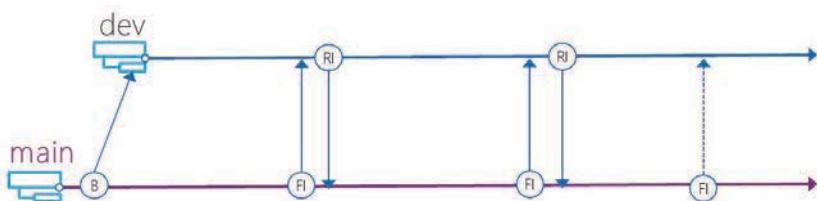


Figura 4.13: Estratégia de branch – Development Isolation (<http://aka.ms/treasure18>)

- **Isolamento das releases e desenvolvimento:**

Essa estratégia tem um nível complexo devido ao fato de que, além do desenvolvimento em paralelo, as releases também existem para permitir o controle de bug fixes e demais correções, ou emergências. O mais importante nesses casos é utilizar sempre que possível fazer Forward Integration (FI) e Reverse Integration (RI) para não ter um custo muito alto de merge ao final de cada entrega. Uma estratégia bem definida de integração continua pode ajudar a manter a qualidade dos códigos.

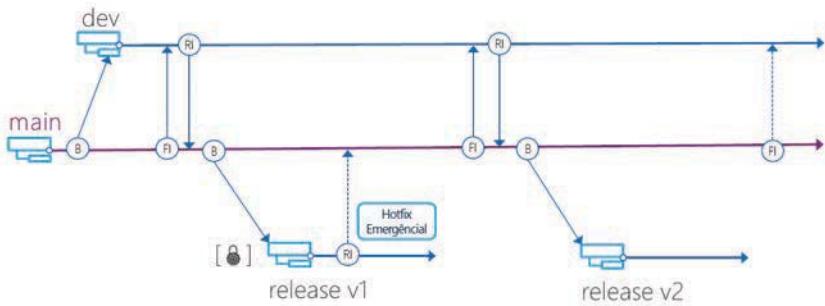


Figura 4.14: Estratégia de branch – Development and Release Isolation

- **Promoção de código:**

Com a estratégia de Code Promotion, as branches são criadas conforme as versões estão mais estáveis e caminham de um ambiente para outro. Normalmente, essa estratégia vinha com a utilização de metodologias *waterfall*, mas ainda é usada em muitos casos.

Como dito anteriormente, o custo de manutenção aumenta conforme escalamos o número de branches. O código da *Main* continua sendo o mais estável, é promovido para o ambiente de testes e assim para produção. Normalmente, era utilizado para fases prolongadas de homologação e correção em ambiente de *staging* (Ambiente de pré-produção).

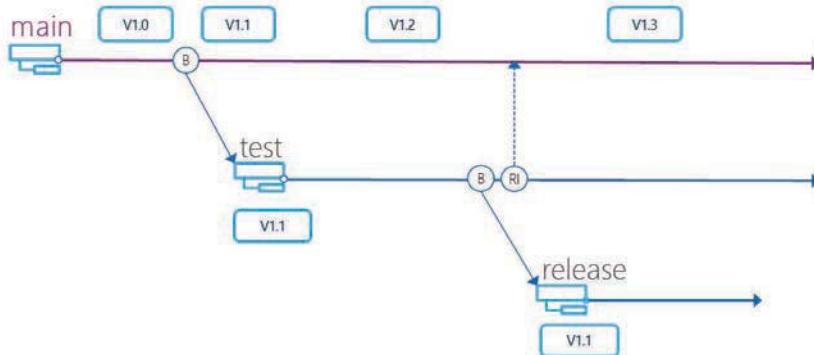


Figura 4.15: Estratégia de branch – Code Promotion

Menos é mais! Feature Toggling e entrega continua

Uma forma bem interessante de trabalhar em conjunto com a estratégia de branch única, *Main Only*, ou com algumas outras estratégias apresentadas é a técnica de *Feature Toggle*. Com ela, as funcionalidades “ligam” e “desligam” de acordo com o momento de distribuição do produto a ser entregue. Elas podem ser recursos de interface de usuário (UI) visíveis, ou não, para os usuários finais ou até mesmo uma funcionalidade de código comentada ou com uma lógica para que não seja executada no release atual, e sim na posterior.

Existem vários nomes para essa técnica que inclui *flags*, *switches* ou outras formas de implementação de código para habilitar as funcionalidades em cada build.

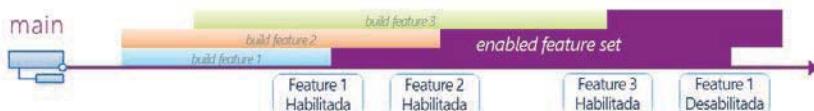


Figura 4.16: Feature Toggle – Features habilitadas e desabilitadas pela necessidade do build
(<http://aka.ms/treasure18>)

A grande vantagem de utilizar Feature Toggling é poder alterar

as funcionalidades ativas e inativas das aplicações sem ter um grande número de *feature branches* e ramificações paralelas. Esse formato de trabalho facilita até mesmo a forma compilação e distribuição das novas funcionalidades.

Fazendo entrega contínua muitas vezes, algumas features não estão totalmente completas e, dessa forma, não tem necessidade de impactar uma entrega toda somente por falta de finalização de uma funcionalidade, podendo assim somente desabilitá-la. Para fazer isso utilizando branches, normalmente será necessário criar branches separadas para manter cada uma das funcionalidades nas iterações do desenvolvimento que controlam a entrega. Isso pode aumentar a complexibilidade para fazer merges, e manter o código-fonte estável e atualizado.

Assim, o código sempre poderá ser atualizado na *Main*, pois as features estarão habilitadas somente se necessário, sem onerar o código principal. Alguns modelos foram apresentados e podem ser muito úteis, porém, podem não atender as necessidades de seu time. Elas podem variar de time para time e também podem ser muito ramificadas. Entretanto, ainda devem ser criadas somente quando o time estiver confiante de que o custo de criar branches e fazer merges é menor do que o de criar uma branch única e fazer o commit diretamente na branch principal, no caso, a *Main*.

Para que o modelo ideal seja encontrado, sempre será necessário considerar a organização do time, arquitetura do projeto etc. Isso, juntamente com as outras circunstâncias do contexto de trabalho, vai ajudar a identificar qual branch deve ser escolhida. Por fim, o objetivo é proteger ao máximo o código para que fique o mais estável possível para gerar um release confiável.

REFERÊNCIAS

Common	Type	System	Overview	-
https://msdn.microsoft.com/en-us/library/2hf02550(v=vs.100).aspx				
Common	Type	System	-	https://msdn.microsoft.com/en-us/library/zcx1eb1e(v=vs.100).aspx
Code	Contracts	-	https://msdn.microsoft.com/en-us/library/dd264808(v=vs.110).aspx	

4.7 QUALIDADE DE CÓDIGO

Por Wanderson Lima

“Tenho várias fábricas de software terceiras que desenvolvem para mim e, às vezes, trabalho com desenvolvimento interno e sustentação. Percebo que, a cada novo projeto concluído em um sistema, mais complexo e instável ele se torna. Existem aplicações que ninguém quer realizar manutenção, porque todos têm medo do impacto que uma pequena alteração pode fazer no sistema como um todo”.

Por mais estranho que pareça, esse não é um cenário isolado, incomum de acontecer na vida real. Muito pelo contrário, esse cenário acontece na maior parte dos clientes atendidos pelos Engenheiros de ALM (Application Lifecycle Management). Ao analisar com um pouco mais de profundidade a raiz do problema, vamos além da competência das pessoas que trabalham nas aplicações e chegamos a uma pergunta: qual é o maior fator de sucesso de um projeto de TI na maioria das empresas? Software

entregue (o que) ou software entregue com qualidade (o que e como)?

Assumindo a resposta incomum, ou seja, software entregue com qualidade, de que qualidade está se falando? Funcional? Performance? Quantidade de erros? Existe uma dimensão que transpassa todos esses aspectos e é capaz de afetá-los drasticamente, bem como ditar o quanto fácil e rápido será alterar uma aplicação, o quanto resiliente ela será e o quanto factível será fazer uma alteração mais profunda em sua estrutura. Esse denominador comum é a pouco lembrada e muito trivializada: qualidade de código.

São premissas básicas de um código de boa qualidade que ele:

- Respeite e faça uso das estruturas do paradigma de programação adotado (por exemplo, Orientação a Objeto);
- Faça o uso adequando, respeitando as boas práticas da linguagem e framework de programação escolhido;
- Persiga a alta coesão dos seus métodos/funções e o baixo acoplamento entre componentes e estruturas;
- Tenha fácil legibilidade e entendimento (comentários ou métodos coesos com assinaturas reveladoras de intenções), (EVANS, 2003);
- Tenha a maior parte do código coberta por testes de unidade.

Infelizmente, esse não é o padrão que se vê em campo, o que culmina por reproduzir cenários como o narrado no início, gerando uma grande quantidade de aplicações “legadas” de baixa qualidade, que sustentam e vão sustentar por muitos anos ainda os negócios das empresas. Como o custo e o risco de evolução dessas aplicações são altos, quem arca com o ônus desses problemas é o próprio negócio, que contará com modificações cada vez mais custosas,

lentas e instáveis nas aplicações.

Isso, supostamente, deveria apoiar nas evoluções feitas para responder ao mercado. E ainda como, na maioria das vezes, as novas aplicações são construídas com os mesmos modelos em que foram construídas as “legadas”, o novo já nasce velho, e continua-se a gerar aplicações com baixíssima qualidade de código, que novamente penalizará o negócio e o custo de manutenção/evolução durante todo seu ciclo de vida (anos ou décadas).

Na plataforma Microsoft de desenvolvimento de software (família Visual Studio), existem ferramentas embarcadas que podem facilmente instrumentar as empresas para começar a reverter esse quadro de forma rápida. Podemos citar como a tríade de qualidade de código utilizando o Visual Studio, as ferramentas que serão detalhadas em seguida:

- Code Analysis - <https://msdn.microsoft.com/en-us/library/dd264897.aspx>
- Code Metrics - <https://msdn.microsoft.com/en-us/library/bb385910.aspx>
- Code Coverage - <https://msdn.microsoft.com/en-us/library/dd264975.aspx>

Code Analysis

O Code Analysis foi desenvolvido com o objetivo de disponibilizar ao desenvolvedor uma maneira fácil e rápida de validar seu código contra uma extensa biblioteca de boas práticas de programação para plataforma Microsoft. Ele realiza uma análise estática de código, ou seja, não analisa o código em execução (*runtime*), mas sim o código compilado de maneira estática.

Diariamente são encontrados problemas que geraram impactos aos usuários e também ao negócio, os quais poderiam ser facilmente

identificados e tratados proativamente com o uso do Code Analysis. Os exemplos mais comuns e trágicos desses cenários ocorrem com implementações de exceções e com componentes que implementam a interface `dispose`.

No caso do tratamento de exceções, detalhes de erros críticos são perdidos ou omitidos, exceções são capturadas desnecessariamente ou tratadas de maneira incorreta, e exceções não tratadas são transferidas ao usuário, expondo informações internas da aplicação. No caso do desenvolvimento de componentes que implementam a interface `dispose`, recursos como memória e conexões têm seu uso elevado desnecessariamente por objetos não gerenciados não terem sido liberados, causando, em casos extremos, paradas das aplicações por exceder a capacidade de memória, do limite de conexões ou até mesmo bloqueio de arquivos.

Para ter acesso inicial ao Code Analysis, acesse as propriedades de um projeto no Visual Studio e clique na aba *Code Analysis*:

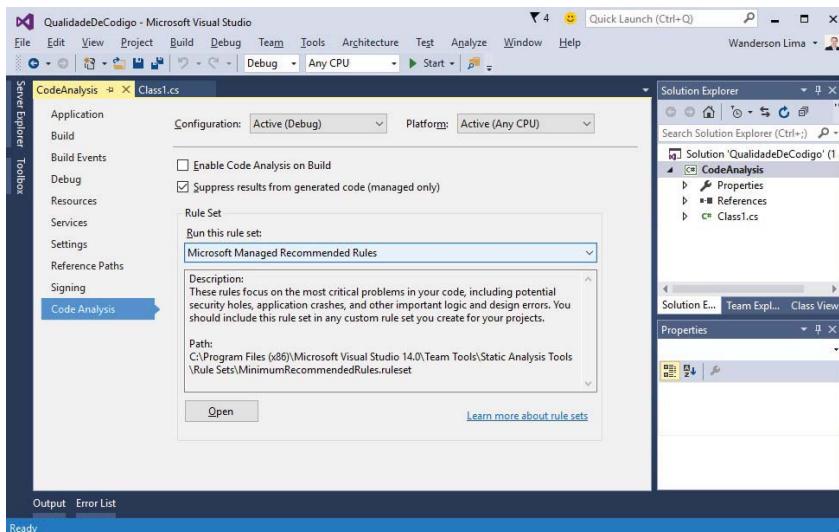


Figura 4.17: Configuração inicial do Code Analysis nas propriedades de um projeto no Visual Studio

As mais de 200 regras preexistentes do Code Analysis podem ser agrupadas em *rulesets*. Por padrão, existem *rulesets* mais básicos, mais abrangentes, focados mais em código gerenciado ou código nativo. Além destes, é possível criar *rulesets* personalizados priorizando as regras mais críticas que se quer atacar em cada momento.

O ruleset padrão, Microsoft Manage Recommended Rules, é certamente um ótimo ponto de partida para verificar se precisará ou não customizar um ruleset específico para a sua necessidade atual. Esse ruleset possui mais de 60 regras divididas em categorias conforme mostra a figura a seguir.

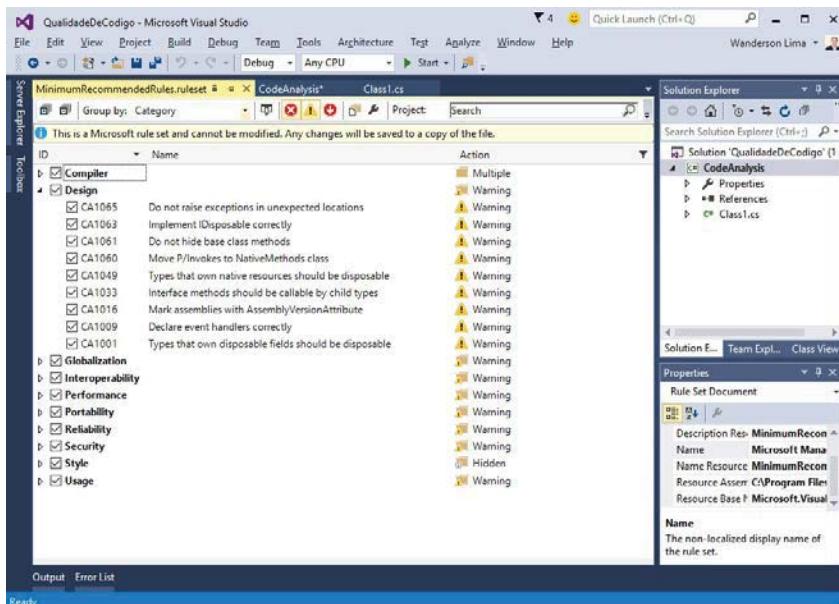


Figura 4.18: Conjunto de regras do code analysis, utilizando Visual Studio

Durante a customização de um ruleset, além de selecionar as regras desejadas, ainda pode ser configurada qual ação será gerada quando elas forem violadas: alerta, erro, informação. Nos rulesets padrões, todas as regras selecionadas são configuradas para gerarem

alertas.

Um ponto importante aqui é que a execução do Code Analysis pode ser associada ao build do projeto. Nesse caso, se alguma regra configurada para gerar erro for violada, o resultado do build falhará, mesmo que a compilação tenha sido executada com sucesso. Este também é um ponto importante a ser explicado: o Code Analysis é executado sobre as DLLs, logo após o processo de compilação bem sucedido. A partir das DLLs é executado um processo de validação sobre a linguagem intermediária gerada, em busca de violações das regras configuradas. Assim, em alguns cenários, as violações não têm uma relação forte com uma linha de código específica.

No total, são mais de 10 categorias nas quais se dividem as mais de 200 regras preexistentes de Code Analysis, formando assim um conjunto bastante abrangente de aspectos pela qual a aplicação será avaliada em relação as boas práticas de codificação. Além disso, cada regra possui uma clara documentação que explica desde a causa da violação, passando pela descrição da regra, como corrigi-la, e quando seria seguro ignorá-la, como descrito na figura seguinte, extraída do portal do MSDN (categorias existentes em azul no menu esquerdo da figura):

The screenshot shows a navigation bar at the top with links to Technologies, Downloads, Programs, Community, Documentation, and Samples. Below this, a breadcrumb trail indicates the current location: ... > Improve Code Quality > Analyzing Application Quality > Analyzing Managed Code Quality. The main content area has a title 'Code Analysis for Managed Code Warnings' and a subtitle 'Visual Studio 2015 | Other Versions'. A sidebar on the left contains links for Anonymous Methods and Code Analysis, How to: View Managed Code Defects, How to: Create a Work Item for a Managed Code Defect, and a expanded section for 'Code Analysis for Managed Code Warnings' which includes 'Warnings By CheckId' and a list of warning types: Design, Globalization, Interoperability, Maintainability, Mobility, Naming, Performance, Portability, Reliability, Security, and Usage. The main content area also contains a table with the following data:

Item	Description
Type	The TypeName for the rule.
CheckId	The unique identifier for the rule. CheckId and Category are used for in-source suppression of a warning.
Category	The category of the warning.
Breaking Change	Whether the fix for a violation of the rule is a breaking change. Breaking change means that an assembly that has a dependency on the target that caused the violation will not recompile with the new fixed version or might fail at run time because of the change. When multiple fixes are available and at least one fix is a breaking change and one fix is not, both 'Breaking' and 'Non Breaking' are specified.
Cause	The specific managed code that causes the rule to generate a warning.
Description	Discusses the issues that are behind the warning.
How to Fix Violations	Explains how to change the source code to satisfy the rule and prevent it from generating a warning.
When to Suppress	Describes when it is safe to suppress a warning from the rule.

Figura 4.19: Categorias das regras de code analysis (Portal MSDN)

Para executar a validação do ruleset previamente configurado no Code Analysis, basta clicar com o botão direito sobre o projeto no Visual Studio e selecionar *Analyse > Run Code Analysis*. Na janela *Error List* serão mostrados os erros, alertas e mensagens não só do processo de compilação, mas também do resultado da validação das regras de Code Analysis (configuração padrão para apresentar mensagens de Build + IntelliSense).

Quando uma violação é selecionada, caso exista uma relação direta com alguma linha de código, ela será colocada em destaque para que facilite a sua correção. A descrição apresentada tenta deixar clara quais variáveis e quais usos do código em específico estão violando a regra. Porém, caso sejam necessárias mais informações sobre ela, ao clicar no código da regra, o usuário é redirecionado para documentação completa sobre ela no MSDN, como descrito anteriormente.

Percebe-se que a regra violada na figura a seguir é exatamente a

que detecta cenários de não desalocação de objetos não gerenciados, o que poderia causar um enorme mau uso de recursos, podendo levar até ao seu esgotamento, como descrito no início desta seção.

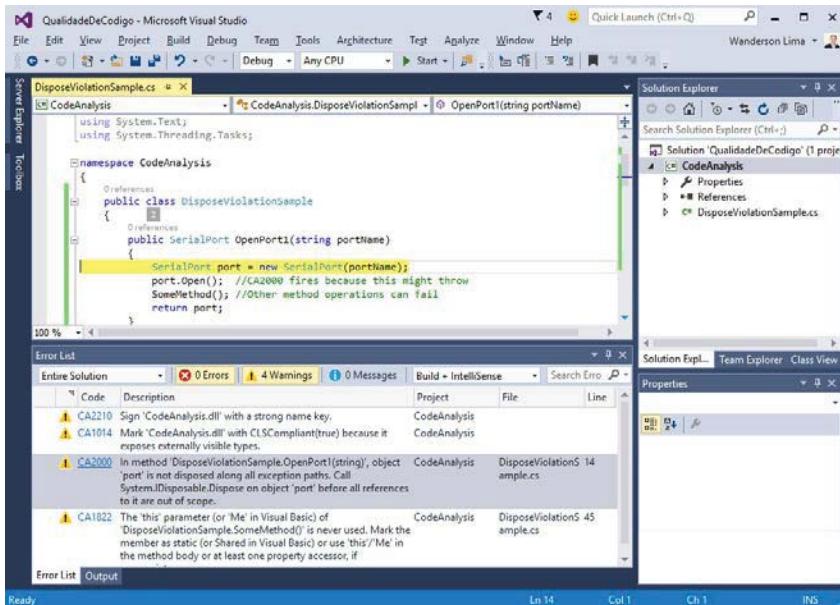


Figura 4.20: Exemplo de alerta de regra do Code Analysis

Assim, temos uma ferramenta consolidada com uma extensa base de boas práticas de programação na plataforma Microsoft, disponível na própria IDE do desenvolvedor, que explica o que é e qual a importância das regras, apoiando a rápida identificação e solução das violações.

Sem dúvida, esta é uma ferramenta mandatória em ciclos maduros de desenvolvimento. Embora cubra apenas parte da disciplina de Qualidade de Código, o seu uso certamente vai garantir um patamar mais elevado nesse tema.

Code Metrics

Uma outra ferramenta presente no Visual Studio, e com um propósito complementar ao Code Analysis, é o Code Metrics. Embora também esteja no âmbito da análise estática de código descrita no tópico anterior, o Code Metrics tem o propósito de medir e atribuir um índice de manutenibilidade de um projeto do Visual Studio. Ou seja, este índice tenta quantificar o quanto será mais fácil ou mais difícil realizar manutenção no código que está sendo gerado.

Certamente este é um ponto importantíssimo da disciplina de Qualidade de Código, uma vez que, diferentemente de construções mais concretas como edificações que tendem a sofrer menos mudanças durante o tempo, sistemas de informação devem ser feitos para serem alterados com relativa facilidade, conseguindo assim catalisar os negócios que estes suportam, na igual velocidade em que os mesmos se adequam para se manter competitivos ou tirar proveito de uma grande oportunidade de mercado. Não é raro ser encontrado em campo, métodos, classes e páginas muito grandes, com lógicas complexas e pouco intuitivas, que geram repulsa em quem precisa fazer algum tipo de melhoria ou alteração nelas. O pior ponto aqui é que, normalmente, esses componentes concentram a maior parte da lógica de negócio do sistema, sendo afetado de alguma forma em quase todas as alterações.

Assim, a manutenibilidade é um aspecto arquitetural de todos os sistemas e deve ser analisada em cada atividade de design da solução. Ignorá-la irá certamente aumentar o custo total de propriedade durante todo o ciclo de vida da aplicação, e penalizará os usuários com uma lenta resposta às mudanças e instabilidade frequente. Não distante da arquitetura, a manutenibilidade depende de conceitos básicos de componentização como alta coesão e baixo acoplamento, de onde irão derivar algumas das suas métricas que compõem o índice final.

Para executar o Code Metrics em um projeto do Visual Studio, devesse clicar com o botão direito sobre o projeto e selecionar as opções Analyze -> Calculate Code Metrics . Na janela Code Metrics Results serão mostradas hierarquicamente, do projeto até o método, o valor do índice de manutenibilidade de cada item:

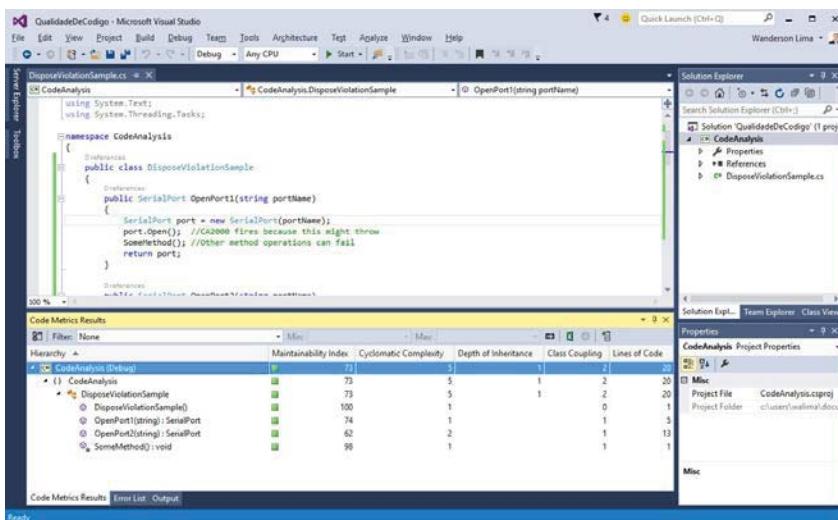


Figura 4.21: Exemplo de execução de Code Metrics no Visual Studio

É possível notar que, após a coluna com o índice de manutenibilidade, existem mais quatro outras métricas que, combinadas, resultam no valor final do índice. Começando pela métrica mais intuitiva tem-se a quantidade de linhas de código. Não se trata de linhas efetivamente escritas em linguagem de alto nível, mas sim instruções compiladas em linguagem intermediária, desprezando linhas em branco, comentários e todos os outros elementos de estruturação. Não se tem um número exato em relação à quantidade ideal de linhas de código, mas certamente um método/função terá dificuldades em ser coeso (fazer um conjunto pequeno e conciso de atividades) se tiver uma quantidade maior do que 300 linhas. Assim, valores altos nessa métrica impactam

negativamente no índice final de manutenibilidade.

Incrementando a métrica anterior, tem-se uma segunda análise mais profunda das linhas de código, que calcula a quantidade de caminhos de execução distintos que podem ser tomados durante a execução de um método/função. Qualquer instrução que possibilite um salto condicional para uma outra instrução adiciona mais um no cálculo da complexidade ciclomática, que é o nome dado a essa métrica (número de caminhos distintos). Métodos/funções com mais de 25 caminhos sinalizam claramente que estão fazendo mais de uma atividade e, certamente, não estão mantendo a coesão.

Normalmente quando se aborda esse ponto surgem várias dúvidas quando temos métodos roteadores que por natureza terão muitos caminhos de saída. É extremamente importante que esses métodos mantenham a coesão em rotear, enquanto outros métodos irão realmente encapsular a lógica da solução. Mantendo essa coesão, um número alto de complexidade ciclomática certamente será balanceado por um número baixo de quantidade de linhas de código, por exemplo. Um cenário que simboliza como métodos com grande complexidade ciclomática são penosos para a manutenibilidade é a implementação de testes de unidade, onde deverá ser implementado pelo menos um teste de unidade para cada caminho de execução. Se for feito pelo menos um teste positivo e outro negativo, esse número duplica.

Entrando agora em um outro conceito arquitetural, acoplamento, temos a métrica de Acoplamento de Classe. Também não é raro no campo calcular o grafo de dependência da solução e chegarmos a um resultado onde a maior parte dos componentes dependem um do outro.

Dessa forma, a avaliação de impactado de uma mudança se torna muito mais complexa, pois não existe o isolamento de conceitos em componentes específicos da aplicação, não sendo

seguro confiar que uma alteração em uma entidade vai se restringir a impactar um ponto particular e a suas poucas dependências que o consomem. Cada dependência que um método tenha para uma outra classe incrementará o contador de acoplamento, impactando negativamente o índice.

Por último, tem-se a métrica de profundidade de herança. Classes que possuem vários níveis de especialização de suas classes ancestrais tornam o entendimento da lógica menos intuitivo e, consequentemente, mais complexo de ser entendido e mantido. A profundidade de herança de uma classe será a profundidade de herança da sua classe-pai mais um. Valores maiores nessa métrica afetarão negativamente no índice final.

Aplicando um cálculo complexo que relaciona todas essas quatro métricas, chega-se ao índice final de manutenibilidade, que recebe o indicador verde para valores entre 100 e 20, amarelo para valores entre 19 e 10, e vermelho para valores entre 9 e 0. Códigos com o indicador vermelho devem ser refatorados urgentemente, enquanto códigos em amarelo são altamente recomendados que se refatorem, e códigos em verde possuem uma manutenibilidade aceitável.

Assim, com o uso do Code Metrics é possível também de maneira fácil, rápida e contínua, avaliar a complexidade das soluções que estão sendo desenvolvidas, podendo refatorá-las rapidamente em um momento próximo a sua concepção, antes que esta se torne um legado e seja de cara manutenção durante todo seu ciclo de vida, até a sua desativação.

Code Coverage

A cobertura de código faz uso da execução de testes de unidade sobre os componentes da solução (análise dinâmica de código em tempo de execução), para calcular quantas linhas de código

(numérica e percentual) foram executadas durante a execução dos testes na solução.

O objetivo desta seção não é se aprofundar em testes de unidade, mas será descrito brevemente o conceito básico necessário para compreender o mecanismo de cálculo da cobertura de código.

Os testes de unidade são os primeiros testes que devem ser feitos, onde normalmente são construídos pelo próprio desenvolvedor, visando testar a menor unidade testável do sistema (métodos e funções normalmente). Existem técnicas de desenvolvimento como TDD (Test Driven Development), que pregam que o teste de unidade deve ser feito antes mesmo do que o próprio código. Isso redefine o objetivo do trabalho de codificar como sendo a implementação do código necessário para fazer com que os testes executem com sucesso.

Testes de unidade são automatizados por definição (código testando código), e o ideal é que sejam isolados de suas dependências via Mocks/Stubs. Assim, tornando a sua execução mais rápida e frequente.

Uma vez que existam projetos de teste de unidade em uma solução do Visual Studio, ao clicar no menu *Test > Analyse Code Coverage > All Tests*, os métodos de teste (*test cases*) serão identificados e executados, abrindo ao término a janela *Code Coverage Results*. Nela é possível descer nível a nível da hierarquia da DLL (assembly, namespace, classe, método), possibilitando a análise do número de blocos cobertos ou não cobertos (numérico ou percentual) pela execução dos testes em cada nível.

Novamente, esse número se refere às instruções de linguagem intermediária resultantes da compilação, e não às linhas de código em linguagem de alto nível. Aqui, temos um ponto de atenção importante sobre o tipo de compilação utilizada. Como o código em

linguagem intermediária é drasticamente otimizado em modo *Release*, se comparado com a compilação em modo *Debug*, os números relativos à cobertura podem variar dependendo do tipo de compilação. Assim, para fins de comparação, é necessário que seja definido o tipo da compilação que será usado para calcular as coberturas a serem comparadas.

Como é mostrado na figura seguinte, uma outra importante funcionalidade da cobertura de código é a coloração das linhas que estão cobertas, não cobertas ou parcialmente cobertas. Quando um teste de unidade executa uma linha de código completamente, esta é considerada coberta e colorida de azul. Caso a linha de código não seja executada na execução de testes, esta é considerada não coberta e colorida de vermelho.

Agora, para entendermos a terceira opção, linhas coloridas em amarelo por estarem parcialmente cobertas, temos de retomar à linguagem intermediária. Uma instrução de alto nível pode ser quebrada em uma ou mais instruções de linguagem intermediária, podendo acontecer cenários onde parte das instruções de baixo nível não é executada, deixando assim a linha como parcialmente coberta.

Um exemplo comum dessa situação ocorre na análise em curto circuito de expressões booleanas, onde, se o primeiro argumento de um operador lógico `E` for avaliado como falso, os demais argumentos não serão avaliados por já ser possível concluir que o valor final da expressão é falso. Nesse cenário, parte das instruções de baixo nível não será executada, embora a linha de código em alto nível foi executada, podendo ser considerada, assim, parcialmente coberta:

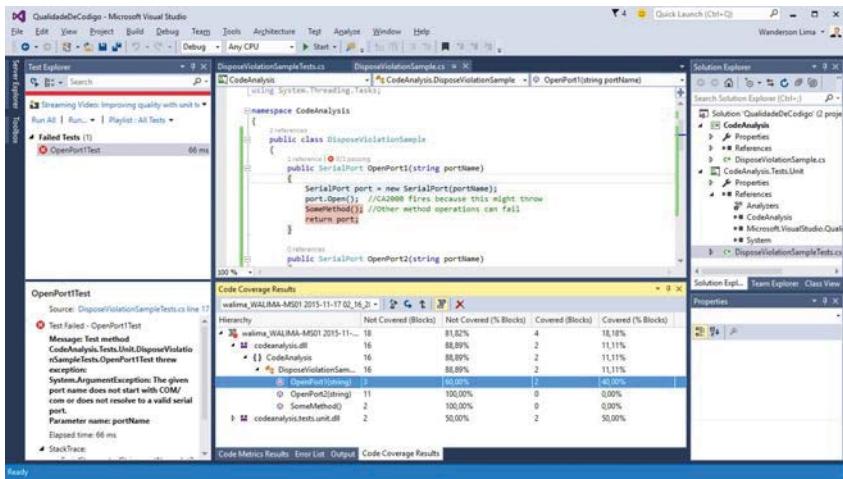


Figura 4.22: Exemplo de execução de teste de unidade no Visual Studio

Assim, com o uso da ferramenta de Code Coverage, conseguimos rapidamente aferir a cobertura mínima do código testado, identificando facilmente linhas de código não tratadas que necessitam de novos casos de teste de unidade.

Infelizmente, não é observado em campo um uso relevante de testes de unidade, se comparado com a amplitude dos seus benefícios. É sabido que o custo de resolução de um defeito cresce exponencialmente ao passar do tempo, desde quando ele foi criado durante o desenvolvimento.

Fatores como a aglomeração de mudanças de múltiplas pessoas, durante um longo período de tempo, propagada por vários ambientes e com regressiva capacidade e flexibilidade de debug em ambientes mais próximos de produção, tornam o diagnóstico e a resolução de um defeito muito mais caro e lento. Como mencionado anteriormente, práticas como TDD fazem com que as unidades sejam mais direcionadas aos cenários reais de uso, tornando o código mais enxuto.

O isolamento das unidades força a implementação considerando a testabilidade, favorecendo assim a adoção de uma arquitetura mais componentizada. A fácil e rápida execução dos testes de unidade permite uma execução frequente, identificando rapidamente não só problemas nas unidades alteradas, como também efeitos colaterais indesejados em outras partes do sistema, não diretamente relacionadas (regressão). O cálculo da cobertura de código facilita a avaliação da abrangência dos cenários de teste de unidade, apoiando assim uma implementação ideal dos testes para que permitam alcançar os benefícios descritos acima.

Qualidade de código durante todo o ciclo de desenvolvimento de software

As três ferramentas mencionadas aqui para qualidade de código (Code Analysis, Code Metrics e Code Coverage) foram apresentadas até o momento como facilitadores muito próximos aos desenvolvedores, para que estes possuam o ferramental necessário para o desenvolvimento de um código com boa qualidade. Contudo, em cenários de maiores proporções - onde há um grande número de projetos concorrentes podendo afetar a base de código de uma mesma aplicação, ou podendo ainda estar sendo executados por recursos internos ou externos, com diferentes estilos e proficiência de programação e uso dos frameworks - essas ferramentas configuraram um ótimo ponto de partida para a avaliação rápida, factual e periódica da qualidade do código. Isso possibilita a definição de metas a serem cumpridas para cada uma das métricas avaliadas pelas ferramentas.

A experiência de campo mostra novamente que existe uma preocupação principal com a entrega do projeto e seu respectivo aceite funcional (o que), deixando a qualidade do código implementado em segundo plano (o como). Como argumentado no início desta seção, essa é uma decisão míope por considerar apenas

o custo eventual da solução construída, não levando em consideração o custo total de propriedade ao qual estará sujeita a aplicação durante toda a sua existência, gerando problemas, impactos ao negócio e retrabalho devido a uma implementação inicial de baixa qualidade. Não obstante, o custo da possibilidade de não conseguir atender prontamente às modificações que o negócio exige também deve ser adicionado à conta.

Aos que já vislumbram a importância não apenas do o que foi entregue, mas também o como, as ferramentas apresentadas aqui fornecem uma visão inicial abrangente sobre o tema. Assim, é possível estabelecer metas para as métricas em contratos de prestação de serviço, em requisitos não funcionais de implementação, em critérios de aceite e\ou definições de pronto para reforçar o compromisso com a qualidade do que está sendo produzido.

Finalizando, o processo de desenvolvimento pode se tornar ainda mais fabril se adotarmos a execução destas ferramentas associadas ao Team Foundation Server (suíte de ALM, gerenciamento do ciclo de vida de aplicações da Microsoft) para que o processo de criação e promoção de um build seja automaticamente rejeitado no início se as metas definidas de qualidade de código não forem atingidas. Desta forma, o time rapidamente tem o feedback, ao mesmo tempo que os passos iniciais do desenvolvimento não evoluem até que a qualidade do código que está sendo gerado seja melhorada.

Estabelecida essa esteira de produção de código de boa qualidade, novas técnicas e ferramentas mais profundas podem ser avaliadas com a certeza de que a qualidade básica está sendo garantida.

REFERÊNCIAS

- Qualidade de código -
https://en.wikipedia.org/wiki/Software_quality
- Alta coesão -
[https://en.wikipedia.org/wiki/Cohesion_\(computer_science\)](https://en.wikipedia.org/wiki/Cohesion_(computer_science))
- Baixo acoplamento -
https://en.wikipedia.org/wiki/Loose_coupling
- Code Metric: linhas de código -
<https://blogs.msdn.microsoft.com/zainab/2011/05/12/code-metrics-lines-of-code>
- Complexidade ciclomática -
<https://blogs.msdn.microsoft.com/zainab/2011/05/17/code-metrics-cyclomatic-complexity>
- Cobertura de código - <https://msdn.microsoft.com/en-us/library/dd537628.aspx>

4.8 POR QUE INVESTIR EM QUALIDADE DO CÓDIGO?

Por Fernando Filiputti

Lembro-me de uma situação em que eu estava entregando um ALM Assessment (mais no box) para um cliente, e identificamos que a área de qualidade de código estava com baixo nível de maturidade devido à ausência de melhores práticas de engenharia de software pelas equipes de desenvolvimento.

ALM Assessment é um serviço que a área de Suporte Premier da Microsoft entrega para os clientes com o intuito de avaliar os seus níveis de maturidade em ALM. Além disso, neste trabalho criamos um plano de melhorias para evolução da maturidade das práticas de ALM.

Quando eu estava apresentando os resultados e o plano de melhoria, o CIO me perguntou:

“Vale a pena investir na qualidade do código? Não vejo razão para priorizarmos este item, uma vez que os desenvolvedores estão entregando com a qualidade esperada as demandas geradas pela nossa área de negócio. Gostaria de priorizar as melhorias no planejamento e controle dos projetos, já que este é um dos pontos que fazem eu perder meu sono. A minha área de negócio me cobra mais agilidade para entregar as novas funcionalidades, principalmente aquelas que nossos concorrentes já implementaram”.

Lembro-me de que durante uma das conversas que tive com a área de negócio durante a avaliação, eles realmente questionaram a produtividade da equipe de desenvolvimento:

“Nós temos uma série de novas funcionalidades que precisamos disponibilizar, mas a equipe de desenvolvimento não consegue “dar conta”, porque eles gastam muito tempo para manter os sistemas existentes e corrigindo os defeitos existentes. E por que os desenvolvedores investem mais tempo corrigindo defeitos em vez de trabalhar no desenvolvimento de novas funcionalidades?”

Bom, nesta empresa, o cenário era bem claro. Havia muitos sistemas complexos para se manter. Os desenvolvedores tinham dificuldades para entender o código existente durante as correções

de defeitos, o risco das implantações no ambiente de produção era alto, além da dificuldade para executar testes para garantir a qualidade das entregas. Infelizmente, eles se tornaram reféns desses sistemas legados. E isso estava impactando cada vez mais o time de desenvolvimento no dia a dia, já que eles não estavam sendo produtivos o suficiente para atender as demandas da área de negócio.

Apesar deste cenário, o planejamento e o controle dos projetos é o que estava tirando o sono do CIO. É claro que temos de levar em conta sua percepção e nosso trabalho é ajudá-lo a resolver seus problemas, mas será que estes problemas técnicos não estavam ajudando a tirar o seu sono? Como estes problemas técnicos estavam prejudicando as metas dos projetos? E por que isso era imperceptível para o CIO?

Eu não estava nesta empresa há alguns anos, mas tinha uma ideia de como tudo começou. Vamos voltar no tempo. Temos o cenário, onde o projeto tem um cronograma com uma data acordada para entrega do release. Em um determinado momento, surge a necessidade de incluir algumas funcionalidades adicionais no release, mas o gerente de projetos resolve manter a mesma data de entrega e também o custo.

No *Triângulo de Gerenciamento de Projetos*, temos uma representação das três variáveis de um projeto: tempo, custo e escopo, sendo que qualquer alteração em uma das variáveis impactará as outras partes. Além disso, temos uma quarta variável localizada no centro, que é a qualidade, onde uma alteração em qualquer uma das três variáveis, sem que alguma das outras partes seja também alterada, vai influenciar diretamente a qualidade.

Aplicando esse conceito no projeto anterior, ao aumentar o escopo e não alterar o prazo e o custo, fez com que a qualidade do projeto fosse impactada. Em um cenário mais prático, as decisões

tomadas neste caso poderiam ser: ausência de testes, ausência de boas práticas de codificação com o intuito de ganhar velocidade, decisões arquiteturais visando apenas velocidade e não manutenibilidade, extensibilidade etc.

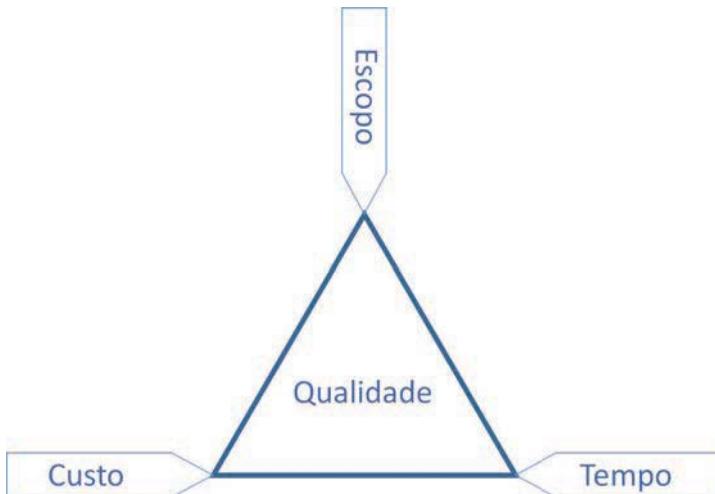


Figura 4.23: O Triângulo de Gerenciamento de Projetos

A soma de todos os problemas no código e na arquitetura da aplicação que impactam a manutenibilidade é conhecida como *Dívida Técnica*. Ao abrimos mão de qualidade, aumentamos a dívida técnica. Adquirir uma dívida técnica traz benefícios a curto prazo, tais como redução do tempo de desenvolvimento ou aumento do escopo, mas pode causar maiores custos a longo prazo.

No cenário anterior, vimos os desenvolvedores investindo muito mais tempo na manutenção de códigos complexos do que trabalhando no desenvolvimento de novas funcionalidades.

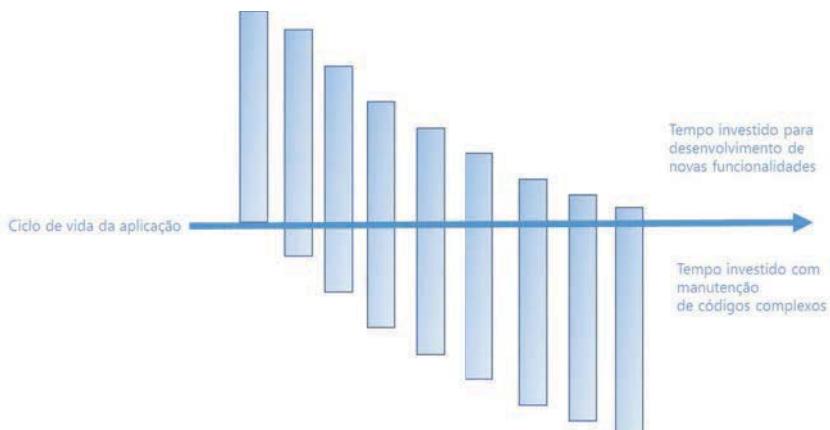


Figura 4.24: Ciclo de vida de uma aplicação com alta dívida técnica

Voltando a pergunta do CIO, vale a pena investir em qualidade de código? A minha resposta é: sem dúvida. Investindo na qualidade de código, deixaremos a nossa dívida técnica baixa e, consequentemente, o custo de manutenção do software será bem menor também, permitindo que os desenvolvedores entreguem valor mais rapidamente e de forma contínua.

Você não pode melhorar o que você não mede

Já foi mostrado o quanto é importante investir em qualidade de código e que uma dívida técnica alta pode ser um grande vilão do time de desenvolvimento e também dos objetivos dos projetos. Embora a dívida técnica seja composta de vários fatores técnicos, um bom começo para remediar o problema é atuar na melhoria da qualidade do código.

E qual o primeiro passo? Medir! Nas ciências exatas, sabemos que o primeiro passo para obter algum tipo de melhoria em algo é medir e ter uma *baseline* para comparação. No caso específico de projetos de software, medir também é muito importante para ter dados suficientes para convencer a equipe de gestão do projeto a

fornecer recursos para que o time de desenvolvimento faça as mudanças necessárias e resolva os problemas de qualidade.

Obviamente, ao fornecer dados técnicos sobre o seu projeto para a equipe de gestão do projeto, não será apresentado que: “o método A tem complexidade ciclomática 100”. Mas uma informação relevante, por exemplo, seria: “o nosso código tem 20% da dívida técnica e precisamos de X dias para consertá-lo”. Desta forma, você está permitindo que a equipe de gestão do projeto possa tomar decisões com base em dados concretos, além de priorizar adequadamente em vez de ter os pedidos sem argumentos para recursos e tempo para corrigir os problemas.

Ferramentas para coletar métricas de qualidade de código

Code Metrics

Já falamos sobre as métricas utilizadas pelo Code Metrics para calcular e exibir dados relacionados a complexidade e manutenibilidade de Código Gerenciado. Complementando a seção, seguem alguns outros exemplos de como o Code Metrics leva em consideração suas métricas:

- **Índice de manutenibilidade** - Calcula um valor entre 0 e 100, que representa a facilidade da manutenção do código. Um valor alto significa um código de melhor manutenção. A classificação verde é entre 20 e 100 e indica que o código tem boa capacidade de manutenção. Uma classificação amarela é entre 10 e 19 e indica que o código tem uma capacidade moderada de manutenção. A classificação vermelha é uma classificação entre 0 e 9 e indica baixa capacidade de manutenção.

- **Complexidade ciclomática** - Mede a complexidade estrutural do código. É calculada através do número de caminhos diferentes disponíveis no fluxo do código. Um código que tem fluxos complexos vai exigir mais testes para conseguir uma boa cobertura e terá uma menor capacidade de manutenção.
- **Profundidade de Herança** - Indica a quantidade de classes que se existem até a raiz da sua hierarquia de classes. Quanto mais profunda a hierarquia mais difícil pode ser para entender onde estão os métodos e campos que devem ser usados, dificultando a manutenção do mesmo.
- **Acoplamento de classes** - Mede o acoplamento de classes através de parâmetros, variáveis locais, chamadas de método, campos definidos em classes externas e etc. Um bom design de software determina que classes e métodos devem ter alta coesão e baixo acoplamento. Classes com alto acoplamento ser tornam difícil de reutilizar e manter devido às suas muitas interdependências com outras classes.
- **Linhas de código** - Indica o número aproximado de linhas no código. O cálculo é baseado em IL code e em virtude disso não é o número exato de linhas que consta no arquivo de código fonte. Uma quantidade muito grande de linhas de código pode indicar que um método não está coeso, tornando a sua manutenção e extensibilidade complexas.

É possível executar o Code Metrics via linha de comando através do utilitário >Visual Studio Code Metrics Powertool for Visual Studio 2015: ><https://www.microsoft.com/en-us/download/details.aspx?id=48213> Desta forma, é possível habilitar a execução do Code Metrics no build automatizado e incluir no processo de integração contínua.

SonarQube

O SonarQube (<http://www.sonarqube.org/>) é uma plataforma open source para gerenciamento de qualidade de código. É possível usar o SonarQube de forma integrada com o Visual Studio Team Foundation Server (TFS), ou com o Visual Studio Team Services (VSTS), para coletar métricas de qualidade de código como parte do processo de build automatizado e apresentá-los de forma que permita a equipe de desenvolvimento fazer o gerenciamento da dívida técnica relacionada ao seu código. O procedimento adiante mostra como fazer esta integração.

Para saber como instalar e configurar o SonarQube para analisar projetos .NET, consulte o guia criado pelos ALM Rangers: <https://github.com/SonarSource-VisualStudio/sonar-.net-documentation>.

1) Como pré-requisito deste procedimento, precisamos do servidor do SonarQube configurado.

2) Estando dentro do Team Project no qual se encontram o repositório de código-fonte e as definições de build, clique no botão

Administer Server no canto superior direito.

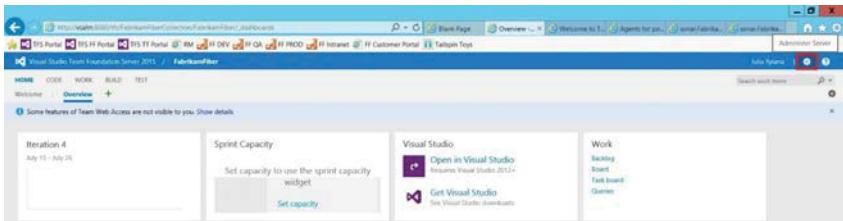


Figura 4.25: Acessando as configurações de Administração

3) Neste passo, vamos configurar um Service Endpoint. Ao acessar o *Painel de Controle*, clique na aba *Services*, depois na opção *New Service Endpoint* e escolha a opção *Generic*.

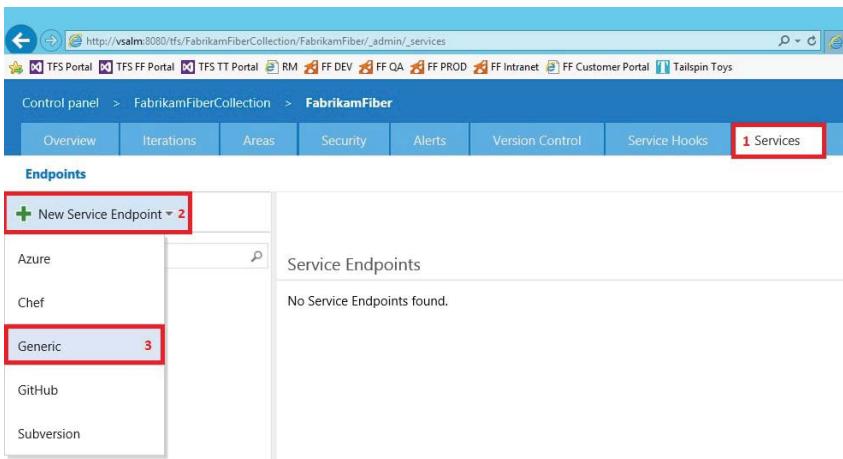


Figura 4.26: Cadastrando um novo Service Endpoint

4) Adicione um novo Service Endpoint para o servidor de SonarQube, informando um nome para a conexão e a URL do servidor do SonarQube. No exemplo a seguir, por se tratar de uma demonstração, o username e o password são configurados como *anonymous*. Após informar os dados, clique em OK.

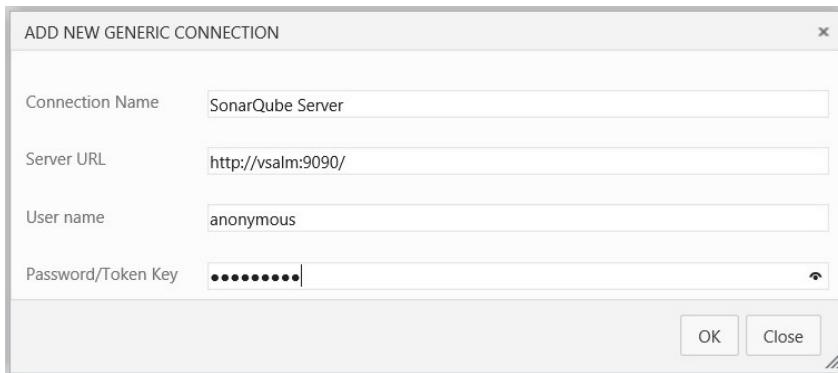


Figura 4.27: Cadastrando um novo Service Endpoint

5) O Service Endpoint está configurado e agora será usado para executarmos o SonarQube no processo de build automatizado.

The image shows the TFS Control Panel interface. In the top navigation bar, the URL is http://vsalm:8080/tfs/FabrikamFiberCollection/FabrikamFiber/_admin/_services. Below the URL, there's a ribbon with tabs: Overview, Iterations, Areas, Security, Alerts, Version Control, Service Hooks, and Services. The Services tab is selected. Under the Services tab, there's a sub-menu for "Endpoints". On the left, there's a search bar labeled "Search Endpoints..." and a list of endpoints. One endpoint, "SonarQube Server", is highlighted with a red box. To the right, there's a detailed view of the selected endpoint. This view includes a "INFORMATION" section with details like "Type: Generic" and "Created by Julia Ilyiana", and an "ACTIONS" section with links like "Update service configuration" and "Disconnect". Both the "INFORMATION" and "ACTIONS" sections are also highlighted with a red box.

Figura 4.28: Service Endpoint cadastrado

6) O próximo passo é configurar a Build Definition para que no processo de build automatizado seja iniciada a análise da qualidade do código-fonte pelo SonarQube. Considerando que já exista uma Build Definition, vamos alterá-la incluindo as tarefas necessárias

para a integração com o SonarQube. Com a Build Definition em modo de edição, clique na opção *Add build step*.

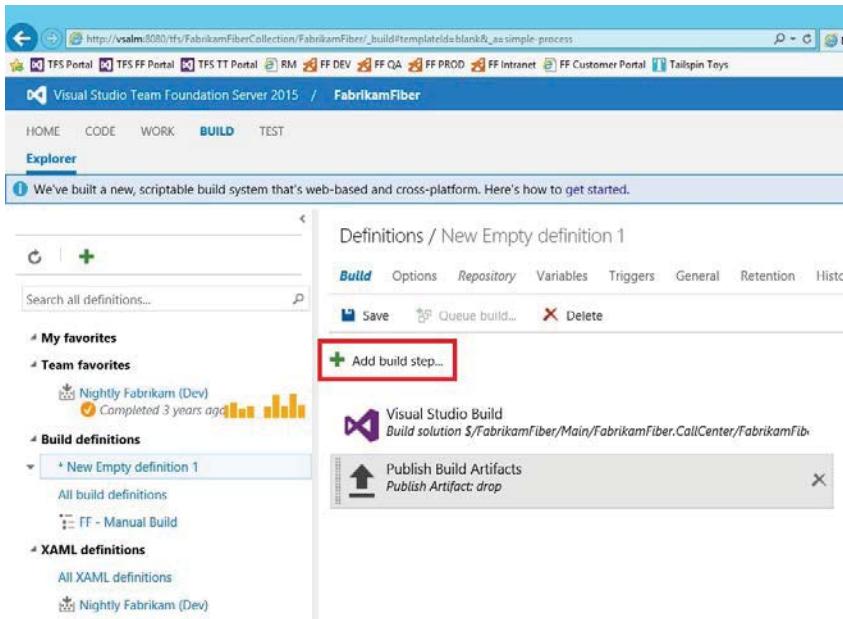


Figura 4.29: Editando a Build Definition

7) Na janela *Add Tasks*, escolha do lado esquerdo a categoria *Build*, e depois procure pelas tarefas: *SonarQube for MSBuild – Begin Analysis* e *SonarQube for MSBuild – End Analysis*. Para as duas tarefas, clique no botão *Add*. Após isso, clique no botão *Close*.

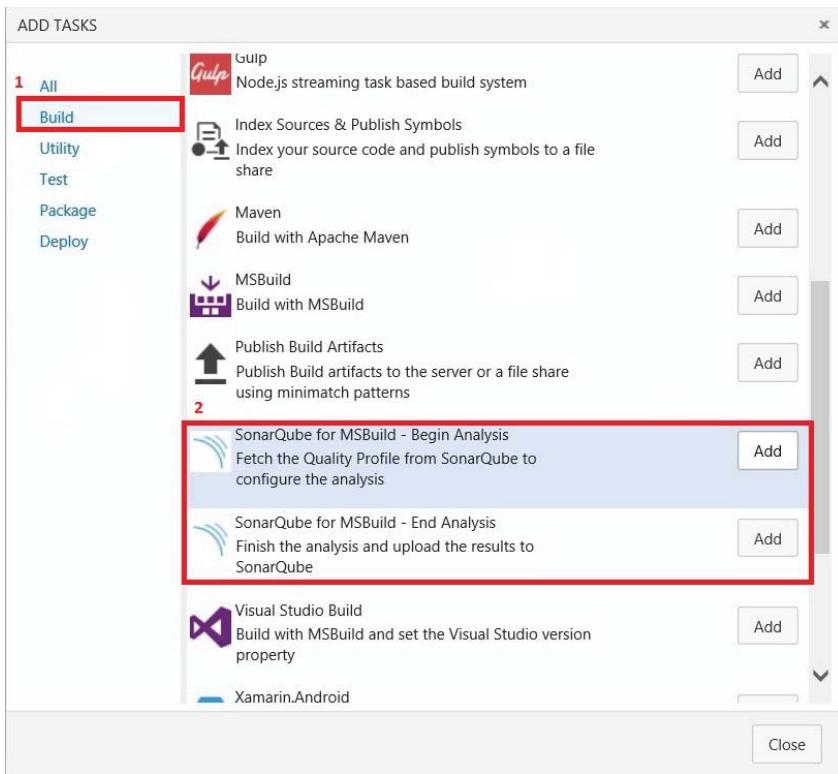


Figura 4.30: Incluindo as tarefas para execução do SonarQube

8) Ao voltar ao modo de edição da Build Definition, clique e arraste as novas atividades para que fiquem de acordo com a figura:

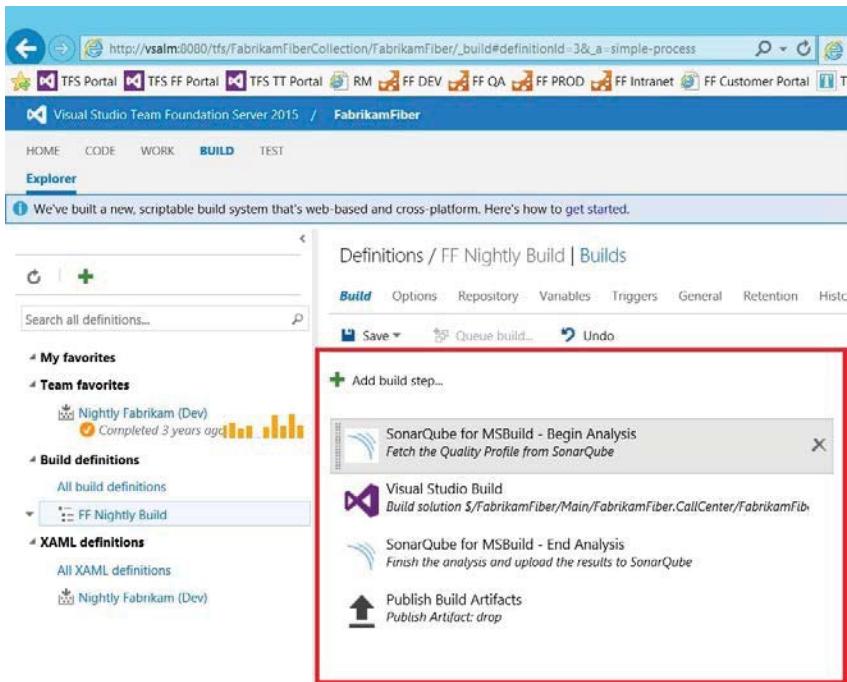


Figura 4.31: Alterando a ordem de execução das tarefas na Build Definition

9) O próximo passo é configurar as propriedades da tarefa *SonarQube for MSBuild – Begin Analysis*. Ao clicar na tarefa, serão exibidas as propriedades do lado direito da janela.

Para a propriedade *SonarQube Endpoint*, basta escolher no *drop down list* o Service Endpoint criado anteriormente. Para as propriedades dentro da categoria *SonarQube Project Settings*, devem-se incluir as propriedades necessárias para identificar as informações do projeto nos dashboards do SonarQube.

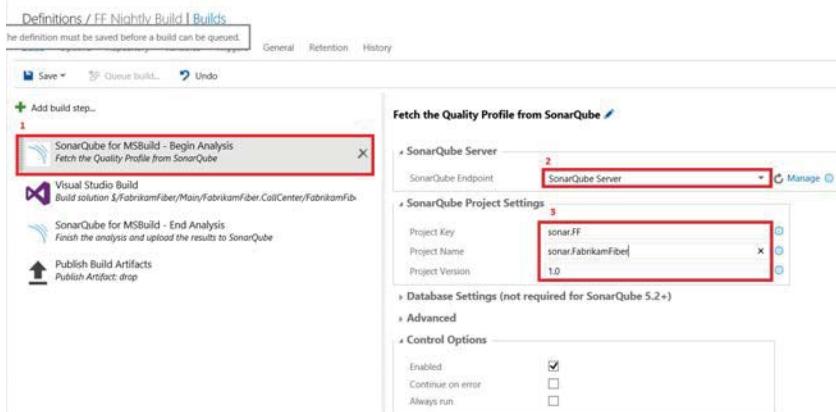


Figura 4.32: Configurando a tarefa do SonarQube

10) Após a configuração da primeira tarefa, clique na tarefa *SonarQube for MSBuild – End Analysis* e verifique as propriedades existentes. Neste exemplo, não alteraremos nenhuma delas.

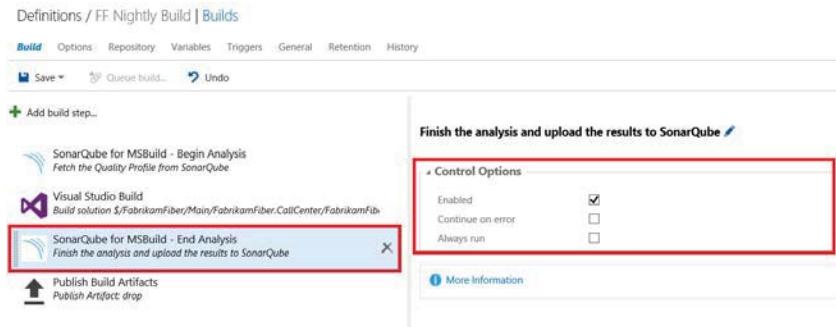


Figura 4.33: Configurando a tarefa do SonarQube

11) Após realizar as configurações, salve a *Build Definition*.

12) Para validar as alterações realizadas, vamos enfileirar um novo build e avaliar o seu resultado após a execução. Do lado esquerdo, podemos notar que as duas tarefas que acabamos de adicionar foram executadas. Do lado inferior direito, temos uma seção *SonarQube Analysis Report*, onde encontramos algumas

informações sobre a execução do SonarQube, e um link para os dashboards, onde encontraremos os detalhes da análise do código-fonte.

The screenshot shows the Microsoft Build interface. A message at the top says: "We've built a new, scriptable build system that's web-based and cross-platform. Here's how to get started." Below this, the build details for "Build 30" are shown. The build status is "Build Succeeded". The summary indicates the build ran for 46 seconds (default), completed 2.9 minutes ago. The "SonarQube Analysis Report" section is highlighted with a red border. It shows the analysis results for the project "sonar.FabrikamFiber", version 1.0. The report includes sections for Test results, Code coverage, and Associated work items. The "Test results" section notes that no test runs are available. The "Code coverage" section notes that no build code coverage data is available. The "Associated work items" section notes that no associated work items are found for this build.

Figura 4.34: Visualizando resultado da execução do build

13) O próximo passo é clicar no link *Analysis results* para abrir o SonarQube, onde encontraremos as seguintes informações sobre a análise do código-fonte:

* ***Dívida técnica*** A soma de todos os problemas no código e na arquitetura da aplicação que impactam a manutenibilidade. É medida em horas de trabalho estimadas para correção das violações e também por meio de uma taxa. A dívida técnica no SonarQube é baseada na metodologia SQALE36, e leva em consideração um conjunto de regras de análise estática de código, baixa cobertura de código pelos testes etc.

* ***Duplicações de código*** As duplicações de código medem a taxa de códigos semelhantes no projeto. O caso mais comum é o reúso de código pelo *_Copy and Paste_*.

* ***Estrutura*** Métricas relacionadas a estrutura do código fonte, por exemplo, quantidade de linhas de código, quantidade de classes, porcentagem de comentários, complexidade etc.

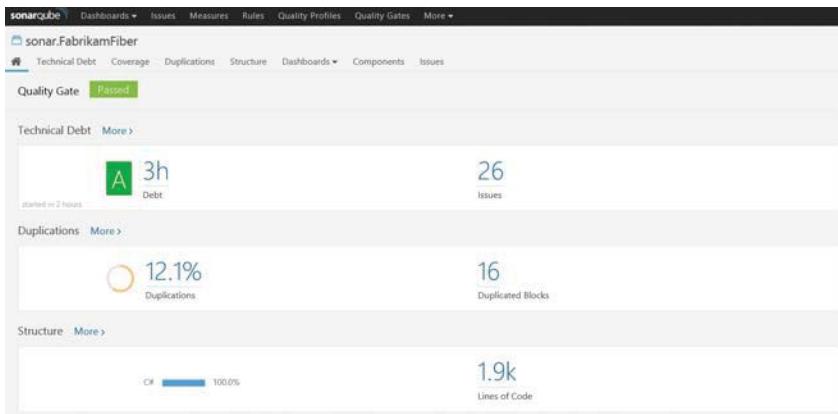


Figura 4.35: Visualizando dashboard do SonarQube

14) Na seção *Technical Debt*, podemos clicar no link *More* para obtermos mais detalhes sobre os pontos encontrados durante a análise. Nesta visão, conseguimos identificar as informações gerais sobre a dívida técnica, como a estimativa de esforço em horas para a correção, a taxa da dívida técnica e a quantidade de violações das regras. Além disso, o SonarQube categoriza as violações em níveis de criticidade diferentes (Blocker, Critical, Major, Minor, Info).

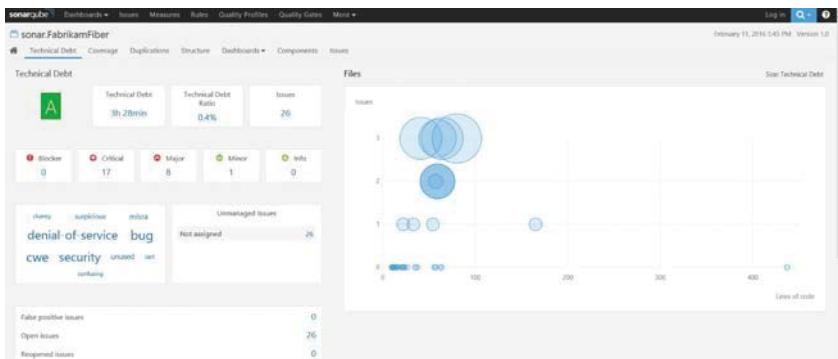


Figura 4.36: Visualizando detalhes da Dívida Técnica no SonarQube

15) Na seção *Structure*, podemos clicar no link *More* para obtermos mais detalhes sobre a estrutura do código analisado.

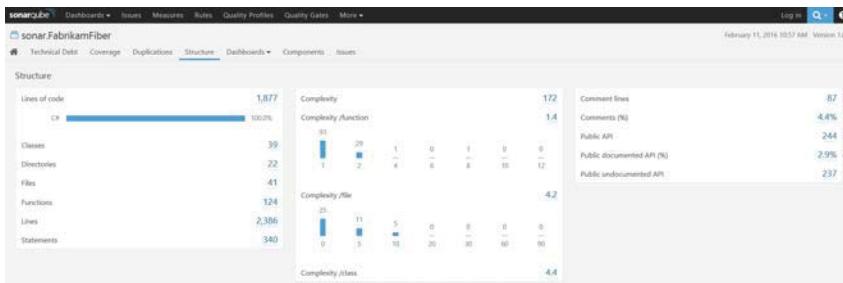


Figura 4.37: Visualizando detalhes da estrutura do código-fonte no SonarQube

REFERÊNCIAS

Triângulo de Gerenciamento de Projetos - <https://support.office.com/en-us/article/The-project-triangle-8c892e06-d761-4d40-8e1f-17b33fdcf810?ui=en-US&rs=en-US&ad=US>

Código gerenciado - [https://msdn.microsoft.com/en-us/library/windows/desktop/bb318664\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb318664(v=vs.85).aspx)

4.9 CONCLUSÃO

Neste capítulo, foram apresentadas técnicas de desenvolvimento recomendadas para todas as aplicações. Sua adoção evita problemas que venham a surgir durante as fases de desenvolvimento e manutenção de sistemas.

Os tópicos apresentados neste capítulo devem ser adotados desde as primeiras fases de concepção dos sistemas. Isso porque suas adoções tardias podem trazer maiores retrabalhos para as equipes de desenvolvimento.

De que adianta tomar todos os cuidados necessários para uma

aplicação ter qualidade se ela nunca é entregue em ambiente produtivo? No próximo capítulo, serão apresentadas técnicas e funcionalidades para garantir a entrega do software em produção com qualidade necessária.

CAPÍTULO 5

GESTÃO E MONITORAMENTO DE RELEASES

Desenvolver um software é tão importante quanto fazer sua publicação nos devidos ambientes. O desenvolvimento não tem valor algum se não for publicado, e a publicação não pode ser realizada se não há software desenvolvido. Um bom gerenciamento e monitoramento de releases permite à organização atuar proativamente em possíveis incidentes e, quando não identificados antes, atuar de maneira precisa e rápida na solução de problemas reativos.

Desta maneira, é possível responder as necessidades dos mercados interno e externo, garantindo a satisfação dos usuários e imagem positiva da organização. Este capítulo descreverá boas práticas de como gerenciar e monitorar seus releases.

5.1 BUILD E RELEASE

Por Leandro Prado

Após o processo de versionamento do código-fonte, é possível automatizar o processo de build e release para trazer qualidade no desenvolvimento de software. O processo de build garante que todos os componentes do software estão trabalhando em harmonia,

e o processo de release garante que o software será implantado de forma correta.

Normalmente os desenvolvedores estão acostumados a utilizar a tecla de atalho (F5, Ctrl+Shift+B etc.) do Visual Studio para compilar a aplicação e executar o projeto em modo de debug ou release, porém essa prática não garante que as alterações realizadas pelos outros membros do time estejam sendo executadas corretamente. Por esse motivo, necessitamos de um processo de build contínuo, conforme a figura a seguir, que seja responsável por executar as tarefas comuns como:

- Recuperar a última versão do código-fonte;
- Restaurar as dependências;
- Compilar o código-fonte;
- Executar o Unit Test;
- Testes de aceitação (Acceptance Test);
- Testes de aceitação do usuário (User Acceptance Test);
- Gerar o resultado do build.

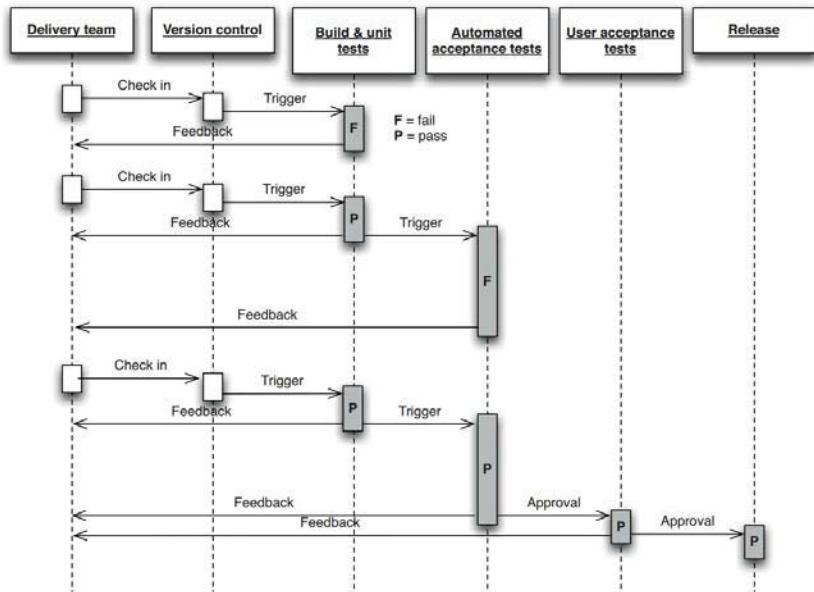


Figura 5.1: Diagrama de sequência para gerar uma release (HUMBLE; FARLEY, 2010)

A seguir, listamos alguns benefícios em automatizar o processo de build e release:

- Garantir a qualidade do software;
- Detectar problemas com antecedência;
- Verificar a todo o momento o que está funcionando e não funcionando;
- Executar testes automatizados;
- Receber feedbacks do código;
- Prover informações em tempo real para tomada de decisões;
- Diminuir o trabalho manual e erros humanos.

Gerenciando o build

“Um dos problemas do meu projeto é realizar a integração de todos os componentes e gerar uma nova versão. Normalmente, esse

processo é caótico e levamos dias para gerar uma versão confiável”.

Quando temos uma equipe trabalhando em diferentes partes do projeto ao mesmo tempo, temos de garantir que todos os componentes estão se integrando de maneira correta. Uma das boas práticas é usar integração contínua.

O QUE É INTEGRAÇÃO CONTÍNUA?

Integração contínua (*Continuous Integration*) é uma prática de engenharia de software para validar a integração de todos os componentes a cada alteração efetuada. Em outras palavras, cada vez que um desenvolvedor altera alguma parte do código, será disparado automaticamente um build onde podemos validar se o código está compilando, se todos os testes estão sendo executados (Unit Tests), e se está atendendo todas as regras de análise de código (Code Analysis).

Quando utilizamos o Team Foundation Server, podemos criar uma definição de build (*Build Definition*) onde podemos configurar como será executado o build da nossa aplicação. A figura adiante demonstra como é possível configurar o build para executar no formato de integração contínua.

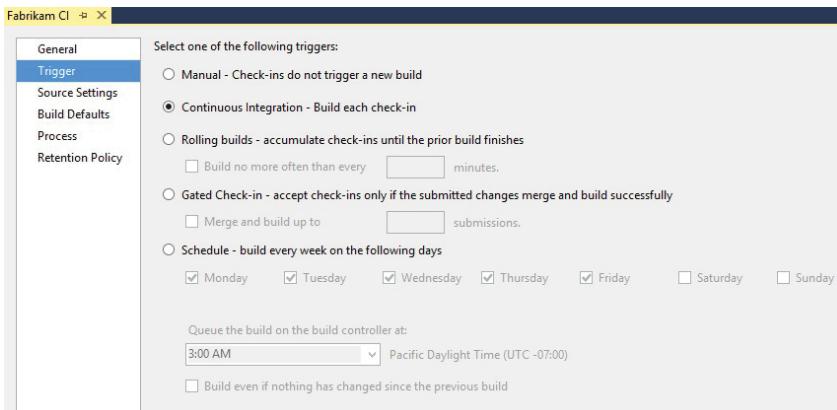


Figura 5.2: Configuração de integração contínua no Team Foundation Server

Além de configurar um build de integração contínua, o time de desenvolvimento deve se preocupar com as seguintes práticas:

- **Mantenha tudo versionado:** todo o código-fonte, banco de dados, scripts etc.
- **Configure seu build para validar seus testes:** na definição de build, configure para executar e validar os testes de software de forma integrada e automatizada.
- **Realize check-in diários:** dessa maneira, todo o time fica sabendo das alterações ocorridas no projeto.
- **Concerte o build quebrado imediatamente:** caso o build não seja executado com sucesso, concerté imediatamente, não deixe para o outro dia.
- **Mantenha seu build rápido:** dessa forma, você receberá mais feedbacks do seu código.
- **Recuperar de forma fácil a versão estável:** qualquer pessoa do seu time pode precisar de uma versão estável do projeto para demonstração, testes exploratórios etc.

Gerenciando o release

“Meu time de desenvolvimento possui acesso ao ambiente de

produção e fazem todos os deployments do projeto de forma manual, dessa forma não conseguimos ter nenhuma visibilidade e rastreabilidade das mudanças que ocorreram”.

Um dos maiores problemas que os times passam é gerar um novo release e colocar em produção. Geralmente, os processos são manuais, e muitas vezes o time de desenvolvimento tem acesso ao ambiente de produção, onde realizam todo tipo de alterações para fazer o projeto funcionar de maneira correta, fazendo com que depois de tantas alterações aquele ambiente não seja mais confiável. Para resolver esse tipo de problema, temos de documentar e automatizar a nosso Release Pipeline.

O QUE É RELEASE PIPELINE?

É o processo que determina como entregar software para o usuário final. Nesse processo, é recuperada uma alteração feita no projeto e realizada a publicação em vários estágios e ambientes. Em cada estágio, temos várias atividades para validar aquela alteração. Esse processo requer colaboração entre todos os membros dos times. É onde podemos ter a visão de todas as alterações que foram realizadas no projeto.

Na figura a seguir, podemos ver um exemplo de Release Pipeline utilizando o Release Management 2013, no qual temos 3 principais estágios (stages), DEV, HML e PRD.

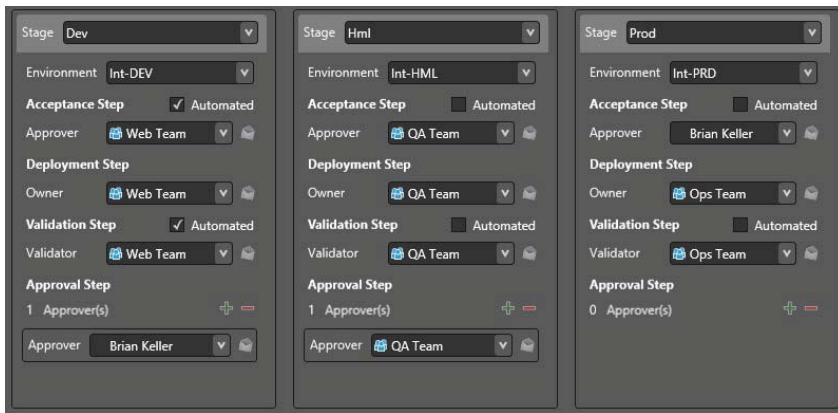


Figura 5.3: Release Pipeline

Nesse exemplo, temos os seguintes stages:

- **DEV:** nesse estágio, o desenvolvedor realiza seus testes e, por se tratar de um ambiente de desenvolvimento, não precisamos de aprovações, por isso podemos deixar tudo automatizado.
- **HML:** nesse estágio, o usuário tem a possibilidade de validar as alterações solicitadas, e nesse ambiente já necessitamos de aprovações.
- **PRD:** nesse estágio, as alterações são liberadas para o usuário final.

Após definir os stages, devemos configurar as tarefas (*steps*) que serão executadas em cada stage. Na figura, podemos ver os steps configurados para o stage de DEV.

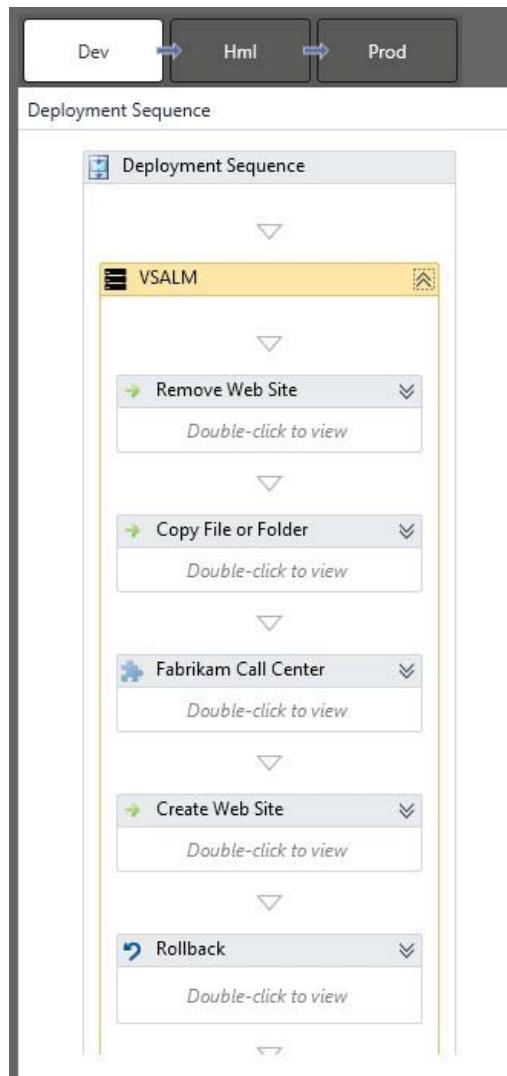


Figura 5.4: Steps para cada Stage

Quando estamos trabalhando com Release Pipeline, temos algumas práticas que devemos seguir:

- **Execute o build somente uma vez:** todas as vezes que compilamos o nosso código, temos o risco de

introduzir erros. Execute o build apenas uma vez e o mesmo build deve passar por todos os estágios, e o binário colocado em produção deve ser exatamente o mesmo que passou pelos testes.

- **Use o mesmo processo de deploy para todos ambientes:** é essencial usar o mesmo processo de deployment em todos os ambientes, pois desenvolvedores realizam deployments no ambiente de DEV diariamente. Dessa forma, asseguramos que o processo foi testado exaustivamente e, quando executarmos esse processo em produção, eliminamos a probabilidade de existirem erros. Use sempre o mesmo processo de deployment de produção nos ambientes de desenvolvimento, assim conseguimos garantir a consistência do processo.
- **Execute Build Verification Tests (BVT):** também conhecido como *Smoke-Test*, deve ser executado após cada deployment para assegurar que as principais funcionalidades da sua aplicação estão funcionando corretamente, como a página inicial, base de dados, webservices e serviços externos.
- **Execute o deploy para um ambiente semelhante ao de produção:** um dos grandes problemas dos times de desenvolvimento é ter um ambiente de produção diferente do ambiente de homologação. Para garantir a qualidade do deploy, sempre que possível você deve ter um ambiente similar ao de produção. Você deve garantir que seus ambientes tenham as mesmas configurações como:
 - Infraestrutura: topologia de rede e configurações de firewall;

- Sistema Operacional: mesmas versões, incluindo atualizações (*Services Packs*).
- **Cada alteração deve propagar por meio do Pipeline imediatamente:** toda alteração realizada no código-fonte deve disparar um build em que deveremos executar os testes (Unit Tests), os testes de aceitação, e depois executar o deployment no primeiro stage (DEV).
- **Se qualquer parte do Pipeline falhar, pare a release:** não importa em qual stage seu release está; caso encontre um problema, pare imediatamente, resolva o problema e recomece todo o processo de pipeline novamente.

Zero Downtime

“Minha aplicação é 24x7x365 e, sempre que vamos colocar uma nova versão em produção, temos de deixar a aplicação off-line por alguns minutos. Como posso fazer um deployment em produção sem deixar minha aplicação fora?”

Quando trabalhamos com aplicações de alta disponibilidade, um grande desafio que enfrentamos é como controlar o processo de release dessa aplicação, já que a cada segundo que ela ficar off-line estamos perdendo acessos e, consequentemente, dinheiro. Nesses casos, temos de partir para uma solução enterprise de gerenciamento de deployments. A seguir, descrevemos duas dessas soluções.

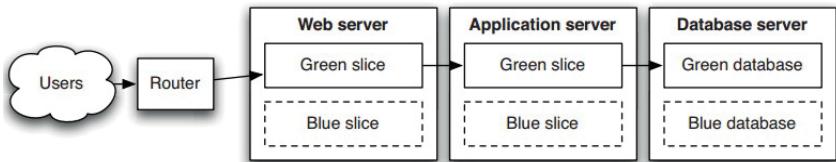


Figura 5.5: Blue-Green deployments (HUMBLE; FARLEY, 2010)

- **Blue-Green Deployments**

Essa técnica é uma das mais famosas usadas para gerenciamento de deployments. A ideia consiste em ter dois ambientes idênticos de produção, onde chamamos um de Blue e o outro de Green, e os usuários são direcionados para um desses ambientes.

Nessa figura, podemos ver um exemplo dessa técnica. Nesse caso, o ambiente de produção atual é o Green, todos os acessos são direcionados para esse ambiente e o ambiente Blue está inativo. Quando existir a necessidade de realizar um novo deployment da aplicação, esse deploy deve ser feito no ambiente atualmente inativo, nesse caso, o ambiente Blue. Todos os testes necessários devem ser executados até que a aplicação esteja de acordo com o esperado. O próximo passo é alterar a rota para que os usuários sejam direcionados para o ambiente Blue. Assim, o ambiente Blue será o novo ambiente de produção e o ambiente Green será inativo. Caso algum problema ocorra nesse novo deployment, de maneira simples, será possível redirecionar os acessos novamente para o ambiente Blue com *zero downtime*.

Uma outra opção para a técnica de Blue-Green deployments é utilizar apenas um ambiente de produção, porém usando portas diferentes: uma porta

para o ambiente Blue e outra porta para o ambiente Green. Dessa forma, podemos diminuir o custo de implementar essa técnica.

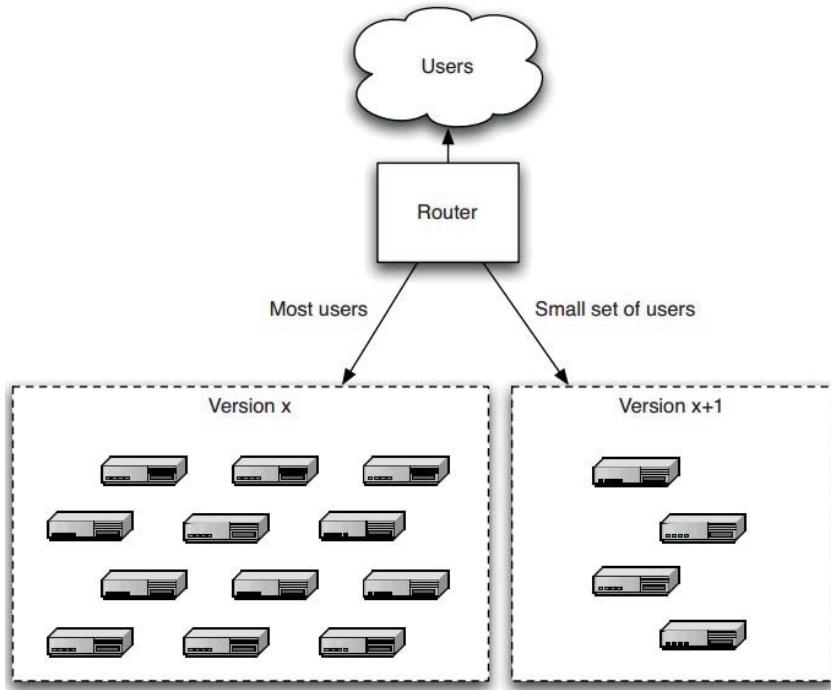


Figura 5.6: Canary Deployment (HUMBLE; FARLEY, 2010)

- **Canary Deployments**

A técnica Canary Deployment é utilizada ao ser executado um novo deployment da aplicação para um conjunto específico de servidores em produção e liberá-los para um grupo restrito de usuários, geralmente para usuários avançados. Assim, conseguimos realizar testes em produção e obter um feedback rápido das alterações realizadas, diminuindo o risco de usuários encontrarem erros.

Assim como no Blue-Green, você precisa fazer o deployment da nova versão da aplicação em um conjunto restrito de servidores onde executamos os testes necessários e, assim, direcionamos os acessos dos usuários para esses servidores, conforme podemos ver na figura a seguir. Assim que a aplicação estiver estável, realizamos o deployment para mais servidores e liberamos acesso para mais usuários.

Um dos benefícios em usar Canary é verificar se sua aplicação está se comportando conforme o esperado com um número pequeno de usuários, e aumentando gradativamente o acesso para mais usuários, sempre monitorando as métricas como utilização de CPU, memória e IO.

5.2 RASTREABILIDADE DE CÓDIGO-FONTE

Por Andreia Otto

No desenvolvimento de software, o termo rastreabilidade refere-se à capacidade de vincular os requisitos do produto de volta à lógica dos interessados (stakeholders) e aos artefatos correspondentes de design, código e casos de teste. Rastreabilidade apoia inúmeras atividades de engenharia de software, tais como análise de impacto de mudanças, verificação de conformidade ou de rastreamento de código, seleção de testes de regressão, e validação dos requisitos.

A utilização de práticas recomendadas leva o desenvolvimento de um software a produzir laços entre os artefatos, de modo que a rastreabilidade seja possível para todos os artefatos em todos os momentos do ciclo de vida da aplicação. Ou seja, é preciso criar os relacionamentos para posteriormente encontrá-los quando

necessário.

O controle de versões provê informações sobre mudanças de código. Lá é possível obter informações de “quem”, “o que” e “quando”. Porém, a pergunta mais difícil relacionada à rastreabilidade é “por que”, e o controle de verão geralmente não terá informações suficientes para esta pergunta. Mesmo quando o desenvolvedor insere um comentário ao fazer um check-in em que explica o motivo da mudança, o trabalho de detetive ainda precisa ser feito para obter todas as razões.

Os itens a seguir focam na rastreabilidade de código-fonte, entretanto, é preciso que, além disso, outras práticas de desenvolvimento sejam utilizadas em todo o projeto.

Código-fonte

“Estou com um problema em produção, os usuários estão recebendo uma mensagem de erro ao tentar executar uma ação no sistema. Não consigo identificar o código-fonte que está no ambiente para verificar a situação.”

Organize seu código-fonte em um repositório confiável, em que seja o único ponto de origem de versões da aplicação. Você deve ser capaz de identificar o código-fonte que originou todas as versões, para ser possível obtê-lo a qualquer momento durante o ciclo de vida da aplicação.

Todo processo de publicação em produção deve ser possível ser rastreado, a fim de identificar código-fonte, aprovações e mudanças incluídas na nova versão. Há diversas maneiras de promover rastreabilidade entre o código-fonte e o pacote de publicação, a seguir estão algumas opções de práticas para garantir a rastreabilidade:

- **Demarcação dos fontes:** atribuir marcas diretamente ao código-fonte é como tirar uma fotografia do momentos atual, em que no futuro será possível fazer referência e/ou obter um determinado estado do código.
- **Automatização de build com check-ins associados:** builds automatizados isolam a compilação do código-fonte da máquina do desenvolvedor, isto é, toda e qualquer dependência que exista deverá estar presente na solução para que não haja o problema de “na minha máquina funciona”. Desta forma, é possível garantir que o pacote gerado pelo build automatizado é uma fonte segura. Ao associar check-ins aos builds automatizados, é possível obter as mudanças ocorridas no build e identificar o estado do código-fonte de cada pacote.
- **Gestão de publicações:** gerenciar publicações permite ter visibilidade de mudanças ocorridas nos ambientes, desde o pacote contendo código-fonte até as devidas aprovações. É essencial possuir uma ferramenta única para gestão de publicação, que proporciona tal visibilidade e rastreabilidade integrada com a plataforma de desenvolvimento / ALM.
- **Gestão de mudanças:** ter um processo formal e regulamentado de mudanças é imprescindível para o sucesso da gestão de ambientes. Toda e qualquer alteração do ambiente deve ser aprovada e controlada por uma única ferramenta, capaz de prover informações de quais itens foram impactados por cada mudança.
- **Rastreamento de atividades:** todas as atividades

desenvolvidas por uma equipe de desenvolvimento devem ser mapeadas e associadas ao código-fonte ou artefato impactado. Ao realizar a associação das atividades de desenvolvimento com ele, é possível identificar responsáveis e alterações executadas para, caso ocorra algum problema, ser possível rastrear desde o código à solicitação.

O fluxo da aplicação deve seguir práticas que facilitam a identificação de possíveis problemas. Garantir que o desenvolvimento do produto seja planejado e rastreado usando ferramentas especializadas e processos definidos permite que o tempo de resposta a problemas / indisponibilidade reduza.

Nem sempre o problema descrito anteriormente é originado por falha no código-fonte; para isto, verifique as possíveis causas de erro antes de precisar dele. Configurações de ambientes, informações no registro, dependências externas, execução correta da funcionalidade por parte do usuário, entre outros devem ser previamente validados.

Banco de dados

“Uma alteração no sistema foi executada ontem ao final do dia, e hoje não está sendo possível cadastrar um novo usuário no sistema.”

“Não foi possível realizar atualização do sistema por falha no banco de dados.”

Quando pensamos sobre código-fonte, é muito comum associarmos apenas ao código da aplicação, e não ao banco de dados, seja ele qual for. Pela experiência em campo, parte considerável de problemas de publicação está relacionada com falhas de atualização de banco de dados, seja por objetos ou dados faltantes. Desta forma, é imprescindível que, além do controle de código-fonte de aplicação, a estrutura do banco de dados também

seja controlada por um repositório de fontes e versionada devidamente.

A Microsoft possui o SQL Server Data Tools (SSDT), uma ferramenta de desenvolvimento de banco de dados declarativa que abrange todas as fases do desenvolvimento dentro do Visual Studio. É possível utilizar recursos de design, Transact-SQL, depurar, realizar manutenções no banco de dados e também refatorar. Também, pode-se trabalhar com Azure SQL Database, modelos de Analysis Services, pacotes de Integration Services e relatórios do Reporting Services. Com SSDT, é possível modelar, desenvolver, compilar, testar e publicar bancos de dados e outros tipos de conteúdo tão simples quanto desenvolver uma aplicação usando Visual Studio.

Existem dois modelos de trabalho com SSDT. Um deles é de forma conectada a uma instância de banco de dados, como o SQL Server Management Studio, porém não se pode controlar as alterações. A outra maneira, que faz possível a rastreabilidade dos códigos-fontes de banco é o trabalho desconectado, utilizando um projeto do tipo banco de dados.

Para trabalhar com SSDT no Visual Studio, é preciso criar um projeto do tipo banco de dados. Abra o Visual Studio, clique em *Novo*, escolha *Projeto* e selecione *SQL Server Database Project*, conforme a figura a seguir:

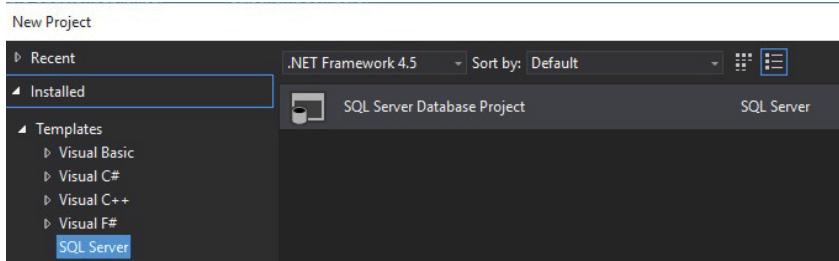


Figura 5.7: Criar novo projeto do tipo banco de dados. Para maiores informações sobre versões, consulte <https://msdn.microsoft.com/en-us/library/mt204009.aspx>

Uma vez criado o projeto, é possível importar um banco de dados já existente ou criar um novo. Para importar um já existente, na aba *Solution Explorer* do *Visual Studio*, clique com o botão direito no projeto de banco criado, conforme a figura seguinte:

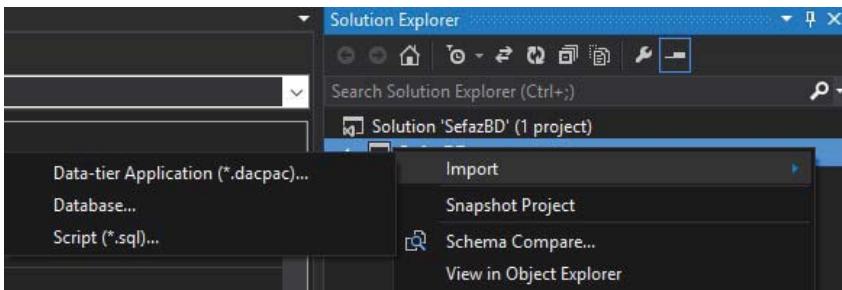


Figura 5.8: Como importar um banco de dados já existente

Para realizar a importação, existem três opções:

- **Data-tier Application (*.dacpac)**: arquivo dacpac pode ser extraído do banco de dados SQL, e contém a representação do schema do banco;
- **Database**: será feita a conexão direta com o banco de dados para importar a estrutura;
- **Script**: Script SQL, que realiza a criação do banco de dados completo. Caso seja um novo banco, basta criar os objetos diretamente no projeto.

As figuras a seguir representam a criação de uma nova tabela na estrutura do banco. Clique com botão direito no projeto, selecione *Add*, depois *New item*:

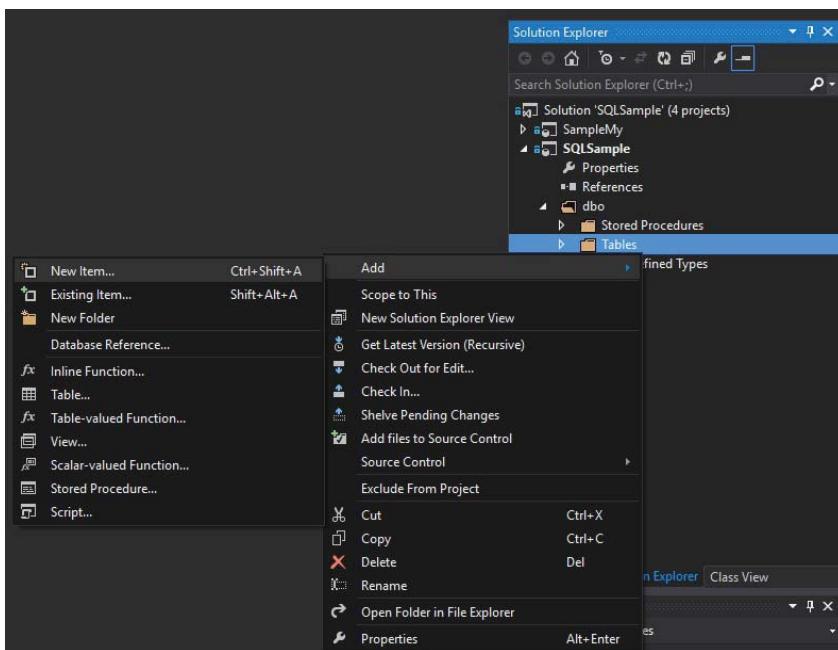


Figura 5.9: Adicionar um objeto ao banco de dados

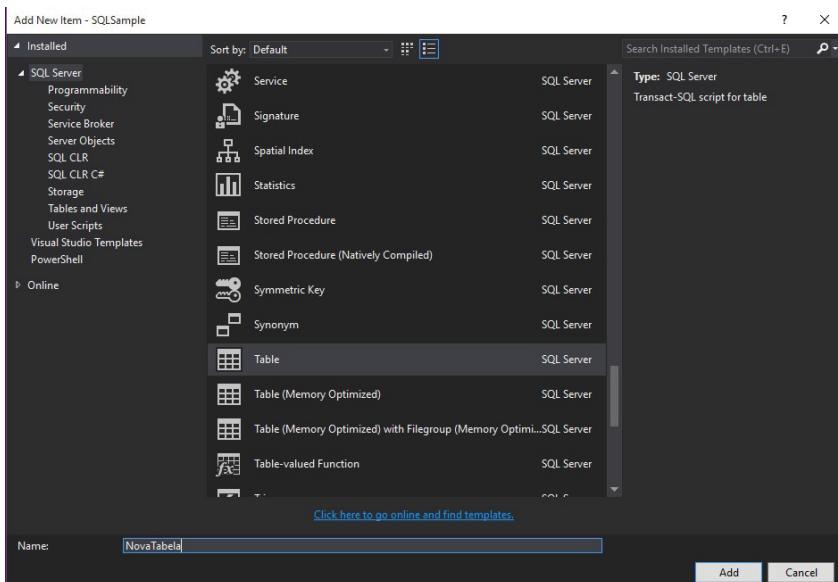


Figura 5.10: Nova tabela

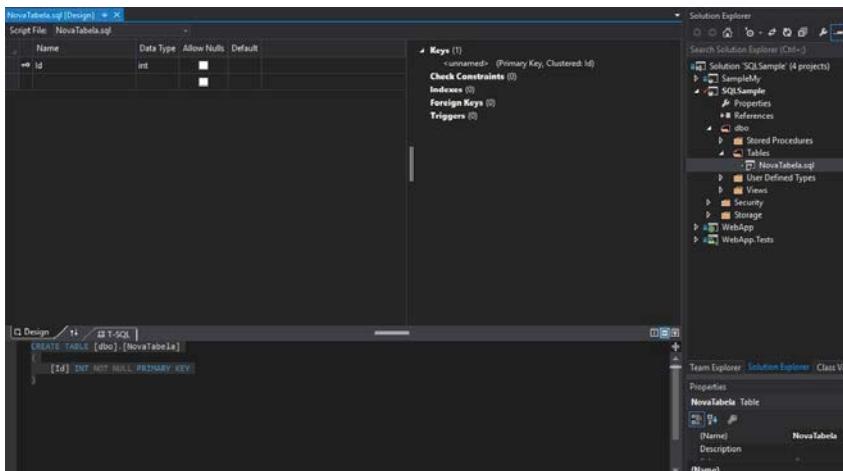


Figura 5.11: Nova tabela

É possível criar não só tabelas, como a grande parte de objetos do SQL Server. Além das práticas recomendadas para manter rastreabilidade de código-fonte, para estrutura de banco de dados existe mais uma, é possível criar uma fotografia (snapshot) do estado atual do projeto. Isto é, pode-se ter uma linha de base de determinados momentos do seu projeto.

Uma funcionalidade extremamente importante quando falamos sobre desenvolvimento de banco de dados é a possibilidade de realizar comparação entre estruturas (schemas). Usando `schema compare`, é possível identificar diferenças tanto de dados quanto de estrutura entre projeto e banco de dados, entre dois bancos de dados e entre projetos.

Assim, antes de realizar qualquer atualização, é recomendado que as diferenças sejam validadas como também toda e qualquer alteração realizada em ambientes controlados devem ser originadas do projeto de banco de dados, garantindo assim a fonte única de origem.

5.3 COMO EXTRAIR O MÁXIMO DO LAB MANAGEMENT PARA GARANTIR A QUALIDADE DO SEU SOFTWARE

Por Ricardo Serradas

Na era atual, já é indiscutível que a qualidade de software é indispensável para projetos e produtos de sucesso. A competitividade está cada vez maior e a velocidade com que novos produtos surgem é alucinante. Com isso, se o produto não for no mínimo excelente, a certeza de um tempo de vida curto para ele é eminente.

Vindo de encontro a isto está a velocidade de entrega, o *Time-to-Market*. É senso comum que testar exaustivamente e corretamente o software não é uma tarefa trivial e exige um esforço considerável. Se mal aplicado e planejado, os esforços para testar o software podem demorar até mais do que a própria construção, tirando um pouco do sentido dos cálculos de prazo e custo do projeto.

Falando um pouco de planejamento e gestão moderna de desenvolvimento de software, e levando em consideração o conceito que podemos chamar de Definição de Pronto, o teste de cada pedaço do software poderia ser parte de sua construção, e não uma etapa após ela. Este ponto de vista torna-se complexo quando o software vai crescendo à medida que é construído, pois testes como os de regressão vão ficando cada vez maiores e mais custosos.

O Visual Studio Lab Management foi elaborado pensando em toda esta cadeia de complexidade que é garantir a qualidade do seu produto. Seu principal objetivo é apoiar os testadores na criação e gestão de ambientes virtuais para testes, além automatizar as tarefas repetitivas do âmbito de qualidade de software. Em breve serão

apresentados mais detalhes sobre esse serviço.

Um caso real do poder da qualidade de software

Há um exemplo de uma situação real sobre os já comuns aplicativos de taxi. Todos sabemos da variedade de opções que temos hoje. Entretanto, esta história se passou em meados de 2012, e os aplicativos eram o assunto do momento. Dois aplicativos de táxi acabavam de ser lançados e havia uma concorrência forte entre eles pela adesão de clientes e taxistas.

Vamos chamar os aplicativos de **TaxiApp A** e **TaxiApp B**. Minha preferência inicial foi pelo TaxiApp A por questões de compatibilidade, velocidade do App e disponibilidade de taxistas. E tudo isso foi se confirmado ao longo do tempo. As corridas do dia a dia tornaram-se muito mais fáceis do que a clássica chamada por táxis através de cooperativas e o pagamento por meio de boletos da empresa.

Entretanto, em um dia em que tinha uma reunião crítica em um cliente no extremo sul de São Paulo e a agenda estava apertadíssima, o aplicativo sofreu uma atualização no período da manhã. No horário da saída do meu escritório para a reunião, não deu outra: o aplicativo travava e fechava de forma inesperada. Não havia jeito de contornar o problema. Chamar um táxi por ele tornou-se impossível. Naquele momento, eu não tinha o telefone da cooperativa fácil. Tudo conspirou contra.

Um amigo que me acompanhava para a reunião me lembrou do TaxiApp B. Com toda a pressa que tinha, baixei-o da loja de aplicativos na hora. Cadastrei-me e consegui pedir o táxi tranquilamente.

A experiência dos dois aplicativos agora era bem parecida. As opções já eram semelhantes e fiquei satisfeito da mesma forma. O

grande recado desta história é que, depois deste dia, eu nunca mais voltei a usar o TaxiApp A.

Sim, eu me mantive no aplicativo que me atendeu pela última vez. Para eu voltar ao TaxiApp A, teria de ficar acompanhando para conferir se havia alguma nova atualização e, caso acontecesse, precisaria testar para ver se o problema foi resolvido. Muito mais simples continuar com o novo aplicativo que já estava funcionando.

Em resumo: o que será que causou o *crash* no TaxiApp A? Será que algum defeito foi injetado naquela atualização da manhã? E por que foi injetado? Será que faltou testar o aplicativo em um cenário parecido com o meu, no que diz respeito a hardware e software? Nós nunca saberemos. A única certeza é de que: por um problema que talvez tenha sido bobo, um cliente foi perdido e nunca mais voltou.

E isto pode acontecer com o seu produto. Concorrentes hoje não faltam quando falamos de software. Um passo errado e seu cliente pode ir para o concorrente em um piscar de olhos.

Os testes de regressão

Todo profissional de qualidade de software sabe da necessidade dos Testes de Regressão em cada entrega de um produto. É por ele que é possível garantir que a compatibilidade foi mantida, que funcionalidades preexistentes não foram impactadas por novas implementações e que, de um modo geral, o produto continua atendendo à necessidade de negócio.

Teste de Recessão, em resumo, significa retestar todo o sistema toda vez que ele for alterado. É notório que esta atividade pode ser muito simples nas primeiras entregas. Entretanto, torna-se um trabalho muito árduo e custoso ao longo da vida do projeto, podendo torná-la inclusive inviável.

Deixar de executar os Testes de Regressão é praticamente abrir mão da qualidade geral do seu produto. Testar somente o que foi recém-implementado não garante a qualidade geral do software.

Visão financeira dos Testes de Regressão manuais e automatizados

Execução de Testes Regressivos manualmente

Na tabela a seguir, veja um exemplo de cálculo de custo dos Testes de Regressão manuais ao longo do ciclo de vida do projeto de 6 meses de duração.

Mês	Funcionalidades Totais	Horas de Teste por Funcionalidade	Custo da Hora de Teste	Custo Total da Qualidade
Janeiro	5	2	R\$ 120.00	R\$ 1,200.00
Fevereiro	12	2	R\$ 120.00	R\$ 2,880.00
Março	20	3	R\$ 120.00	R\$ 7,200.00
Abril	30	3	R\$ 120.00	R\$ 10,800.00
Maio	35	4	R\$ 120.00	R\$ 16,800.00
Junho	44	5	R\$ 120.00	R\$ 26,400.00
Total				R\$ 65,280.00

Figura 5.12: Exemplo de custos de Teste de Regressão com técnicas manuais

Observações importantes:

- *Funcionalidades Totais* significa a quantidade total de funcionalidades entregues no produto no dado mês. Em junho, por exemplo, o sistema foi entregue com 44 funcionalidades.
- Por que as *Horas de Teste por Funcionalidade* foram aumentando? Seguindo à risca o conceito de Teste de Regressão, ao encontrarmos um novo bug durante uma rodada de testes, este deve ser reportado e, após corrigido, a rodada deve começar novamente. Testar a partir de onde parou não caracteriza Teste Regressivo.

Com isto, à medida que o sistema vai crescendo, a tendência é que mais bugs sejam encontrados, e mais tempo seja necessário para fechar a execução de uma rodada de testes.

- O *Custo Total da Qualidade* é o cálculo da multiplicação entre *Funcionalidades Totais*, *Horas de Teste por Funcionalidade* e *Custo da Hora de Teste*. Vale ressaltar que os outros custos operacionais, como custo da hora do desenvolvimento, ambientes, entre outros não estão considerados aqui. Por isto, esta coluna está nomeada como *Custo da Qualidade*, somente.

Automação de Testes Regressivos

Levando em consideração a possibilidade que as ferramentas atuais trazem para automação de testes e os custos para executar as atividades de Teste de Regressão manualmente como vimos anteriormente, torna-se viável considerar trabalhos de automação de testes de interface para que estes venham a compor os Testes de Regressão em cada entrega. Para o cálculo a seguir, vamos considerar a mesma velocidade de entrega de cada mês.

Mês	Funcionalidades a Automatizar	Horas de Teste por Funcionalidade	Custo da Hora de Teste	Custo Total da Qualidade
Janeiro	5	4	R\$ 160.00	R\$ 3,200.00
Fevereiro	7	4	R\$ 160.00	R\$ 4,480.00
Março	8	4	R\$ 160.00	R\$ 5,120.00
Abril	10	5	R\$ 160.00	R\$ 8,000.00
Maio	5	5	R\$ 160.00	R\$ 4,000.00
Junho	9	6	R\$ 160.00	R\$ 8,640.00
Total				R\$ 33,440.00

Figura 5.13: Exemplo de custos do Teste de Regressão com técnicas de automação

Para este raciocínio, é bom salientar:

- *Funcionalidades a Automatizar* são as funcionalidades novas da entrega que ainda não têm automação

associada. Para os demais meses, não é necessário automatizar o que já foi entregue.

- Em *Horas de Automação por Funcionalidade*, considera-se que o esforço inicial é maior do que uma execução manual em 100%. Entretanto, ao longo do tempo, este esforço vai se equalizando com a execução manual devido ao que já foi relatado anteriormente.
- Assume-se que o *Custo da Hora de Automação* é naturalmente maior que o custo de uma execução manual de teste. A razão considerada foi de cerca de 35%.
- Nota-se um aumento no *Custo Total da Qualidade*, de janeiro a junho, de 170%. Parece alto, mas de acordo com o mesmo raciocínio para testes manuais, o aumento é de 2.100%. Além disso, o aumento parece mais linear, previsível e gerenciável do que o custo total com testes manuais.

Nos exemplos anteriormente, em resumo, tivemos um *Custo Total de Qualidade com Testes de Regressão* manuais de R\$ 65.280,00. Por outro lado, o mesmo projeto com esforços de Testes de Regressão voltados para automação, o custo foi de R\$ 33.440,00. Nota-se então que, por meio da automação dos Testes Regressivos, pode-se gerar uma economia de aproximadamente 49% no custo da qualidade do produto.

A longevidade dos Testes Regressivos automatizados

Além do fator financeiro, é importante ter em mente que os Testes de Regressão que forem automatizados durante o ciclo de desenvolvimento do produto serão úteis até o momento em que o produto não for mais usado.

Isto é possível porque a automação, falando de *Team*

Foundation Server e *Lab Management*, é código que pode ser mantido e versionado no mesmo repositório que o seu produto. Pequenas manutenções podem ser necessárias devido às evoluções das funcionalidades, mas o esforço não pode ser comparado com uma nova execução de testes completos de forma manual.

Como exemplo, pode-se pensar em um cenário de um produto estável, o qual, após alguns meses de estabilidade em produção, precisa de uma mudança razoável em seu funcionamento para atender a uma nova legislação. O corpo de testadores que trabalham na sustentação do produto já não é mais o mesmo do ciclo de desenvolvimento, e a rotatividade fez com que os poucos que sobraram conhecessem pouco do sistema de ponta a ponta.

A execução manual dos Testes de Regressão torna-se então muito arriscada e custosa, podendo levar esta etapa à inviabilização. Se os Testes de Regressão automatizados forem levados em consideração, basta acompanhar sua execução e avaliar os resultados, que podem levar a algumas atividades extras devido a alguma anomalia encontrada.

Por último, é importante ter em mente que as automações codificadas para os Testes de Regressão podem também ser reaproveitadas para fins de monitoramento e/ou Testes de Fumaça, por exemplo.

O Visual Studio Lab Management

Como foi possível entender no final da introdução, o Visual Studio Lab Management faz parte da suíte de Application Lifecycle Management, o Visual Studio Team Foundation Server, e é uma ferramenta criada pela Microsoft para apoiar os testadores na criação e gestão de ambientes virtuais de teste. Por ele, torna-se possível automatizar as atividades de provisionamento de ambientes propícios para cada cenário de testes do software, além da execução

do teste em si.

Com ele, é possível viabilizar o provisionamento dinâmico de ambientes, os chamados *SCVMM Environments*, de acordo com a necessidade do momento. Depois do uso, ele libera os recursos para que possam ser usados para outras necessidades.

Para cenários onde o ambiente gerenciado pelo Lab Management é o Server-Side, ele permite que dados de diagnóstico sejam facilmente coletados, enquanto os testes são executados da máquina do próprio testador. Como exemplo, trechos de logs do Visualizador de Eventos do Windows ou dados de Intellitrace podem ser capturados. Se um bug é criado a partir do teste em execução, estes dados colhidos são facilmente anexados a ele, tornando a depuração por parte do desenvolvedor mais fácil e prática.

Além do uso inteligente dos recursos de hardware e software, o Lab Management torna possível testar o software em ambientes sempre livres de qualquer fator que possa influenciar o resultado do teste. Isto é possível porque, como ele cria ambientes de teste sempre que necessário, este pode ser baseado em uma imagem de sistema operacional e softwares previamente instalados, os chamados snapshots, uma captura do sistema operacional no estado atual.

Outra grande vantagem é a integração com o Team Foundation Server e o Visual Studio Test Manager. Por meio dele, pode-se classificar cada imagem de ambiente com algumas categorias como Sistema Operacional e softwares instalados. Com isto, sempre que um novo teste for executado, são levados em consideração os ambientes nos quais aquele teste precisa ser executado.

Por exemplo, em um cenário onde o teste é executado dentro do ambiente e este precise ser compatível com Windows 7, Windows 8

e Internet Explorer 10, e tem-se imagens com estes Sistemas Operacionais e Navegadores, o Lab Management vai se encarregar de provisionar as máquinas necessárias usando estas imagens, disponibilizando-as para a execução dos testes.

Ambientes estáticos também podem ser gerenciados e usados para testes com o Lab Management. Isto quer dizer que, se não houver uma instalação do System Center Virtual Machine Manager no ambiente, máquinas instaladas manualmente e definidas como ambientes de teste pelo time podem ser gerenciadas pela ferramenta e usadas para automação de testes. Estes ambientes são chamados de *Standard Environments*.

SCVMM Environments x Standard Environments

Como foi visto há pouco, o Lab Management suporta trabalhar com ambientes dinâmicos (SCVMM) ou estáticos (Standard). Na tabela a seguir, são listadas as principais diferenças entre os dois tipos de ambiente:

Funcionalidade	Ambientes SCVMM	Ambientes Standard
Testes	-	-
Executar testes manuais	Suportado	Suportado
Executar testes Coded UI e outros testes automatizados	Suportado	Suportado
Criar bugs ricos em detalhes usando os adaptadores de diagnóstico	Suportado	Suportado
Deployment de pacotes	-	-
Workflows automáticos de build-publicação-teste	Suportado	Suportado
Gestão e criação de ambientes	-	-
Usar máquinas físicas além de máquinas virtuais	Não Suportado	Suportado
Usar máquinas virtuais de tecnologias de terceiros	Não Suportado	Suportado
Instalar o Test Agent automaticamente em máquinas no ambiente de Lab	Suportado	Suportado
Salvar e provisionar o estado de um ambiente de Lab usando os snapshots de ambiente	Suportado	Suportado
Criar ambientes de Lab a partir de Templates de Máquinas Virtuais	Suportado	Não Suportado
Iniciar/Parar/Tirar snapshot de ambientes	Suportado	Não Suportado
Conectar-se ao ambiente usando o Environment Viewer	Suportado	Suportado
Executar múltiplas cópias de um ambiente simultaneamente usando o isolamento de rede	Suportado	Não Suportado

Figura 5.14: Fonte: MSDN - Using a Lab Environment for Your Application Lifecycle (<https://msdn.microsoft.com/en-us/library/dd997438.aspx>)

Esta tabela é importante para dar a ciência do que pode e do que não pode ser feito com cada um dos tipos de ambiente.

Configuração do Lab Management

O processo de configuração do Lab Management para Standard Environments envolve três etapas: a configuração do Team Foundation Server, a instalação do Test Controller e o setup dos Test Agents. Vamos entender cada um deles.

- **Team Foundation Server**

O Team Foundation Server, como para todas as outras ferramentas da suíte de ALM, é o ponto central de gestão de ferramentas, ambientes e dados. É nele que se conectam os Test Controllers e também ficam armazenados os dados de associação entre Planos, Suítes e Casos de Testes com ambientes de Lab.

Se a sua configuração for direcionada para SCVMM Environments, é necessário que o System Center Virtual Machine Manager Administration Console seja instalado no Application Tier. Caso contrário, veremos à frente que basta o Test Controller se conectar a ele.

- **Test Controller**

Este componente da instalação é o responsável por orquestrar os trabalhos do Lab Management. É ele quem direciona os testes a serem executados para os Test Agents corretos.

Se configurado para execução e automação de testes, um Test Controller deve ser associado a uma, e somente uma, Team Project Collection do TFS. É aqui o único ponto de associação entre TFS e Lab Management quando falamos de Standard Environments.

- **Test Agent**

É o host responsável pela execução dos testes ou, em ambientes SCVMM, o Deployment Stage - o ambiente, de fato. São os Test Agents que são classificados de acordo com seu Sistema Operacional, Browser, entre outros softwares que podem estar instalados como pré-requisitos.

Os Test Agents são comumente provisionados por meio do Microsoft Test Manager, durante a configuração de um novo ambiente. Os Test Agents podem ser instalados manualmente e, posteriormente, adicionados como ambiente também.

Lab Management e os Testes de Regressão

É pelo Lab Management integrado ao TFS que é possível automatizar os Testes de Regressão codificados, acompanhando sua execução e analisando seus resultados. A responsabilidade do Team Foundation Server nesta cadeia é a de prover repositório de versionamento, a interface para associação entre códigos automatizados e os Casos de Teste a serem automatizados e a estrutura de build, que é basicamente a ponte com o Lab Management.

O Lab Management, por sua vez, é responsável por receber as requisições de execução de testes, fazê-las e prover feedback de cada uma delas de volta para o Test Manager e/ou o Build que o disparou.

A importância do Team Build

O Team Build é uma das mais poderosas ferramentas da suíte Visual Studio, pois é capaz de realizar muitas atividades por meio

dos seus customizáveis templates de Build. Nativamente, o build possui alguns templates padrão. O mais conhecido deles, o `DefaultTemplate.xaml`, já é capaz de compilar, testar unidades e fazer análise estática do seu código. A grande sacada é o `LabDefaultTemplate.xaml`, que faz a integração com o Lab Management e também já vem junto com o Team Build.

Basicamente, o que este template faz é “amarrar todas as 3 pontas” para fazer a automação do Teste Regressivo acontecer, pois nele definimos:

- Qual build será utilizado para gerar o pacote de binários;
- Quais os Test Plans/Test Suites serão executados;
- Armazena o resultado da execução dos testes.

Tendo isto, é possível usar as características do Team Build para automatizar completamente seus Testes Regressivos, como:

- Agendamento de builds de Testes de Regressão;
- Disparo de um build de Teste de Regressão de acordo com uma condição de mudanças nos códigos-fonte;
- Entre outros.

Os resultados destes testes também podem ser visualizados em detalhes na interface do Test Manager. Por conta do disparo dos testes pelo build, torna-se muito mais fácil, por exemplo, analisar os resultados e saber qual versão foi testada, porque na prática o pacote gerado no build é uma versão.

Com toda esta integração e as possibilidades anteriores descritas, o racional financeiro apresentado na figura *Exemplo de custos do Teste de Regressão com técnicas de automação* torna-se tranquilamente possível e, mais importante de tudo, profissionaliza a gestão do capital intelectual dos produtos desenvolvidos.

A nova geração do Lab Management

Vale ressaltar que, no momento em que este capítulo é escrito, um novo mecanismo de Lab Management já está em evolução, o Test Hub. Esse mecanismo irá permitir manutenção dos laboratórios de teste via experiência web.

O Team Build vNext é o mecanismo de build do TFS totalmente repaginado, reescrito. O que antes víamos ser baseado em workflows XAML agora tem um motor muito mais enxuto e baseado em JavaScript e PowerShell.

Além disso, o Build vNext tornou-se multiplataforma, sendo capaz de trabalhar com tecnologias como iOS, Android, Java, entre outros. Dadas as tecnologias nas quais ele foi escrito, a sua customização tornou-se muito mais fácil. Para saber mais sobre o novo Team Build, este link é a referência: <https://msdn.microsoft.com/en-us/Library/vs/alm/Build/feature-overview>.

No Build vNext, não há necessidade de um Test Controller vinculado à Team Project Collection (para certos tipos de testes). Uma atividade de build é capaz de fazer o deploy do Test Agent direto em uma máquina e usá-la para executar os testes devidamente elencados.

REFERÊNCIAS

Where do I get the test controller and test agents? -
<https://msdn.microsoft.com/en-us/library/vs/alm/test/test-machines/install-configure-test-agents>

5.4 CONCLUSÃO

Ter o software em produção com qualidade é muito importante, assim como saber identificar quais as funcionalidades estão sendo entregues nos ambientes da esteira de projetos. Neste capítulo, foram demonstradas formas de identificar e ter rastreabilidade entre o código que está sendo desenvolvido e o que foi entregue em produção, além de técnicas para garantir a qualidade dos entregáveis.

Partindo do princípio de que toda regra tem exceção, nem todas as recomendações são obrigatórias, mas sim, sugestões que podem funcionar para determinada situação ou projeto. No último capítulo deste livro, são apresentadas boas práticas que podem ajudar a garantir qualidade no ciclo de vida no desenvolvimento software e evitar problemas comumente encontrados.

CAPÍTULO 6

BOAS PRÁTICAS

Tudo tem um padrão, seja em desenvolvimento de software, gestão de projetos, demandas etc. Mas como ter certeza do que é melhor para seu projeto, ou até mesmo garantir quais são as melhores formas de executar algo?

Muitas vezes o que funciona para um projeto, pode não funcionar para outro devido às suas particularidades. Dessa forma, são criados alguns padrões ou boas práticas para garantir uma linha de pensamento homogênea entre os envolvidos do processo para garantir que não haverá muita divergência no que é entregue.

Neste capítulo, serão apresentadas algumas boas práticas baseadas na experiência dos engenheiros que lhe escrevem sobre como melhorar a qualidade das entregas no ciclo de desenvolvimento de software.

6.1 INVISTA EM REVISÃO DE CÓDIGO

Por Luís Henrique Demetrio

“Não tenho tempo, orçamento ou não está na minha prioridade são geralmente as desculpas para não investir na revisão de código durante a fase de desenvolvimento de software”.

O ideal é que o ciclo de vida do desenvolvimento de software contemple a revisão de código em seu ciclo, visando aumentar a

qualidade. Quanto antes uma falha for identificada, menor será o custo de desenvolvimento do projeto (MYERS, 2015).

Atualmente, a ferramenta de desenvolvimento Microsoft Visual Studio 2015 permite executar a revisão de código por meio das regras do Microsoft Code Analysis. Além da execução manual, também é possível configurar o Microsoft Team Foundation Server (TFS) para automatizar a execução da revisão de código durante o build.

A figura a seguir exibe as diferentes formas de executar manualmente o Code Analysis no Visual Studio 2015:

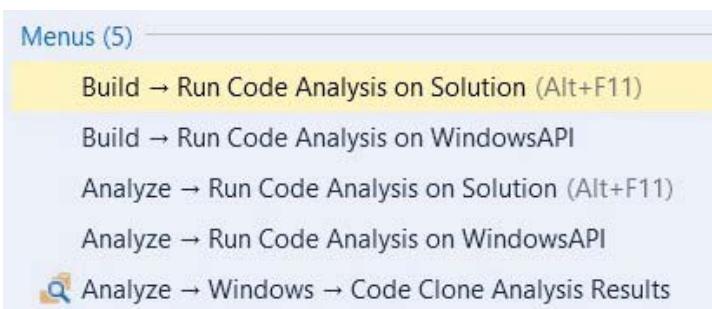


Figura 6.1: Diversas formas de executar o Code Analysis

As próximas seções descrevem os problemas mais comuns que podem ser identificados pela revisão de código via Code Analysis.

Não liberação de recursos

Esse é um problema muito comum, sendo os dois principais sintomas o consumo excessivo de memória e de conexões com o banco de dados. O resultado, além do consumo excessivo de recursos, nesse caso memória e conexões, pode resultar no esgotamento de memória e de conexões com o banco de dados.

Os erros mais comuns nesses casos são:

- `System.OutOfMemoryException`
- `System.InvalidOperationException`

Esse tipo de violação pode ser identificado, por exemplo, através da regra CA2213 e CA2000. As regras citadas prezam respectivamente que: “Objetos que implementem a interface `IDisposable` devem ser descartados” e “Descartar objetos antes de que eles percam o seu escopo”. O fragmento de código a seguir fornece um exemplo da violação da regra CA2000, onde o método `Dispose` do objeto de conexão não é executado apesar do objeto perder o escopo.

```
private void GetData()
{
    string connectionString = string.Empty;
    SqlConnection connection =
        new SqlConnection(connectionString);
    connection.Open();
    //Do work here;
}
```

Apesar do objeto de conexão perder o escopo e estar apto a ser coletado pelo Garbage Collector, a liberação da conexão com o banco de dados só será liberada quando o Garbage Collector for executado, o que não é determinístico. Observe pela figura a seguir que a violação dessa regra é identificada durante a compilação do projeto.

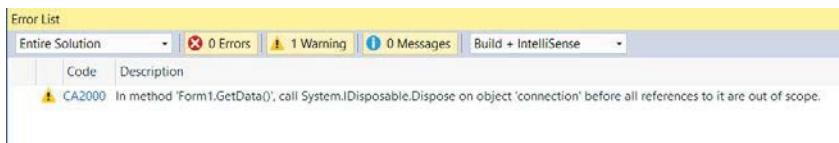


Figura 6.2: Violação CA2000 identificada durante a compilação do Projeto

Tratamento de erros inadequado

O tratamento de erros inadequado pode resultar na degradação

de desempenho e dificultar a rastreabilidade das exceções. O resultado é no aumento de processamento e em um maior esforço para identificar a causa raiz do erro.

O fragmento de código a seguir fornece um exemplo de um tratamento de erro inadequado. Observe que uma nova exceção é gerada, e que a exceção original não é passada como parâmetro. O resultado é consumo de processamento e perda da pilha de exceção original.

```
void CatchAndThrowNewException() {
    try
    {
        Method1();
        Method2();
        Method3();
    }
    catch (ArithmetricException e)
    {
        throw new Exception(e.Message); // Violates the rule.
    }
}
```

A figura a seguir exibe a pilha de execução gerada pela exceção anterior. Observe que a pilha de exceção mostra que o erro ocorreu no método `CatchAndThrowNewException`, sendo impossível identificar em qual parte do método o erro foi gerado.

Throw new exception specified:

```
at DemoApp.Demos.TestsRethrow.CatchAndThrowNewException()
at DemoApp.Demos.TestsRethrow.CatchException()
```

Figura 6.3: Pilha de execução gerada através do uso do `throw new Exception`

O mesmo problema quando a exceção é gerada conforme o fragmento de código a seguir.

```
void CatchAndRethrowExplicitly()
{
    try
```

```
{  
    Method1();  
    Method2();  
    Method3();  
}  
catch (ArithmetricException e) {  
    throw e; // Violates the rule.  
}  
}  
}
```

Para satisfazer a regra, o correto é gerar a exceção da seguinte forma:

```
void CatchAndRethrowImplicitly() {  
    try {  
        Method1();  
        Method2();  
        Method3();  
    }  
    catch (ArithmetricException e) {  
        throw; // Satisfies the rule.  
    }  
}
```

Entretanto, como não existe tratamento algum no bloco de captura de erro, o bloco `try/catch` nesse caso se torna desnecessário. O correto seria conforme ilustrado no fragmento a seguir:

```
void NoTryCatch()  
{  
    Method1();  
    Method2();  
    Method3();  
}
```

Para esses últimos dois exemplos, o resultado da pilha de execução seria conforme ilustrado pela figura a seguir. Observe que é possível identificar exatamente onde a exceção é gerada.

Implicitly specified (throw):

```
at DemoApp.Demos.TestsRethrow.ThrowException()
at DemoApp.Demos.TestsRethrow.Method2()
at DemoApp.Demos.TestsRethrow.CatchAndRethrowImplicitly()
at DemoApp.Demos.TestsRethrow.CatchException()
```

Figura 6.4: Pilha de execução complete

As seguintes regras do Code Analysis são referentes ao tratamento de exceções:

- CA1032 - Implementar construtores de exceção padrão
<https://msdn.microsoft.com/en-us/library/ms182151.aspx>
- CA1064 - As exceções devem ser públicas
<https://msdn.microsoft.com/en-us/library/bb264484.aspx>
- CA1065 - Não acione exceções em locais inesperados
<https://msdn.microsoft.com/en-us/library/bb386039.aspx>
- CA2102 - Capturar exceções que não sejam CLSCompliant em manipuladores gerais
<https://msdn.microsoft.com/en-us/library/bb264489.aspx>
- CA2139 - Os métodos transparentes talvez não usem o atributo HandleProcessCorruptingExceptions
<https://msdn.microsoft.com/en-us/library/dd997565.aspx>
- CA2201 - Não acionar tipos de exceção reservados
<https://msdn.microsoft.com/en-us/library/ms182338.aspx>
- CA2208 - Instanciar exceções de argumento corretamente
<https://msdn.microsoft.com/en-us/library/ms182347.aspx>

- CA2219 - Não acione exceções em cláusulas de exceção
<https://msdn.microsoft.com/en-us/library/bb386041.aspx>

6.2 EVITE A CODIFICAÇÃO DE MÉTODOS COMPLEXOS

Por Luís Henrique Demetrio

“Há uma função na minha aplicação conhecida como função faz tudo. Ela verifica as informações do cliente, adiciona o pedido no carrinho de compras, valida o CEP, o cartão de crédito etc. Como posso melhorar meu código?”

A criação de métodos complexos dificulta o entendimento dos passos que seu algoritmo deve seguir para resolver um problema. Assim, se tem como consequência o aumento do esforço de manutenção e de testes do código e, tendo como possível resultado, o aumento na inserção de bugs durante as modificações.

No livro *Clean Code*, de Robert C Martin, ele nos propõe que a primeira regra para a criação de um método ou função é que esta seja pequena. Mas o que podemos considerar como “uma função pequena?”. Dada a dificuldade em obter tal resposta de forma exata, podemos colocar alguns itens a serem considerados, como por exemplo, o que o algoritmo deve fazer? O código precisa ser dividido em outras funções para reduzir a complexidade? O código é de fácil leitura?

Dado o cenário anterior, se faz necessário empregar ferramentas que nos auxilie durante o processo de criação de novos projetos, tendo como objetivo manter nosso código simples no que diz respeito à complexidade, ao acoplamento e à profundidade de herança, conceitos que debateremos a seguir.

O Microsoft Code Metrics nos fornece uma série de indicadores que nos ajuda a avaliar os itens citados anteriormente, refletindo diretamente na qualidade do produto final: nosso código.

Complexidade ciclomática

“Vejo que meus sistemas crescem e ficam difíceis de manter e testar. Como posso criar softwares que permitam testes e tenham simples manutenção?”

A resposta ao questionamento anterior pode ter sua resposta direcionada por uma técnica proposta por Thomas J. McCabe em seu artigo intitulado *A complexity Measure*, definindo uma nova forma para o cálculo de complexidade de algoritmos baseada em teoria dos grafos.

O ponto fundamental desta técnica é a consideração da quantidade de caminhos possíveis que um código pode ter durante sua execução, o que vai influenciar diretamente no número de testes necessários para testar todas as possibilidades de execução em um algoritmo. O fragmento de código a seguir fornece um exemplo da complexidade ciclomática para dois métodos:

```
// complexidade ciclomática 1
public void Method()
{
    Console.WriteLine("Hello World!");
}
// complexidade ciclomática 2
void Method2(bool condition)
{
    if (condition)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Observe que o método `Method2` faz uso de uma condicional (`if`) que possibilita dois caminhos possíveis de execução. Em

decorrência disso, esse método é classificado como sendo de complexidade ciclomática 2, onde é necessário realizar dois testes unitários para cobrir cada caminho possível através de um valor verdadeiro, e outro com um valor falso.

Mas para os exemplos anteriores, a determinação da complexidade de ambos os métodos é simples. E como endereçar algoritmos mais complexos? Como automatizar quando meu código ser tornar razoavelmente extenso?

O Microsoft Code Analysis, ferramenta incorporada ao Microsoft Visual Studio, possui um conjunto de regras para avaliação de boas práticas de codificação. Uma das regras existentes é a regra CA1502, que tem como objetivo identificar o nível de complexidade ciclomática no código. Esta regra preza que a complexidade ciclomática de um método seja menor do que 25 para que seja não seja considerado demasiadamente complexo.

Observe que, para um código cuja complexidade ciclomática atinja o limite aceito nesta regra, será necessária a implementação de 25 testes unitários para que se cubram todos os possíveis caminhos de execução do fluxo em questão. Dessa forma, podemos dizer que a complexidade ciclomática está diretamente relacionada ao esforço empregado na manutenção do código. Observe que, para um código cuja complexidade ciclomática atinja o limite aceito nesta regra, será necessária a implementação de 25 testes unitários para que se cubram todos os possíveis caminhos de execução do fluxo em questão. Dessa forma, podemos dizer que a complexidade ciclomática está diretamente relacionada ao esforço empregado na manutenção do código. Para maiores detalhes sobre o Code Analysis, consulte a seção *Qualidade de código*, no capítulo 4.

Evite métodos longos

Ao basear uma análise na quantidade de linhas dispostas em um

método, este cálculo é realizado pela linguagem intermediária (IL). Uma quantidade elevada de código pode indicar uma sobrecarga de responsabilidades atribuídas ao método ou à classe. Como citado anteriormente, a recomendação é que tenhamos métodos ou funções pequenas.

Sempre que possível, a recomendação nesse caso é quebrar o código em partes menores visando melhorar a manutenção, mas sempre considerando a legibilidade do código. Grandes quebras podem trazer dificuldades de leitura.

Via de regra, os métodos com muitas linhas de código e com uma alta complexidade ciclomática tendem a ter menor confiabilidade.

Acoplamento e Coesão excessivos

Duas características desejadas no desenvolvimento de software são: escalabilidade e manutenção, e ambas estão relacionadas aos conceitos de Coesão e Acoplamento. Estes princípios foram originalmente introduzidos por Constantine e Yourdon, em sua obra *Structured Design* (1976).

A definição dada por Sposito e Saltarello, em *Microsoft .NET* (2008), para Coesão é: “*Coesão indica que um determinado módulo de um software, seja ele uma sub-rotina, classe ou biblioteca, possui um conjunto de responsabilidades que estão fortemente relacionados*”. Os mesmos autores se referem a acoplamento como sendo: “*Acoplamento é o nível de dependência existente entre dois módulos de um software, como por exemplo, duas classes*”.

O elevado nível de acoplamento dificulta a reutilização e a manutenção decorrente de suas muitas interdependências para outros tipos, enquanto o alto nível de coesão elimina sequências complexas de interações entre vários componentes por serem

logicamente agrupadas de forma correta

Novamente o Visual Studio pode ser empregado para validar a qualidade do código. Há uma regra específica, a CA1506 (“Evitar acoplamento excessivo de classes”), do Code Analysis, responsável em medir o acoplamento de classes através de parâmetros, variáveis locais, tipos de retorno, chamadas de método, genérico ou modelo instanciações, classes base, implementações de interface, campos definidos em tipos externos e decoração atributo.

O baixo acoplamento pode ser conquistado pelo uso das seguintes técnicas:

- **Programação orientada a interfaces:** o uso de interfaces diminui a dependência entre os objetos;
- **Uso de composição ao invés de herança:** as classes devem atingir o comportamento polimórfico e reutilização de código por composição ao invés de o fazer por meio de herança.

Evite o uso excessivo de herança

O uso excessivo de heranças aninhadas dificulta a manutenção e legibilidade do código ao aumentar o acoplamento do código e dificultar modificações, uma vez que uma modificação na classe base pode impactar as classes derivadas. O Code Analysis fornece a regra CA1501, que valida quando uma classe possui mais de 4 níveis na hierarquia de herança. O fragmento de código a seguir mostra um exemplo que viola a regra.

```
namespace MaintainabilityLibrary
{
    class BaseClass {}
    class FirstDerivedClass : BaseClass {}
    class SecondDerivedClass : FirstDerivedClass {}
    class ThirdDerivedClass : SecondDerivedClass {}
    class FourthDerivedClass : ThirdDerivedClass {}
```

```
// This class violates the rule.  
class FifthDerivedClass : FourthDerivedClass {}  
}
```

REFERÊNCIAS

Linguagem intermediária (IL) -
<https://msdn.microsoft.com/pt-br/library/z1zx9t92.aspx>

Software Metrics And Reliability -
<http://academic.research.microsoft.com/Paper/2101790.aspx>

Improving .NET Application Performance and Scalability -
<https://msdn.microsoft.com/en-us/library/ff649152.aspx>

Regra CA1502 - <https://msdn.microsoft.com/pt-br/library/ms182212.aspx>

Regra CA1506 - <https://msdn.microsoft.com/pt-br/library/bb397994.aspx>

Regra CA1501 - <https://msdn.microsoft.com/pt-br/library/ms182213.aspx>

6.3 COMO DEVO ME PREPARAR PARA UM TESTE DE CARGA?

Por Rodrigo Gallazzi Leite

“Essa aplicação será utilizada por pelo menos 45 mil pessoas durante o horário comercial. Não podemos correr o risco de ter problemas de performance, isso pode e vai abalar nossa imagem como provedores desse serviço. Como podemos prever a performance da aplicação?”

Para responder à pergunta em destaque, em cenários como este recomenda-se a execução de testes de carga e testes de stress para aferir a performance de aplicações. Mas, apesar de próximos, ambos os testes têm objetivos distintos: um teste de carga visa simular uma carga e aferir o comportamento da aplicação e do ambiente onde a aplicação está hospedada, enquanto, em um teste de stress, o objetivo é aumentar a carga até que a aplicação deixe de responder.

Os primeiros questionamentos que devemos ter em mente para a realização de qualquer um destes testes são: quais são os objetivos que a área de negócios visa atingir com o teste? Quantos usuários simultâneos minha aplicação tem de suportar? Qual o tempo de resposta esperado com esse número de usuários? Qual o tempo máximo de processamento das requisições?

Com essas perguntas respondidas, podemos ir para o próximo passo. O teste de carga tem como objetivo simular carga sobre o sistema e aferir como a aplicação responde a essa carga. Por exemplo, dada uma empresa de venda de ingressos online, no dia da abertura das vendas de um novo evento, a aplicação terá de suportar uma carga de 50 mil usuários simultaneamente. Sua aplicação está apta para suportar essa carga? Estaria o sistema pronto para atender essa demanda de usuários?

Para que seja executado um teste de carga, é necessário gravar os cenários de teste que se deseja testar. Esses testes serão usados para reproduzir uma operação e simular as requisições sobre o sistema.

Além dos cenários de teste, é necessário validar se a aplicação mantém-se estável, e se todas as suas dependências funcionam corretamente, como por exemplo, componentes externos, pastas de arquivos e permissões de usuários.

Por experiência, recomenda-se que não apenas o time responsável pelo projeto participe de sua execução, mas que outras

equipes de infraestrutura também estejam envolvidas, incluindo administradores de banco de dados, analistas de rede, analistas de segurança, administradores de servidores etc. Assim, caso algum problema gerado pela carga aplicada esteja relacionado à infraestrutura, os times já podem identificá-lo e endereçar sua correção.

Os testes de carga e de stress devem ser o mais próximo possível da realidade. Assim, o resultado obtido com os testes reproduzirá com fidelidade o contexto que será enfrentado nos momentos de maior consumo dos sistemas.

Desta forma, com o objetivo de simular situações próximas a realidade, recomendamos que o ambiente sob teste seja uma cópia do ambiente que hospedará o sistema. Este ambiente, se não for o mesmo ambiente que hospedará a aplicação, deve conter a mesma configuração (ou seja, memória, CPU, disco, sistema operacional), disposição e número de servidores do ambiente de produção.

Ainda assim, para tornar o teste mais próximo da realidade, pode ser necessário executar uma carga no banco de dados para simular não apenas a carga de acessos, mas também um banco de dados volumoso e que represente a realidade do sistema. Para isso, deve-se considerar a geração prévia de uma carga de dados. A geração dessa massa de dados demanda tempo e espaço em disco. É prudente planejá-la com antecedência. Os benefícios de se avaliar também o banco de dados são: detectar gargalos de performance, verificar se os índices estão corretamente aplicados e se a infraestrutura provisionada atende à demanda.

Para avaliar o modo como a aplicação consome o ambiente e como é executada, costuma-se utilizar contadores de performance. Também se recomenda a instrumentação da aplicação com contadores de performance customizados. Os contadores de performance indicam o modo com os servidores são consumidos e

os contadores customizados podem ser adequados as funcionalidades da aplicação que precisam ser rastreadas. Existem diferentes tipos de contadores de performance, cada tipo pode ser adequado a uma situação e cenário diferente a ser medido na análise de performance de uma funcionalidade de um sistema.

REFERÊNCIAS

Contadores customizados - <https://msdn.microsoft.com/en-us/library/ff650681.aspx>

6.4 COMO SIMULAR A CARGA NECESSÁRIA PARA MINHA APLICAÇÃO?

Por Rodrigo Gallazzi Leite

“Como é possível avaliar a quantidade de usuários que minha aplicação suporta com performance aceitável? É imprescindível ter esse número antes de colocarmos a aplicação em ambiente produtivo!”

Há algum tempo, era necessário criar todo um ambiente com agentes de testes dentro da sua rede para gerar a carga de dados necessária para testar uma aplicação. A criação deste ambiente com agentes de testes era um processo que consumia muito tempo e muitos recursos (ou seja, pessoas, hardware, licenças, máquinas).

Após a integração com o Microsoft Azure, o Microsoft Visual Studio tornou-se uma solução completa para execução de testes de carga, além de tornar todo o processo simples e de fácil realização. Isso porque o Microsoft Azure provisiona toda a infraestrutura necessária de agentes de teste dinamicamente. E, da mesma

maneira, toda essa infraestrutura, ao final do teste, é desalocada.

Diante do desafio de criar ambientes com agentes de testes para execução de testes de carga, esse tópico visa demonstrar como é possível criar um ambiente de agentes de teste dinamicamente no Microsoft Azure.

A ligação entre o seu Visual Studio e a alocação de recursos no Microsoft Azure acontece por meio do Visual Studio Team Services (VSTS). É preciso ter uma conta no VSTS para que os agentes sejam alocados dinamicamente para a execução do teste de carga.

Após obtida uma conta de acesso no Visual Studio Team Services, basta abrir o Visual Studio e se conectar ao VSTS na aba *Team Explorer*, ou abrir uma nova instância do Visual Studio pelo website do VSTS (figuras a seguir).

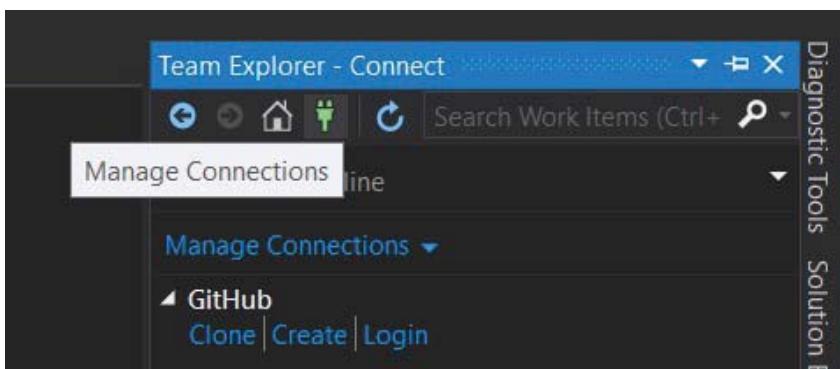


Figura 6.5: Ícone de conexão com o Visual Studio Team Explorer

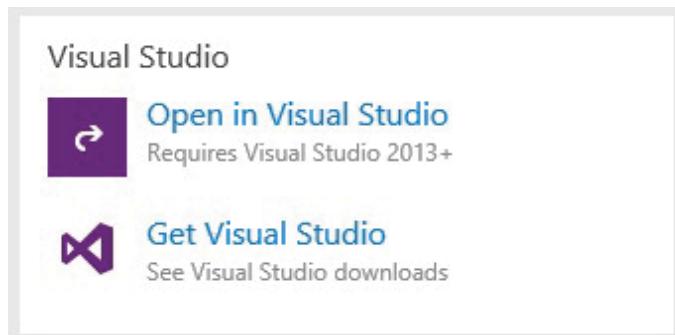


Figura 6.6: Link de abertura do Visual Studio em um projeto criado no Visual Studio Team Services

Depois de conectado ao VSTS, para criar um projeto de teste, é preciso selecionar um novo projeto do tipo *Web Performance and Load Test Project*, como mostra a figura:

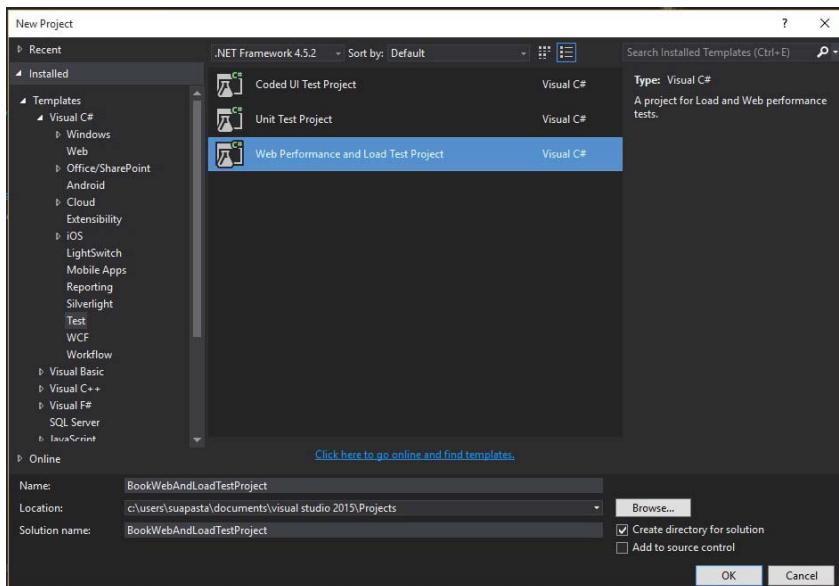


Figura 6.7: Criando projeto de teste no Microsoft Visual Studio 2015

Com o projeto criado, vamos agora gravar nosso primeiro cenário de teste ou webtest. Na criação do projeto de testes, o Visual

Studio cria automaticamente um webtest. Para gravarmos o teste, basta clicar no botão *Adicionar Gravação*, conforme figura adiante. Essa ação fará com que o Visual Studio abra um novo navegador e a cada interação sua com a aplicação, seja um clique de botão ou preenchimento de campo, tudo será gravado pelo Visual Studio como parte do seu cenário de teste.

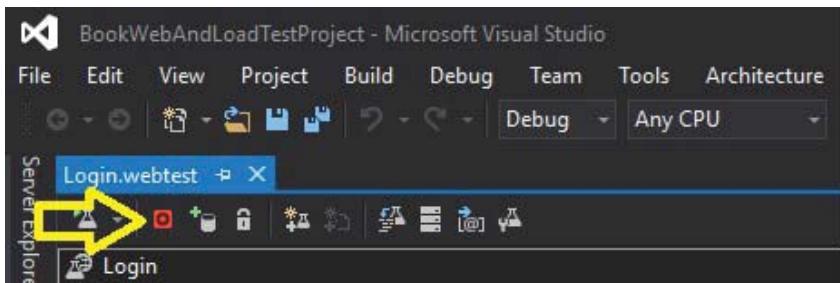


Figura 6.8: Gravando webtest

Na figura a seguir, após interagir com o sistema, podemos verificar que todas as requisições (à esquerda) decorrentes da navegação pela aplicação foram gravadas para reprodução.

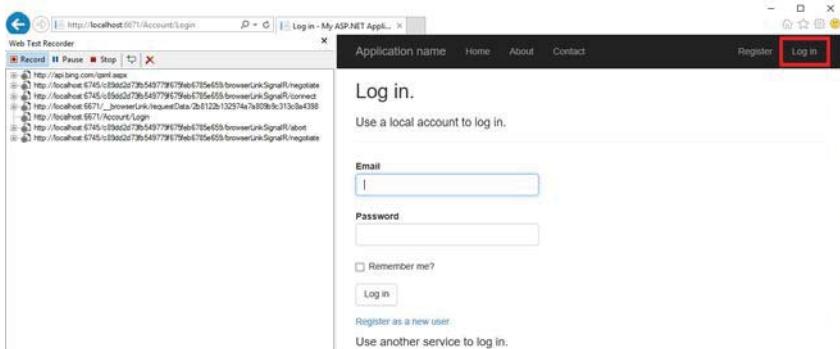


Figura 6.9: Navegador gravando cenário de teste

Ao terminar a gravação do cenário, basta parar a gravação

clicando no botão *Parar* (no menu de gravação dentro de seu navegador). Essa ação fará com que o navegador seja fechado e com que o Visual Studio configure o cenário de teste, conforme a figura:

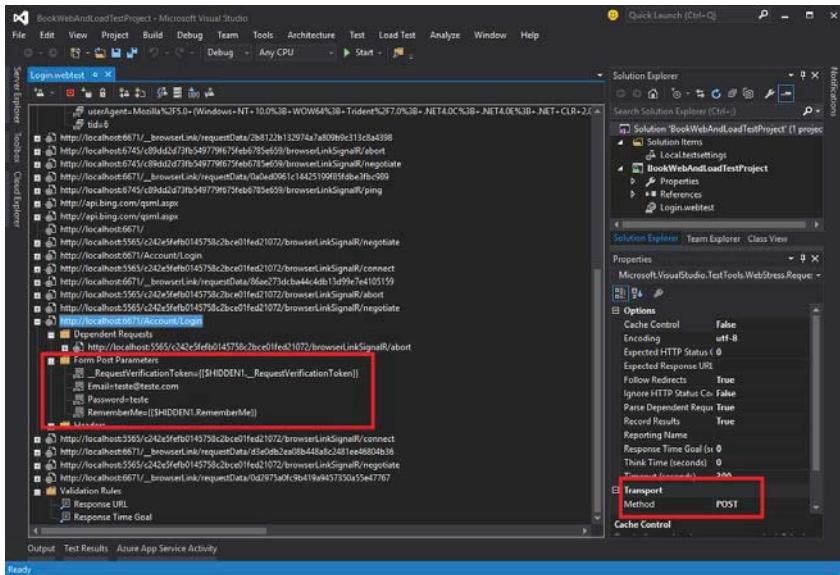


Figura 6.10: Cenário de login gravado

A figura anterior mostra o teste gerado em detalhes. Nesta imagem, podemos ver cada requisição feita pela aplicação (seja ela **GET** ou **POST**), cada parâmetro enviado e cada componente do formulário web.

Se o teste gerado for puramente executado, ele sempre manterá todos os valores utilizados durante a gravação do cenário de teste. É possível parametrizar esses valores conforme a necessidade do sistema. É possível associar uma fonte de dados ao sistema, parametrizando os valores usados durante o teste de carga. A figura seguinte mostra brevemente a seleção de uma fonte de dados para um teste de carga clicando no item de menu *Data Sources*.

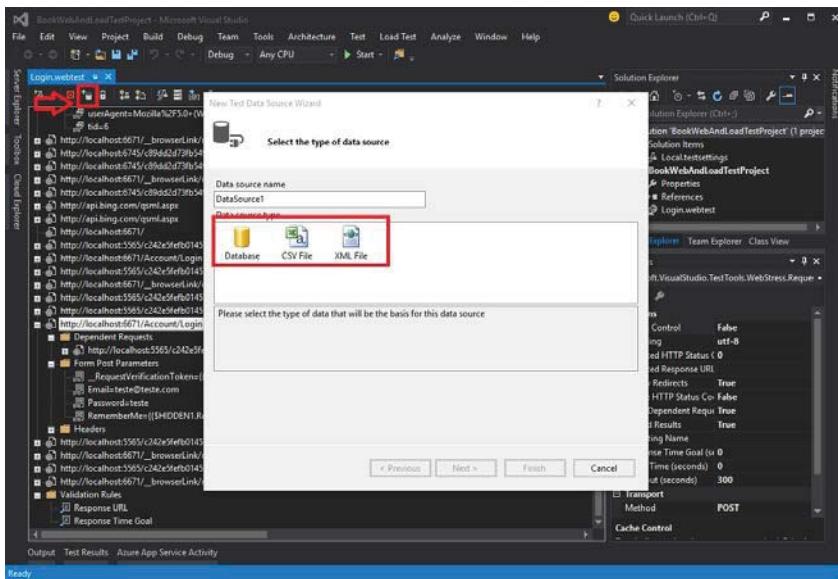


Figura 6.11: Adicionar fonte de dados

Após adicionar a fonte de dados, é hora de fazer o uso desses dados. Note que, quando interagimos com o sistema, a requisição feita foi gravada juntamente com seus parâmetros. Ao expandirmos os parâmetros do post do formulário, podemos ver que os campos Email e Password tem os valores teste@teste.com e senha , respectivamente. Vamos variar esses valores utilizando a fonte de dados.

Para fazer a ligação entre a fonte de dados e os parâmetros, basta clicar no campo e alterar a propriedade `Value` , selecionar a fonte de dados, expandir a tabela e selecionar o campo a ser utilizado, conforme a figura:

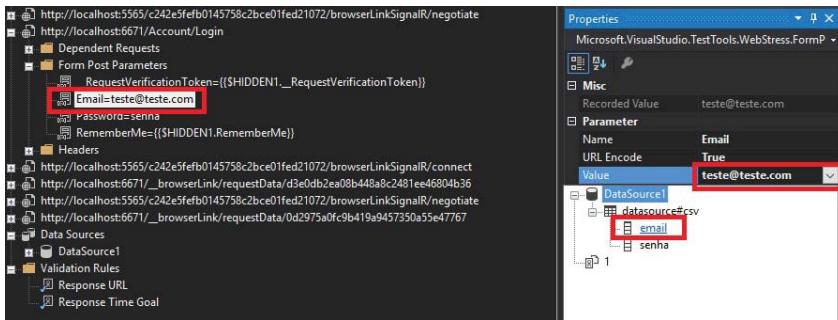


Figura 6.12: Ligação entre campo e fonte de dados

Ao selecionar o campo, o valor do parâmetro deverá ficar no formato indicado na figura:

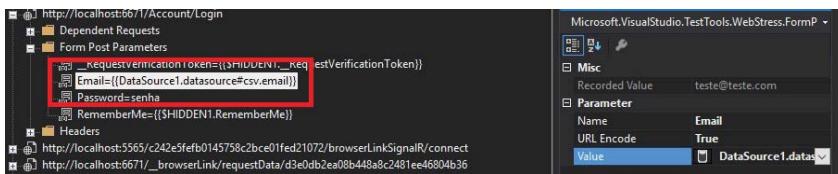


Figura 6.13: Alteração do campo, baseado em fonte de dados

Dessa forma, é possível parametrizar os dados passados nas requisições do webtest de forma simples, o que torna a simulação ainda mais próxima da realidade.

Com o cenário pronto, é hora de criar um componente do projeto chamado teste de carga (*Load Test*). Esse componente terá todas as configurações do seu teste, como por exemplo, duração da execução do teste, número de usuários simultâneos, porcentagem de distribuição de cada cenário de teste e o tipo de rede dos clientes que acessam a aplicação.

Para criar o teste de carga, basta clicar na solução, adicionar um novo item, procurar pelo tipo de projeto Teste e selecionar o item *Load Test*. Essa ação fará com que o Visual Studio abra um configurador do teste de carga.

Na aba *Scenario*, configuraremos o *Think time*. Essa variável simula o tempo que o usuário espera para realizar ações na aplicação, por exemplo: depois de realizado o login no sistema, o usuário leva cinco segundos para verificar qual será a próxima ação.

Na aplicação, cada ação pode gerar uma requisição. O Visual Studio simula essa espera entre as requisições, o chamado “*Think time*”. Para nosso teste, não vamos utilizar essa configuração, então selecionaremos a opção *Do not use think times*, conforme a figura:

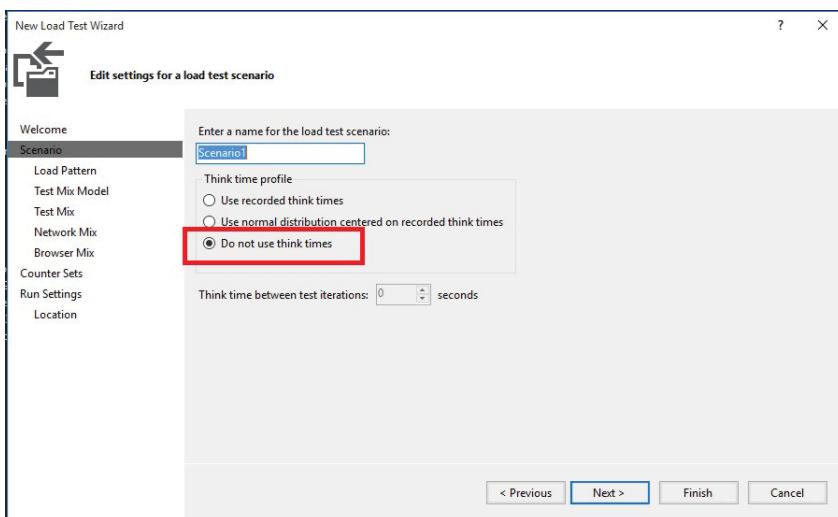


Figura 6.14: Configuração de Think Time

A aba *Load Pattern* é responsável por configurar o número de usuários simulados no teste. No exemplo, usaremos 25 usuários simultâneos, conforme vemos a seguir:

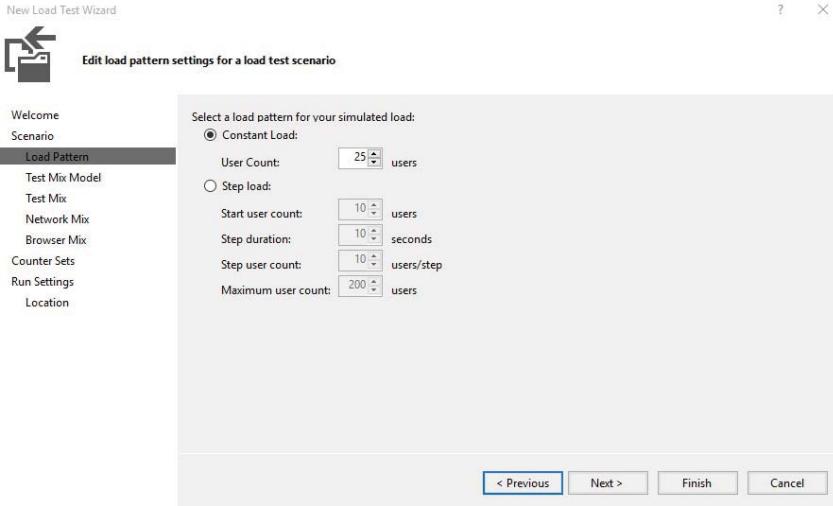


Figura 6.15: Configuração de número de usuários

A próxima aba que veremos chama-se *Test Mix*. Nela é possível selecionar os cenários de teste e atribuir uma porcentagem de distribuição de requisições para cada um dos testes. Na figura a seguir, temos dois testes configurados e a distribuição deles é igual a 50% cada.

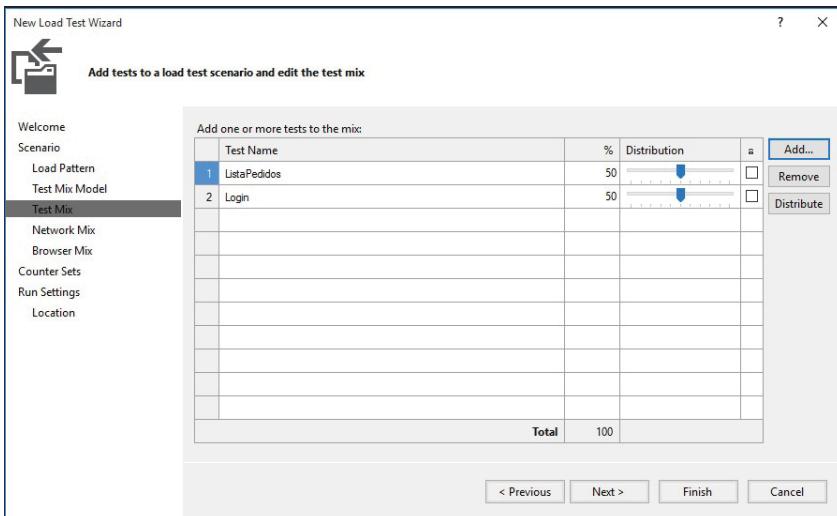


Figura 6.16: Configuração de cenários de teste

As abas *Network Mix* e *Browser Mix* são responsáveis por configurar diferentes tipos de rede e diferentes tipos de navegadores, respectivamente. Não é necessário alterar essas abas neste momento.

Na aba *Counter Sets*, é possível adicionar computadores e contadores de performance para que o Visual Studio colete essas informações e mostre-as posteriormente no relatório do teste de carga. As informações dessa aba permanecerão inalteradas.

Na aba *Run Settings*, temos o tempo de aquecimento, tempo de duração, número de iterações e o intervalo em segundos para captura dos dados. Não vamos alterar esses valores. É possível também escolher a região onde a carga será gerada, essa informação fica na aba *Location*. Após clicar em *Finish*, o teste de carga estará configurado e pronto para uso.

Para executar o teste, basta clicar no botão *Executar Teste*, conforme figura:

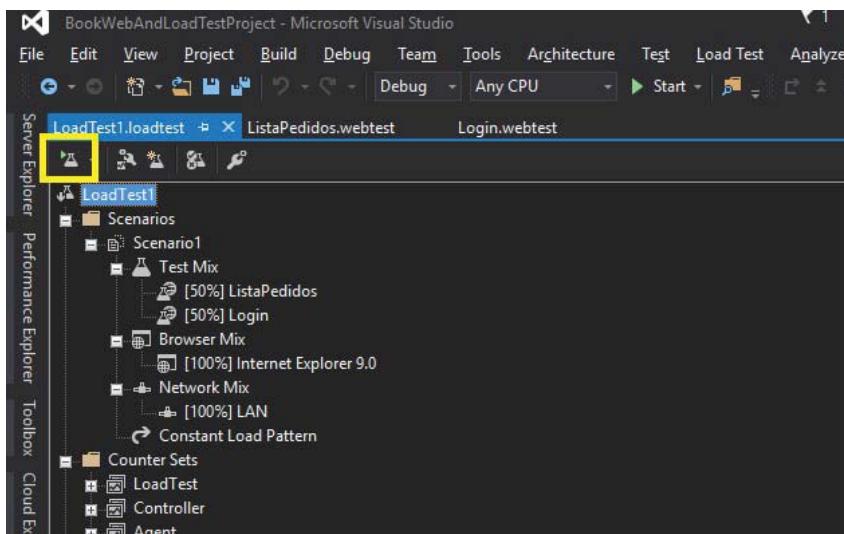


Figura 6.17: Executar teste de carga

O novo modelo padrão de execução de teste de carga é o modelo de uso do Azure, no qual as máquinas e os recursos são alocados na nuvem. Vale lembrar de que, para que os agentes de teste do Azure consigam simular sua carga sobre a aplicação, é preciso que ela esteja acessível das máquinas na nuvem. A seguir é demonstrado como é executado um teste de carga.

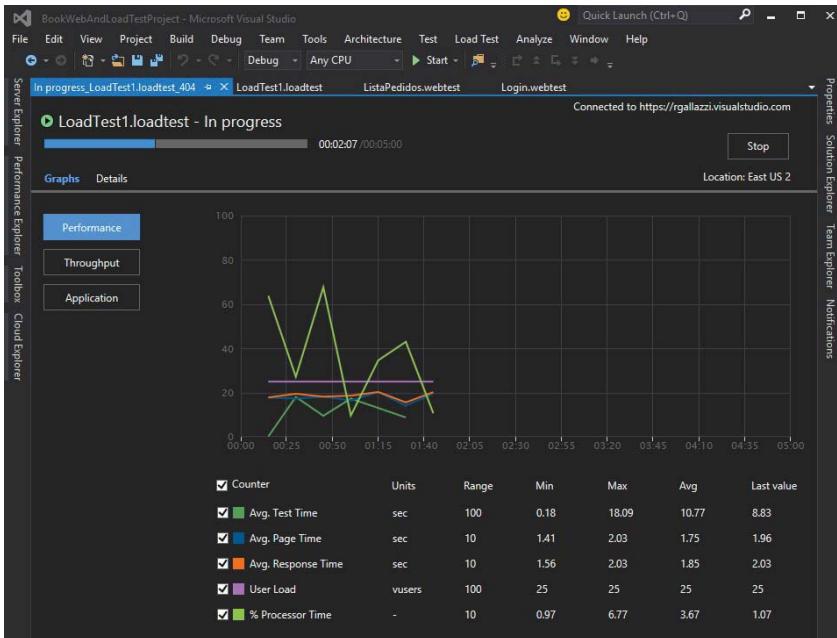


Figura 6.18: Executar teste de carga

Ao findar a execução, basta fazer o download do relatório, conforme figura:

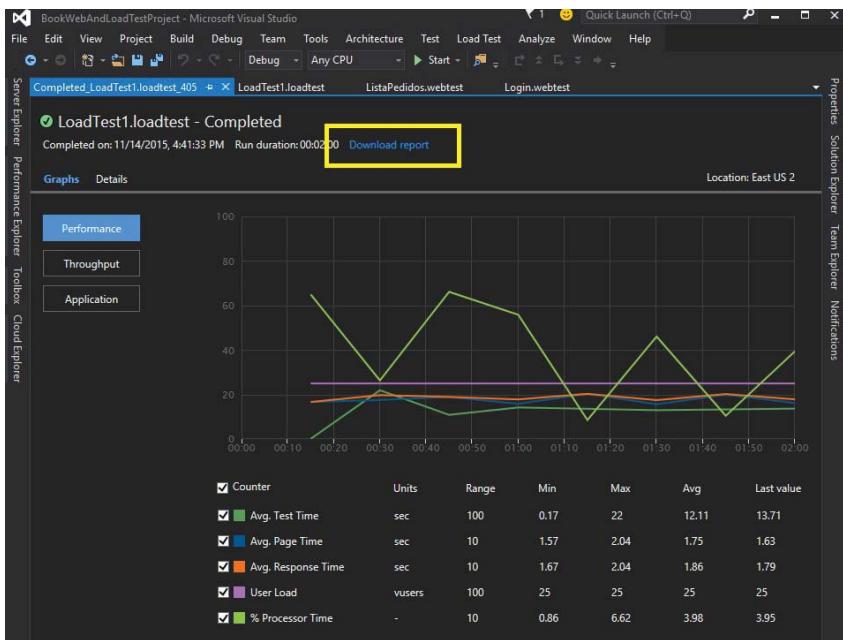


Figura 6.19: Download do relatório

Ao abrir o relatório, o Visual Studio apresenta uma série de informações, entre elas:

- Número de requisições por segundo;
- Número de testes por segundo;
- Número de testes falhos;
- Tempo médio de cada teste;
- Consumo de CPU de cada agente de carga, entre outros.

Assim, este tópico demonstrou como é possível usar o Microsoft Visual Studio atrelado ao Microsoft Azure para execução de testes de carga. Trazendo maior independência aos times, tornado o processo de execução de testes de carga mais natural, e assim agregando maior qualidade as soluções entregues pelos times de desenvolvimento.

6.5 PROFILING DE APLICAÇÕES .NET

Por Bruno de Oliveira

“Os usuários da minha aplicação reclamam constantemente da demora no processamento. Preciso entender o que acontece na execução dos meus métodos. Como posso fazer isto?”

Desde o início do desenvolvimento de uma aplicação, a busca pela eficácia para o atendimento dos requisitos de negócio é tratada como prioridade. Porém, com a agilidade necessária no mundo atual, muitas vezes também é exigida a eficiência da aplicação, ou seja, é necessário fazer com que a aplicação execute o que é requisitado utilizando os recursos computacionais disponíveis – memória, CPU, rede - da melhor forma possível.

Apesar de importante, é muito comum que a falta de eficiência seja notada apenas quando a aplicação está pronta e sendo usada pelo usuário. Nestas situações, é necessário o entendimento de quais problemas estão levando à falta de desempenho da aplicação.

Esta análise, quando feita manualmente, é uma atividade árdua onde normalmente é necessário um grande empenho e que nem sempre trará os ganhos esperados. Para facilitar este trabalho, são usadas ferramentas de *profiling* que são capazes, por meio de diferentes técnicas, de monitorar a execução da aplicação identificando os recursos gastos por um método. Desde a versão 2005, o Microsoft Visual Studio disponibiliza um conjunto de ferramentas para este fim, denominadas Profiling Tools.

Existem diferentes técnicas de coleta disponíveis, sendo que cada uma delas atende a cenários específicos. Estas técnicas são:

- **Amostragem de CPU (sampling):** periodicamente são coletados os métodos em execução;
- **Instrumentação:** avalia a quantidade de chamadas a

uma função e o seu tempo consumido;

- **Alocação de memória:** coleta dados sobre a alocação de memória .NET;
- **Concorrência:** obtém dados sobre a contenção de recursos.

Entre todas estas técnicas, recomenda-se que seja utilizada inicialmente a análise de amostragem de CPU por ser a menos intrusiva, o que simplifica o processo. Esta técnica permite a avaliação de quais métodos estão consumindo a maior quantidade de tempo. Caso seja exigida uma avaliação mais profunda de qualquer um dos métodos da aplicação, identificados como candidatos a causadores do problema de desempenho, podem ser utilizadas as demais técnicas.

Dentre os resultados apresentados pela análise, destaca-se primeiramente a visualização dos caminhos críticos percorridos pela aplicação, denominado *Hot Path*. A figura a seguir apresenta um exemplo desta visualização.

Hot Path		
Function Name	Inclusive Samples %	Exclusive Samples %
mscoree.dll	99.99	0.00
Book.Samples.PerformanceIssues.App.Program.Main	99.98	0.00
Book.Samples.PerformanceIssues.App.Program.Run	90.85	0.00
Book.Samples.PerformanceIssues.App.Program.DoLongRunning	90.85	0.00
System.Data.dll	90.85	90.43

Related Views: Call Tree Functions

Figura 6.20: Caminho crítico da aplicação – Hot Path

Uma informação de grande valia para toda análise e critério usado para determinar o caminho crítico são os valores Inclusive e Exclusive Samples que, dependendo da técnica usada na análise, podem ser identificados como Inclusive e Exclusive Times.

- **Inclusive:** valor calculado incluindo a execução das operações “filhas”.

- **Exclusive:** valor calculado exclusivamente baseando com as ações realizadas pelo método em questão.

Com esta visão inicial, muitas vezes é possível identificar os métodos da aplicação que devem ser analisados com maiores detalhes. Para a análise destes métodos, a ferramenta permite a visualização dos detalhes da função, caso os símbolos estejam disponíveis, conforme visualizamos na figura:

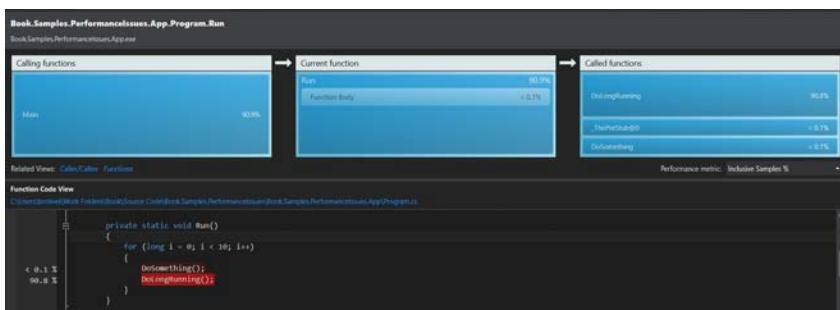


Figura 6.21: Detalhes da função

Esta visualização exibe o percentual de tempo gasto em cada um dos passos do método. Pode-se ter como exemplo os detalhes apresentados na figura anterior que demonstra que 90.8% do tempo de execução do método está sendo gasto no método `DoLongRunning`. Além disso, esta visualização permite a utilização de todas as funcionalidades disponíveis no Visual Studio para navegação do código.

Como apresentado anteriormente, caso as informações geradas pela análise não sejam suficientes, é possível a utilização de outras técnicas, como o caso da instrumentação. Esta técnica intercepta cada execução e agrupa informações como número de chamadas e tempo de execução.

Function Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time	Module Name
BookSamples.PerformanceIssues.App.exe	0	100.00	0.00	0.00	0.00	
\ BookSamples.PerformanceIssues.App.Program.Main(string[])	1	100.00	0.00	6091.21	44.45	BookSamples.PerformanceIssues
\ BookSamples.PerformanceIssues.App.Program.Run()	1	89.61	0.00	71,878.79	0.30	BookSamples.PerformanceIssues
\ BookSamples.PerformanceIssues.App.Program.DoLongRunning()	10	89.60	0.00	7,387.08	0.00	BookSamples.PerformanceIssues
\ System.Data.Common.DbCommand.ExecuteReader()	10	89.60	89.60	7,387.08	7,387.08	System.Data.dll
\ System.Data.SqlClient.SqlConnection.Open()	10	0.00	0.00	0.04	0.04	System.Data.dll
\ System.Data.SqlClient.SqlCommand.ExecuteReader(CommandBehavior)	10	0.00	0.00	0.01	0.01	System.Data.dll
\ System.Data.SqlClient.SqlConnection.CreateCommand()	10	0.00	0.00	0.00	0.00	System.Data.dll
\ System.Data.Common.DbCommand.set_CommandType(CommandType)	10	0.00	0.00	0.00	0.00	System.Data.dll
\ System.Data.Common.DbCommand.set_CommandText(string)	10	0.00	0.00	0.00	0.00	System.Data.dll
\ System.Data.SqlClient.SqlCommand.set_CommandText(string)	10	0.00	0.00	0.00	0.00	System.Data.dll
\ System.Data.SqlClient.SqlCommand.set_CommandTimeout(int)	10	0.00	0.00	0.00	0.00	System.Data.dll
\ System.Data.SqlClient.SqlCommand.set_CommandType(CommandType)	10	0.01	0.00	0.77	0.00	System.Data.dll
\ BookSamples.PerformanceIssues.App.Program.DoNothing()	10	0.01	0.00	0.00	0.00	BookSamples.PerformanceIssues
\ BookSamples.PerformanceIssues.App.Program.DoLongRunning()	1	10.34	0.00	8,967.87	0.08	BookSamples.PerformanceIssues

Figura 6.22: Cadeia de execução – Instrumentação

Além das visões apresentadas, a coleta permite o profilling da execução de comandos SQL, inclusive procedimentos armazenados, realizados através do ADO.NET. Esta coleta é chamada de Tier Interactions Profiler, ou simplesmente TIP.

A figura seguinte apresenta um exemplo de coleta realizado, monitorando duas execuções realizadas contra um banco de dados SQL Server:

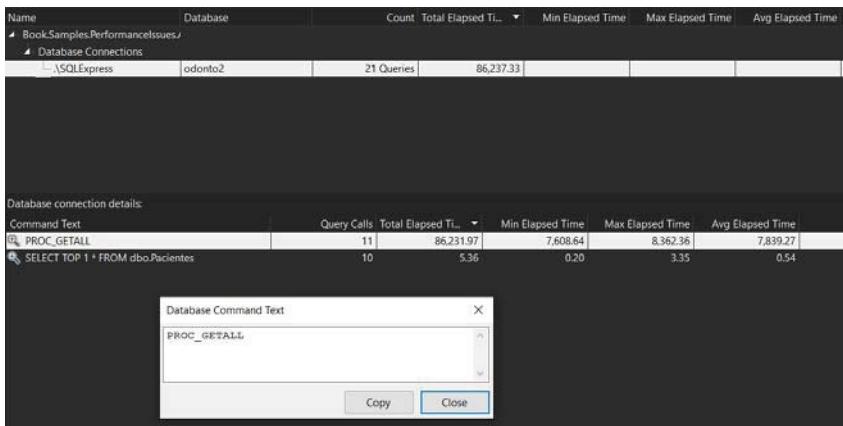


Figura 6.23: Tier Interactions

REFERÊNCIAS

Visual Studio Team Services (VSTS) -
<https://www.visualstudio.com/get-started/setup/sign-up-for-visual-studio-online>

Profiling Tools - <https://msdn.microsoft.com/en-us/library/z9z62c29.aspx>

6.6 CUIDADOS AO DEFINIR CONTRATOS DE SERVIÇOS

Por Tiago Soczek

“O contrato desse serviço possui muitas propriedades. Quais eu devo realmente informar ao chamar essa operação? Como faço para diminuir o tamanho dos pacotes enviados e recebidos pelo meu serviço?”

Esses questionamentos são normalmente encontrados em campo e, em muitos casos, o principal motivo da dúvida é: contratos de serviços mal definidos. Em uma arquitetura orientada a serviços (*Service-Oriented Architecture – SOA*), os contratos têm um papel fundamental de definir formalmente a interface técnica entre o consumidor e o serviço.

Essa interface determina a estrutura dos dados que serão enviados e recebidos em cada operação. Estes podem ser chamados de mensagem de pedido e mensagem de resposta, respectivamente.

Com a larga adoção de SOA, os serviços são usados por diversos tipos de cliente, desde uma empresa parceira, um sistema interno, ou ainda um componente do próprio sistema, como uma interface

front-end.

Ao definir esses contratos, devemos levar em consideração alguns pontos:

- **Forma de comunicação:** normalmente esses serviços são acessados pela rede, por isso, é necessária atenção com a escolha do protocolo de comunicação, formato da mensagem e payload (tamanho de cada mensagem).
- **Relação com entidades externas:** como os serviços representam a interface entre o serviço e o cliente, no qual muitas vezes não se tem total domínio, deve-se tomar cuidado com o modelo de evolução desses contratos, considerando a compatibilidade, versionamento, depreciação de operações/propriedades e a definição de uma forma de comunicação oficial das alterações realizadas nesses contratos.
- **Nível de encapsulamento:** uma das motivações do uso de serviços é o encapsulamento e reúso de funcionalidades do sistema. Para não perder esses benefícios, é essencial encontrar o nível de granularidade adequado para cada operação. Um exemplo disso é a definição de um serviço no padrão CRUD de uma tabela do banco de dados. Nesse caso, toda a regra negocial seria transferida para o cliente, sacrificando os benefícios e promovendo más práticas, como duplicação de código.

Levando em consideração esses pontos, segue um exemplo de um contrato mal definido e seus problemas, e uma proposta de solução com base em cuidados que devem ser tomados.

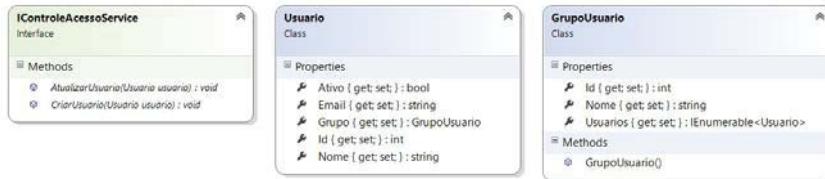


Figura 6.24: Diagrama de classes representando o serviço IContratoService

No diagrama, temos o contrato de um serviço WCF (Windows Communication Foundation) representado pela interface `IContratoService`. Ele possui duas operações: `CriarUsuario` e `AtualizarUsuario`, que recebem por parâmetro um usuário. A classe `Usuario` contém, além de suas propriedades, a relação com a classe `GrupoUsuario`. Tanto a classe `Usuario` como a `GrupoUsuario` são entidades mapeadas pelo Entity Framework (framework ORM).

Esse contrato, pode funcionar em um primeiro momento, mas possui as seguintes falhas:

- **Reaproveitamento de classes com responsabilidades já estabelecidas:** é muito comum encontrar esse tipo de situação, em que classes com responsabilidades já estabelecidas são reutilizadas para a definição de contratos. Isso torna o contrato altamente acoplado a esse conjunto de classes. Assim, qualquer alteração, mesmo uma alteração interna, como renomear uma classe base, afetará diretamente o contrato, o que muitas vezes não é o desejado.
- **Falta de clareza na obrigatoriedade das propriedades:** um desenvolvedor ao consumir esse serviço vai se perguntar "O que eu devo passar no campo `Id`? Devo passar o valor 0 ou escolher um outro valor? E a classe `GrupoUsuario`, devo preencher todos os campos? Ou

apenas o Id ?"

- **Operações distintas reaproveitando a mesma classe:** tanto a operação de `CriarUsuario` como a de `AtualizarUsuario` receberá como parâmetro a mesma classe de `Usuario`. Isso estaria correto se fizesse sentido negocial, mas neste caso cada operação precisa de um conjunto diferente de propriedades. Por um lado, seria evitada a duplicação de código, mas por outro, sacrificaria a evolução e manutenção do contrato, o que não é o desejado.
- **Dificuldade em controlar o payload:** quanto mais complexas forem as classes usadas no contrato, maior será a dificuldade para controlar o tamanho das mensagens. Por exemplo, várias propriedades da classe `Usuario` não são necessárias, e temos um relacionamento com a classe `GrupoUsuario`, em que apenas o `Id` será utilizado. Alguns formatos de mensagem emitem a estrutura e o valor default para propriedades não preenchidas, aumentando assim o tamanho da mensagem final.
- **Problemas de serialização:** serialização é o processo de transformação de objetos em dados binários ou textuais (como XML e JSON). Isso é necessário para que eles sejam salvos em arquivos ou transferidos via rede. A deserialização é o processo inverso, que transforma esses dados em objetos novamente.

A serialização e a deserialização acontecem em todas as chamadas de serviço, a definição do contrato dita diretamente o que será serializado, e quanto mais complexas as classes, mais custoso será esse processo. Quando classes são reaproveitadas no contrato, como

no caso descrito na figura anterior, que as entidades estão mapeadas com o ORM, alguns mecanismos podem entrar em conflito.

Normalmente, frameworks ORM oferecem uma funcionalidade de Lazy Loading, que basicamente é a instrumentação das classes para que os dados sejam carregados sob demanda. Ao realizar a serialização, todas as propriedades serão visitadas, e isso disparará acidentalmente o Lazy Loading. Caso o contexto do ORM não estiver mais disponível para realizar esse carregamento, será disparado uma exceção.

Também podem ocorrer problemas quando as classes são utilizadas em cenários multi-thread. Um exemplo disso é que, enquanto o processo de serialização itera sobre as coleções para obter os valores, outra thread está modificando a mesma coleção. No .NET será disparado uma exception: *InvalidOperationException: Collection was modified; enumeration operation may not execute.*

É importante observar que vários problemas têm sua causa raiz o reaproveitamento de classes, e isso infringe o primeiro princípio do SOLID (*Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion*). Esses princípios da programação orientada a objetos foram introduzidos por Robert C. Martin (Uncle Bob) no início dos anos 2000 (MARTIN, 2000).

Em seguida, a estratégia de definição do contrato será alterada. Veja o diagrama:



Figura 6.25: Diagrama de classes representando o serviço IContratoService

Essa estratégia insere duas novas classes, `CriarUsuarioRequestDto` e `AtualizarUsuarioRequestDto`, e altera as operações do serviço. `CriarUsuario` e `AtualizarUsuario` agora recebem como parâmetro as classes `CriarUsuarioRequestDto` e `AtualizarUsuarioRequestDto`, respectivamente.

Seguem detalhes dos cuidados que foram tomados na definição da nova estratégia:

- **Conjunto exclusivo de classes para o contrato:** manter um conjunto de classes exclusivo para o contrato pode ser trabalhoso, todavia é um trabalho que deve ser feito. No exemplo, foi usado o padrão DTO (Data Transfer Object) (FOWLER, 2013), que visa à criação de uma classe exclusiva para transferência. Essa classe não deve conter comportamento, apenas as propriedades necessárias para a operação.

Com esse conjunto exclusivo de classes, é mais fácil controlar quais serão as propriedades que serão recebidas/enviadas pelo serviço, e isso também torna o contrato isolado de mudanças, sendo que ele será alterado somente quando isso for feito explicitamente.

- **Utilizar classes exclusivas para cada operação:** essa estratégia possui um DTO para cada operação (`CriarUsuario` e `AtualizarUsuario`). Apesar do

conjunto de propriedades ser muito parecido, a operação negocial é diferente. Então reaproveitar o mesmo DTO para as duas operações não é uma abordagem coerente, pois terá novamente alguns problemas, como campos não preenchidos ou desnecessários. Quando fizer sentido negocial, o mesmo DTO poderá ser utilizado; a ideia não é encorajar a duplicação de código, mas sim, a definição correta dos DTO para cada operação.

Essa estratégia, exigirá novas atividades, como o mapeamento dos DTOs para as entidades e vice-versa. Isso pode se tornar um pouco trabalhoso, porém é um processo simples e de controle total do time da aplicação, que torna mais fácil a manutenção.

Esse cuidados não devem ser vistos como um guia completo de definição de contratos de serviços, mas, com eles, os problemas mencionados serão evitados e você terá mais controle sobre o processo, permitindo que a camada de serviços da aplicação cresça de forma organizada.

6.7 BUNDLING E MINIFICATION

Por Christiano Donke

“Minhas páginas estão muito lentas para carregar todos os scripts, e isso consumiu todo o meu pacote de internet. E agora?”

Ao se trabalhar com aplicações Web, é comum ter páginas façam uso de arquivos JavaScript e CSS. Quando uma página tem muitos links para recursos externos, seu carregamento pode ser tornar algo demorado e custoso, já que o usuário terá de esperar todo esse conteúdo ser baixado. *Bundling* (empacotamento) e

minification (minificação) são duas técnicas introduzidas no ASP.NET 4.5 para melhorar o tempo de carregamento de páginas.

Bundling

BUNDLE /'bəndl/ vt+vi embrulhar, empacotar, entrouxar, enfeixar

A maioria dos navegadores modernos limitam o número de conexões simultâneas a um mesmo servidor a 6 (ver *Lista de limites por navegador* mais à frente). Isso significa que, enquanto essas 6 requisições não terminarem, as outras ficarão em espera no navegador, como mostra a figura:

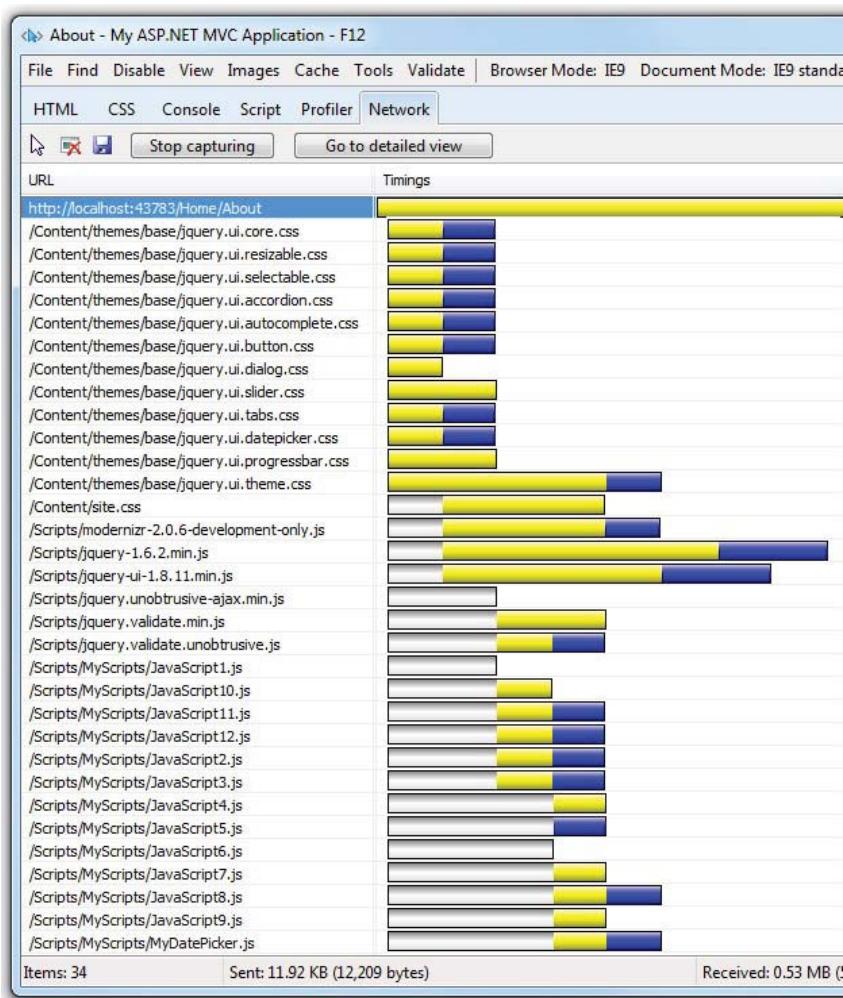


Figura 6.26: A aba Rede do Ferramentas de Desenvolvedor do IE, mostrando os tempos de requisição de recursos de uma página

Nessa figura, notamos que cada artefato da página contém uma barra. Essa barra divide-se em três partes, sendo: a primeira parte indicando o tempo de espera na fila de requisições; a segunda parte corresponde ao tempo de envio das requisições para o servidor; e por fim, a última parte detalha o tempo de download do item.

A funcionalidade de Bundling do ASP.NET MVC permite juntar diversos arquivos em um único, para limitar a quantidade de requisições ao servidor para download. Isso pode ser feito em CSS, JavaScripts ou criar um próprio. Esta técnica não reduz o tamanho dos dados trafegados, apenas a quantidade de arquivos.

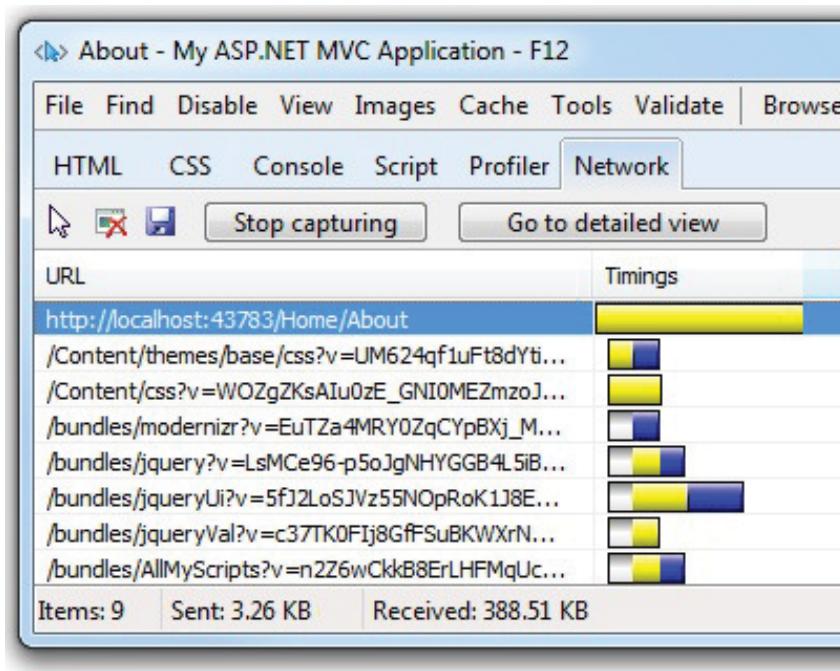


Figura 6.27: A aba Rede do Ferramentas de Desenvolvedor do IE, mostrando os tempos de requisição de recursos de uma página, após aplicar o Bundling e Minification

Após aplicar o bundling nos JavaScripts e CSS da página de exemplo, nota-se a redução na quantidade de arquivos (Bundling) e do tamanho total transferido (Minification).

Minification

MINIFICATION /'mInifɪ'keɪʃən/ vt ato de diminuir, reduzir

Minificação é o processo em que a aplicação percorre todo um script ou CSS removendo toda presença de conteúdo não necessário, como comentários, espaços extras etc. Além disso, os nomes de variável também são encurtados. Todo esse trabalho para deixar os arquivos mais enxutos, logo, mais rápidos para download.

Não existe custo nenhuma a mais para o JavaScript. Ele não se importa se a variável chama `SaborDoSorvete` ou `q`. E ainda há um pequeno ganho, pois o script ocupará menos espaço em memória.

Considere o seguinte script:

```
AddAltToImg = function (imageTagAndImageID, imageContext) {  
    ///<signature>  
    ///<summary> Adds an alt tab to the image  
    // </summary>  
    //<param name="imgElement" type="String">The image selector.</param>  
    //<param name="ContextForImage" type="String">The image  
    context.</param>  
    ///</signature>  
  
    var imageElement = $(imageTagAndImageID, imageContext);  
  
    imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));  
}
```

Após a minificação, a função é reduzida para:

```
AddAltToImg = function (n, t) { var i = $(n, t); i.attr("alt", i.attr("id").replace(/ID/, "")) }
```

Controlando o Bundling e a Minification

Tanto o bundle quanto a minificação são habilitados ou desabilitados de acordo com o valor do atributo debug no elemento system.web/compilation no Web.Config. Se o valor for true (verdadeiro), os dois são desabilitados.

```
<system.web>
    <compilation debug="true" />
        <! Linhas removidas para brevidade >
</system.web>
```

Para forçar com que a aplicação habilite ou não a funcionalidade, definindo o valor na propriedade EnableOptimizations da classe BundleTable .

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
        "~/Scripts/jquery {version}.js"));

    // Código removido para brevidade.
    BundleTable.EnableOptimizations = true;
```

Lista de limites por navegador

	Conexões por servidor	Limite de conexões	Downloads em paralelo		
			Script/Script	Script/CSS	Script/Imagem
IE10	6	17	Sim	Sim	Sim
IE11	13	17	Sim	Sim	Sim
Chrome 34	6	10	Sim	Sim	Sim
Firefox 27	6	17	Sim	Sim	Sim
IEMobile 9	6	60	Sim	Sim	Sim
Android 4	6	17	Sim	Sim	Sim
Safari 34	6	10	Sim	Sim	Sim

As definições são:

- **Conexões por servidor:** quando o HTTP/1.1 foi

lançado, com conexões persistentes habilitadas por padrão, a recomendação era que os navegadores abrissem apenas 2 conexões por servidor. Páginas que tinham 10 ou 20 recursos servidores por apenas 1 servidor carregavam lentamente, pois esses recursos eram baixados dois-a-dois. Com o tempo, os navegadores aumentaram esse limite.

- **Límite de conexões:** este limite faz referência ao máximo de conexões que um navegador pode abrir para todas as requisições. O limite superior é 60, então, mesmo que um navegador consiga mais, vai aparecer apenas 60.
- **Downloads em paralelo:** alguns navegadores quando começam a baixar arquivos externos, eles aguardam até o download terminar, ser analisado e executado, antes de começar o próximo script. Embora a ordem de análise e execução dos scripts seja importante por causa de suas dependências, os downloads podem ser em paralelo. Cada item valida o download em paralelo dos itens indicados.

6.8 LUTANDO CONTRA ALTERAÇÕES INADVERTIDAS NO PLANEJAMENTO

Por Thiago Camargos Lopes

Quando se adotam metodologias ágeis, valoriza-se “*Software em funcionamento mais que documentação abrangente*”.

É comum, então, que o planejamento da equipe se torne parte da documentação do sistema. O TFS (Team Foundation Server) facilita o planejamento de equipes que se propõe a serem ágeis, facilitando a inserção, edição e exclusão de itens de trabalho. Pelo

TFS, é possível definir permissões que controlam quais ações cada membro da equipe pode realizar em um determinado planejamento.

Permissões para equipes ágeis: mais ou menos?

Ao iniciar o desenvolvimento de software usando o TFS, pode-se definir um conjunto de permissões para cada envolvido no desenvolvimento daquele sistema. É recomendado que equipes ágeis adotem uma abordagem menos restritiva no conjunto de permissões.

Em um ambiente com muitas restrições, é necessário definir uma ou mais pessoas para realizar tarefas operacionais que os desenvolvedores não têm como fazer. As frases seguintes se tornam comuns:

- “Preciso que crie uma nova área para uma funcionalidade”.
- “Preciso que crie uma nova iteração para a próxima Sprint”.
- “Faltarei um dia nessa Sprint, preciso que altere a capacidade de trabalho definida para mim”.
- “Vou iniciar o desenvolvimento de uma nova funcionalidade. Faça um branch para mim”.
- “Acabei o desenvolvimento. Preciso que faça o merge com o código da entrega do mês passado”.

O processo em que poucas pessoas ficam responsáveis pela execução dessas tarefas operacionais cria uma nova camada de gerenciamento. Isso vai contra a agilidade (“Indivíduos e interações mais que processos e ferramentas”). Isso gera desperdícios. Pode impactar em pelo menos 7 tipos de desperdício, descritos pelos Poppendiecks.

- Partially Done Work (Trabalho parcialmente

finalizado)

- Extra Features (Funcionalidades extras)
- Relearning (Reaprendizagem)
- Handoffs (Transferência)
- Delays (Espera)
- Task Switching (Troca de tarefas)
- Defects (Defeitos)

Como manter o controle sem limitar as permissões?

No TFS, informações sobre todas as operações são gravadas. Essas informações permitem que seja possível auditar e descobrir “quem”, “como”, “onde” e “quando” foram realizadas determinadas operações. Permite-se que todos possam executar todas as operações necessárias para seu trabalho, e audita-se o que for necessário.

Essas informações estão gravadas em dois lugares:

- **Tabelas do SQL Server:** guardam por um tempo definido metadados sobre todas as operações executadas no TFS. Existe uma interface web disponível para consultar as operações.
- **Logs do IIS:** as requisições de operações podem ser salvas no log do IIS, dado que o IIS é usado pela camada de aplicação (application tier) do TFS.

Adiante, serão apresentadas duas maneiras de se auditar as operações executadas.

Cenário de exemplo

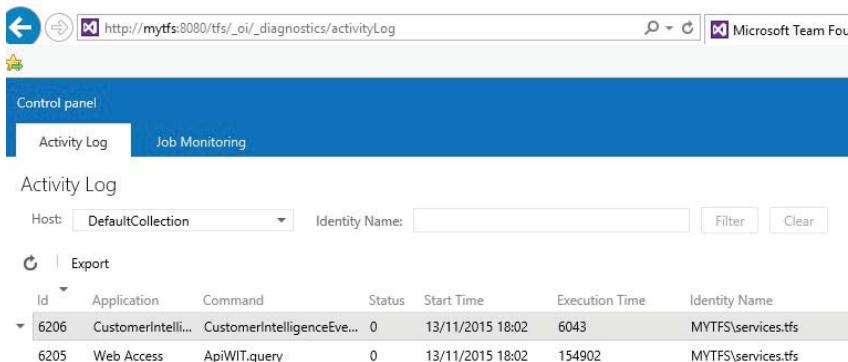
Em uma segunda-feira de manhã, ao iniciar o trabalho da semana, um desenvolvedor abre a lista de tarefas do projeto para localizar a próxima tarefa a ser feita. Porém, a consulta retorna mais

de 5.000 itens. Dos itens retornados, o desenvolvedor verifica que há tarefas de vários projetos. Todos esses itens estão com o campo *Area Path* definidos como a raiz, e as subáreas não existem mais. Na sexta-feira à noite, os valores ainda existiam. Portanto, as subáreas foram excluídas durante o final de semana. Quem foi?

A forma simples é:

1. Acessar o Activity Log via interface web;
2. Exportar o conteúdo para um arquivo CSV;
3. Localizar a operação desejada através do Microsoft Excel.

A interface web é acessível via URL no formato ([http|https://\(nome\):\(porta\)/tfs/_oi](http://mytfs:8080/tfs/_oi/_diagnostics/activityLog)).



The screenshot shows the Microsoft Team Foundation Server (TFS) Activity Log interface. At the top, there is a navigation bar with icons for back, forward, search, and refresh, followed by the URL 'http://mytfs:8080/tfs/_oi/_diagnostics/activityLog'. Below the URL is a Microsoft Team Foundation logo. The main area has a blue header bar with the text 'Control panel' and two tabs: 'Activity Log' (which is selected) and 'Job Monitoring'. Under the 'Activity Log' tab, there is a search bar with 'Host: DefaultCollection' and 'Identity Name:' fields, along with 'Filter' and 'Clear' buttons. Below the search bar is a table with columns: Id, Application, Command, Status, Start Time, Execution Time, and Identity Name. Two rows of data are visible: one for 'CustomerIntelli...' with ID 6206 and another for 'Web Access' with ID 6205. Both rows show 'CustomerIntelligenceEve...' and 'ApiWIT.query' respectively as the command, and '0' as the status. The execution times are 6043 and 154902 milliseconds, and the identity name is 'MYTFS\services.tfs' for both.

Id	Application	Command	Status	Start Time	Execution Time	Identity Name
6206	CustomerIntelli...	CustomerIntelligenceEve...	0	13/11/2015 18:02	6043	MYTFS\services.tfs
6205	Web Access	ApiWIT.query	0	13/11/2015 18:02	154902	MYTFS\services.tfs

O campo `host` filtra os resultados pelo escopo das operações. É preenchido com o nome de cada Team Project Collection e o Team Foundation.

O campo `Identity Name` filtra as operações executadas pelo usuário especificado. No cenário proposto, deseja-se descobrir quem foi o usuário a excluir valores do campo `Area Path`, portanto, não será preenchido.

Ao clicar em *Export*, será recebido um arquivo `.csv`, conforme

a figura:



Por meio do Excel, pode-se filtrar as operações até se chegar à linha desejada.

Application	Command	StartTime	IdentityName	IpAddress
Web Access	AdminAreas.DeleteClassificationNode	15/11/2015 13:10	MYTFS\joao	189.102.210.171

É importante que se conheça os valores disponíveis na coluna **Command** e o que fazem. Uma maneira simples de se descobrir o valor registrado quando se exclui áreas é criando e excluindo uma em um ambiente de testes. Assim, é possível localizar o log específico da operação utilizando filtros conhecidos (por exemplo, IP, usuário, horário, tipo de acesso).

- Pontos positivos:
 - Facilidade de acesso.
 - Facilidade para filtrar.
- Pontos negativos:
 - Log de atividades limitado a poucos dias.
 - Só é possível filtrar pelo escopo e por usuário pela interface web.
 - A quantidade de operações gravadas pode gerar um arquivo CSV muito grande, fazendo com que não seja aberto por edições 32 bits do MS Excel.
 - Disponível apenas a partir do TFS 2012.

Forma complexa:

A forma complexa é navegar entre os logs do IIS para localizar o evento desejado. Usando a ferramenta LogParser28, é possível consultar os logs e gerar um arquivo contendo apenas as operações

desejadas.

Os logs do TFS são encontrados no caminho C:\inetpub\logs\LogFiles de cada servidor da camada de aplicação (*application tier*). Para esse cenário, após localizar o caminho dos logs e instalar o LogParser, aplicaremos os seguintes comandos:

```
C:\analysis>xcopy c:\inetpub\logs\LogFiles\W3SVC2\* .  
C:\analysis>"C:\Program Files (x86)\Log Parser 2.2\LogParser.exe" -i:IISW3C -o:CSV "SELECT * FROM *.log WHERE cs_uri_stem like '%_areas'" > output.csv
```



The screenshot shows a Windows Command Prompt window titled "Administrator: Log Parser 2.2". It displays two commands being run. The first command is "xcopy c:\inetpub\logs\LogFiles\W3SVC2* ." which copies files from the W3SVC2 log directory. The second command is "\"C:\Program Files (x86)\Log Parser 2.2\LogParser.exe" -i:IISW3C -o:CSV "SELECT * FROM *.log WHERE cs_uri_stem like '%_areas'" > output.csv" which runs the LogParser to extract specific log entries based on the URI stem and outputs them to a CSV file named "output.csv".

O arquivo será gerado contendo apenas os dados resultantes da consulta efetuada pelo LogParser:

date	time	c-ip	cs-username	cs-method	cs-uri-stem
16/11/2015	01:10:27	189.102.210.171	mytfs\joao	GET	/tfs/DefaultCollection/Gitsrum/_admin/_areas
16/11/2015	01:10:38	189.102.210.171	mytfs\joao	POST	/tfs/DefaultCollection/713f444c-6f93-4bf1-b7ad-6190dbad200a/_admin/_Areas/DeleteClassificationNode
16/11/2015	01:10:38	189.102.210.171	mytfs\joao	POST	/tfs/DefaultCollection/713f444c-6f93-4bf1-b7ad-6190dbad200a/_admin/_Areas/UpdateAreasData

- **Pontos positivos:**

- Não há limite padrão de retenção de logs.
- O filtro das operações é executado antes da exportação dos dados, gerando um arquivo menor.

- **Pontos negativos:**

- Dificuldade maior em criar a consulta necessária para filtrar as operações desejadas.

- Necessita acesso aos logs do servidor do IIS.

Outras formas de utilização:

O log de atividades do TFS possui várias outras informações úteis. São elas:

- Número de usuários únicos que acessam o TFS.
- Padrões de acesso (O que é mais acessado? Por exemplo, Áreas de testes, build, source control etc.).
- Localidades de acesso através do IP.
- Ferramentas e versões utilizadas para acessar o TFS (Browser, Excel, Project, Visual Studio, MTM etc.)

REFERÊNCIAS

Permissões para Team Foundation Server (TFS) -
<https://msdn.microsoft.com/en-us/library/ms252587.aspx>

6.9 AUMENTANDO A DISPONIBILIDADE E O DESEMPENHO DE WEBSITES POR MEIO DE SEUS APPLICATION POOLS

Por Henrique Resende da Silva

“Minhas aplicações web estão demorando muito tempo para carregar as páginas e estão muito lentas. Além disso, muitas vezes param de funcionar. O que eu posso fazer para aumentar a disponibilidade das minhas aplicações?”

Para o IIS (Internet Information Services), um *application pool* é definido como um recipiente (*container*) que abriga uma ou mais aplicações web. Cada um desses recipientes é representado no

sistema operacional por um ou mais processos (*worker processes*, `w3wp.exe`). Esses processos são os responsáveis por monitorar a fila de requisições, processar e devolver a resposta dessas requisições para o cliente.

A partir dessa definição, é possível então concluir que o desempenho e a disponibilidade de uma aplicação web é diretamente afetada pelo desempenho e disponibilidade de seu application pool. Cada application pool possui um conjunto de configurações que afetam diversos aspectos de sua execução. Nessa seção, serão discutidas as principais configurações e suas respectivas alterações com o objetivo de maximizar a disponibilidade e desempenho das aplicações.

É importante ressaltar que as práticas a seguir consideram apenas cenários em que a aplicação é hospedada em um servidor ou um conjunto de servidores dedicados ou compartilhados. Porém, os requisitos de hardware estão corretamente dimensionados para suportar a carga extra gerada pelas configurações, que tem como foco o aumento do desempenho e da disponibilidade ao custo desta carga extra.

Acessando o painel de configurações avançadas de application pool

Para acessar o painel de configurações avançadas de um application pool, primeiramente deve-se selecionar o nível *Application Pools* no menu do *IIS Manager*. Depois, devemos selecionar qual o application pool desejamos configurar e clicar em *Advanced Settings*.

Caso seja necessário alterar a configuração padrão para todos os application pools, deve-se selecionar a opção *Set Application Pool Defaults*. Com isso feito, o painel de configurações será mostrado conforme a seguir:

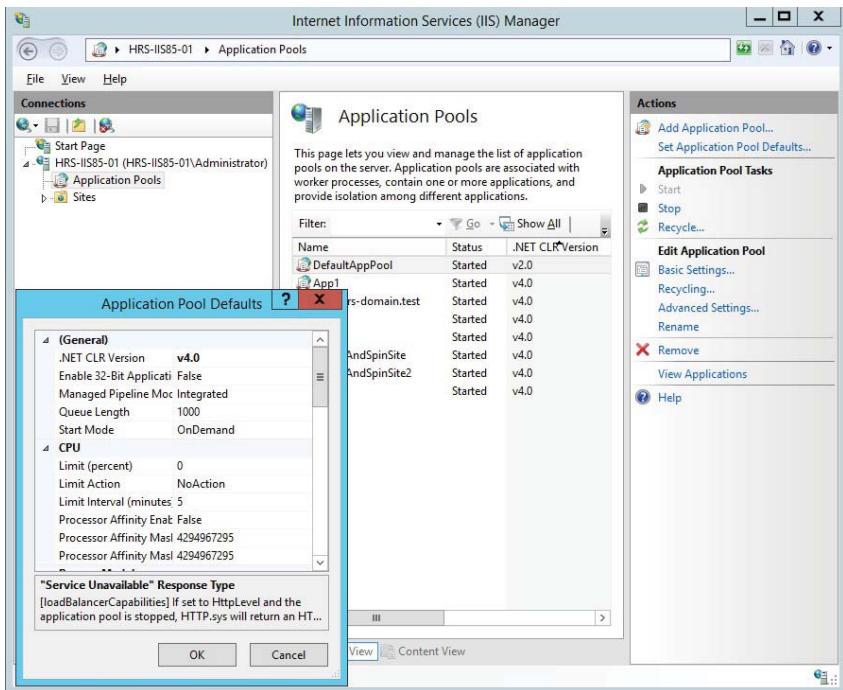


Figura 6.34: IIS Manager e painel de configurações de application pools

Configurando o application pool para maximizar disponibilidade

Ping Enabled

Configura o IIS para monitorar os processos do application pool (w3wp.exe) através do envio de um ping periódico. As respostas recebidas são usadas pelo IIS para determinar a saúde de cada processo. Caso comecem a ocorrer falhas nessa comunicação, o processo é finalizado e um novo é criado para o substituir, evitando então que a aplicação pare de processar as requisições.

Esse monitoramento é feito por meio do componente WAS (Windows Process Activation Service) do IIS, visto com maior detalhe na seção *Suportando o IIS e entendendo o seu*

funcionamento, contida no capítulo 1.

Por padrão, essa opção já é habilitada e configurada para enviar um ping a cada 30 segundos, com um tempo máximo de resposta de 90 segundos. A recomendação é que se mantenha essa configuração.

Ping Enabled	True	<input checked="" type="checkbox"/>
Ping Maximum Response Time (seconds)	90	
Ping Period (seconds)	30	
Shutdown Time Limit (seconds)	90	
Startup Time Limit (seconds)	90	

Figura 6.35: Funcionalidade Ping Enabled

Rapid-Fail Protection

Com esta configuração, o IIS passa a monitorar quantas vezes os processos do application pool (w3wp.exe) são inesperadamente terminados (*crash*). Caso esta quantidade ultrapasse um número máximo dentro de um limite de tempo (ambos configuráveis), o application pool será desativado, não permitindo que novos processos sejam criados, efetivamente tirando as aplicações ali hospedadas do ar.

Em ambientes nos quais a disponibilidade das aplicações é de extrema importância (produção), essa funcionalidade pode deixar aplicações inteiras ou múltiplas aplicações fora do ar por problemas muito pontuais, que afetam apenas uma pequena parte daquela aplicação problemática.

Por padrão, esta funcionalidade vem habilitada e configurada para monitorar um limite de 5 crashes em 5 minutos. Ou seja, caso ocorram cinco problemas nos processos do application pool dentro de 5 minutos, o application pool será desabilitado, deixando a aplicação fora do ar. A recomendação é que se desabilite essa funcionalidade.

4 Rapid-Fail Protection	
"Service Unavailable" Response Type	HttpLevel
Enabled	False
Failure Interval (minutes)	5
Maximum Failures	5

Figura 6.36: Funcionalidade Rapid-Fail Protection

Recycling

Recycle de um application pool é a técnica que o IIS usa para substituir os processos de um application pool (`w3wp.exe`) por novos. As razões para se executar ou não o recycle são muitas e não fazem parte do escopo desta seção. O importante é entender que, ao se executar um recycle, potencialmente estamos causando uma interrupção na execução da aplicação, principalmente quando se trata de aplicações que mantêm as sessões de usuários guardadas dentro do processo.

Recycles podem ser executados manualmente ou automaticamente, conforme parâmetros definidos nas configurações do application pool. Por padrão, os application pools são configurados para executar um recycle a cada 29 horas (1.740 minutos). As outras configurações de recycle automáticos são desabilitadas por padrão.

Quando se pensa em aumentar a disponibilidade de uma aplicação, estamos falando em manter os processos do seu application pool (`w3wp.exe`) no ar pelo maior tempo possível. Portanto, a execução de recycles, principalmente automáticos, deve ser restringida ao mínimo ou até erradicada. Com isso, a recomendação é de que os recycles automáticos sejam desligados.

Private Memory Limit (KB)	0
Regular Time Interval (minutes)	0
Request Limit	0
Specific Times	TimeSpan[] Array
Virtual Memory Limit (KB)	0

Figura 6.37: Funcionalidade de Recycle automático

Configurando o application pool para maximizar desempenho

.NET CLR Version

O IIS possui a capacidade de hospedar tanto aplicações escritas em código gerenciado (.NET) quanto as escritas em código não gerenciado (ASP clássico, PHP etc.). Para o correto funcionamento de aplicações escritas em .NET, existe a necessidade de se carregar nos processos do application pool (w3wp.exe) o CLR (Common Language Runtime) específico para a versão de .NET sendo usada. Esse runtime, além de permitir a execução do código, fornece serviços básicos como compilação, gerenciamento de memória, segurança e outros.

Para aplicações escritas em código não gerenciado ou que não dependem da CLR do .NET para executar, o carregamento desse runtime vai apenas sobrecarregar os processos do application pool, pois não será utilizado em momento algum. Portanto, para essas aplicações, a recomendação é não carregar o runtime. Para isso existe a opção *No Managed Code* nesta configuração.

NET CLR Version	No Managed Code	<input checked="" type="checkbox"/>
-----------------	-----------------	-------------------------------------

Figura 6.38: Funcionalidade de Recycle automático

Managed Pipeline Mode

A partir da versão 7.0 do IIS (Windows 2008), foi introduzido o conceito de modo de execução das requisições ou Managed Pipeline Modes. Existem dois modos de execução: clássico e integrado.

Aplicações escritas em .NET se beneficiam do modo de execução integrado, pois, nesse modo, o pipeline de execução do IIS é, como sugere o nome, integrado ao pipeline de execução do .NET. Isso otimiza o número e remove a duplicidade de passos necessários para executar as requisições .NET, além de tornar execução e resposta dessas requisições mais rápidas.

Por esse motivo, a recomendação para aplicações .NET é ter o application pool configurado para o modo integrado de execução.



Figura 6.39: Funcionalidade de Recycle Automático

Start Mode

Trata-se do modo de inicialização dos processos do application pool. O valor padrão (*OnDemand*) configura o IIS para apenas iniciar os processos (`w3wp.exe`) mediante a chegada de uma requisição em sua fila. Isso, entretanto, pode afetar o desempenho dessa requisição, devido aos passos extras necessários para subir um processo no sistema operacional. Como o nosso objetivo é maximizar o desempenho da aplicação, a recomendação aqui é configurar para os processos ficarem sempre no ar a partir da inicialização do IIS. Essa opção chama-se *AlwaysRunning*.



Figura 6.40: Funcionalidade de Start Mode

Idle Time-out

Por padrão, os application pools vem configurados para

terminar os processos (`w3wp.exe`) após 20 minutos de inatividade (quando não há nenhuma requisição sendo processada). Isso ocorre para diminuir o consumo de memória do servidor. Porém, existe um custo de processamento para subir um novo processo, principalmente quando se trata de .NET, por causa da compilação que o código sofre antes de começar a processar as requisições. Para evitar que esse atraso ocorra continuamente, a recomendação é desabilitar essa configuração.



Figura 6.41: Funcionalidade de Idle Time-out

Matriz de configurações para maximizar desempenho e disponibilidade de application pools

Configuração	Padrão	Recomendado
Ping Enabled	True	True
Rapid Fail Protection	Enabled = True	Enabled = False
Recycling	Private Memory Limit = 0; Regular Time Interval = 1740; Request Limit = 0; Specific Times = 0; Virtual Memory Limit = 0	Private Memory Limit = 0; Regular Time Interval = 0; Request Limit = 0; Specific Times = 0; Virtual Memory Limit = 0
.NET CLR Version (para código não gerenciado)	v2.0 ou v4.0	No Managed Code
Managed Pipeline Mode (para código .NET)	Integrated	Integrated
Start Mode	OnDemand	AlwaysRunning
Idle Timeout	20	0

REFERÊNCIAS

Configure the Request Processing Mode for an Application Pool - [https://technet.microsoft.com/pt-pt/library/cc725564\(v=ws.10\).aspx](https://technet.microsoft.com/pt-pt/library/cc725564(v=ws.10).aspx)

IIS.NET: Recycling Settings for an Application Pool - <http://www.iis.net/configreference/system.applicationhost/applicationpools/add/recycling>

MSDN: Internet Information Services (IIS) - [https://msdn.microsoft.com/en-us/library/ee532514\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/ee532514(v=vs.90).aspx)

MSDN: ManagedPipelineMode Enumeration - [https://msdn.microsoft.com/en-us/library/microsoft.web.administration.managedpipelinemode\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/microsoft.web.administration.managedpipelinemode(v=vs.90).aspx)

Technet: Common Language Runtime - [https://technet.microsoft.com/en-us/subscriptions/cc265151\(v=vs.95\).aspx](https://technet.microsoft.com/en-us/subscriptions/cc265151(v=vs.95).aspx)

Technet: IIS Manager - [https://technet.microsoft.com/en-us/library/cc753842\(v=ws.10\).aspx](https://technet.microsoft.com/en-us/library/cc753842(v=ws.10).aspx)

Technet: IIS Windows Access Process Activation Service (WAS) - [https://technet.microsoft.com/en-us/library/cc735229\(v=ws.10\).aspx](https://technet.microsoft.com/en-us/library/cc735229(v=ws.10).aspx)

Technet: IIS Worker Process - <https://technet.microsoft.com/en-us/library/cc735084%28v=ws.10%29.aspx?f=255&MSPPError=-2147217396>

6.10 COMO O WEB DEPLOY PODE SER ÚTIL?

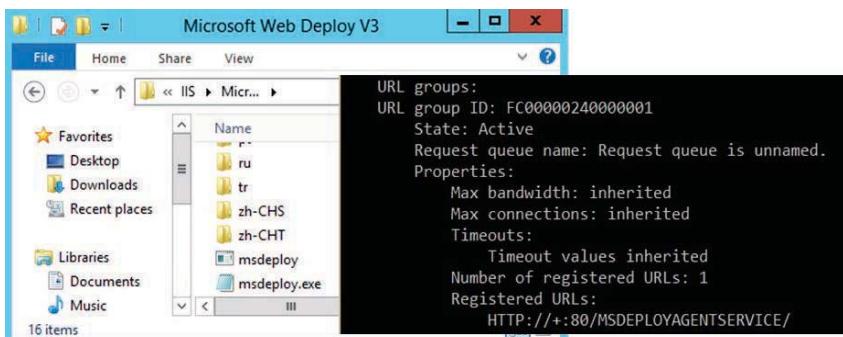
Por Demetrio Costa

“Recebo muitos pacotes para implantação. Às vezes, 3 ou 4 por dia. E tenho de implantá-las em dezenas de servidores. O que eu posso fazer para publicar todas estas versões em todos os servidores de maneira automatizada?”

A ferramenta *Web Deploy* simplifica a migração, implantação, publicação e distribuição de aplicações web, além da gestão das versões aplicadas em ambiente web com IIS (Internet Information Services). Atualmente, na versão 3.6, a ferramenta incorpora muitas features, como suporte a encriptação de arquivos de configuração de aplicações ASP.NET e instrumentação com ETW.

Quando o *Web Deploy* estiver instalado, será possível constatar que é composto por dois componentes principais: um executável (`MSDeploy.exe`) e um Windows Service (*Web Deployment Agent Service*).

O *Web Deployment Agent Service* é o serviço responsável por permitir que comandos do `MSDeploy.exe` sejam executados remotamente. É possível verificar que o serviço, quando iniciado, começa a “ouvir” requisições na porta TCP 80.



O Web Deploy foi desenvolvido para trabalhar com *server farms*. Um *server farm* é uma estrutura composta por um ou mais servidores, sendo que estes estão interligados, dividem a mesma localidade e função. Desta forma, possuem um maior poder computacional e trabalham como se fossem um único servidor.

Sincronização de uma aplicação entre servidores web

O cenário mais comum é garantir que todos os servidores da farm tenham a mesma versão do código publicado, as mesmas permissões e a mesma configuração. A ferramenta possui dois tipos de argumentos que podem ser combinados para gerar a ação necessária. Estes argumentos são: *verb* (verbo) e *provider* (provedor).

O verbo é utilizado para identificar a ação que está sendo realizada. Por exemplo, *sync* é um verbo referente a sincronismo, e *delete* é um verbo para exclusão.

O provedor identifica quem sofrerá a ação (origem ou destino). Por exemplo, *appPoolConfig* identifica a configuração de um determinado application pool, que pode ser de base para criar ou alterar a configuração de um application pool em outro servidor, ou que pode ser o alvo da alteração.

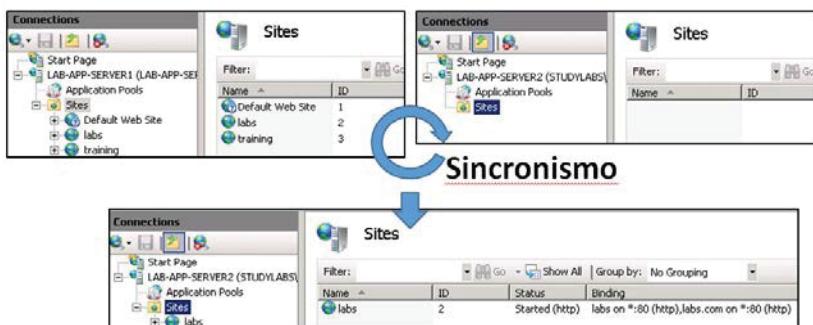
A seguir, veja um exemplo que possui como ação sincronizar uma aplicação entre dois servidores. Neste exemplo, o servidor origem (*source*) já possui a aplicação publicada, e o destino (*dest*) não possui aplicação e nenhuma configuração:

```
msdeploy  
    verb:sync  
    source:apphostconfig=[iis application] ,computername=[FQDN se  
rvidor origem]  
    dest:apphostconfig=[iis application],computername=[FQDN servi  
dor destinatário]  
    enableLink:AppPoolExtension
```

```
enableLink:ContentExtension  
setParamFile=[arquivo de parâmetros]
```

Observando melhor o comando, é possível verificar que é efetuado o sincronismo da configuração do Application Pool (enableLink:AppPoolExtension) e também o conteúdo (arquivos e pastas) pertencentes à aplicação (enableLink:ContentExtension).

E como as configurações podem ter diferenças entre os servidores, é possível trabalhar com um arquivo de parâmetros que trabalha de forma similar aos arquivos de transformação do msbuild11 (setParamFile).



Sincronização de todas as aplicações entre servidores web

"E se for necessário sincronizar todos os sites e aplicações web do servidor?"

Esta tarefa também é completamente atendida pelo Web Deploy. Por exemplo, com o seguinte comando:

```
msdeploy  
verb:sync  
source:webServer ,computername=[FQDN servidor de origem]  
dest:webServer,computername=[FQDN servidor destinatário]
```



Com esta linha de comando, é possível sincronizar o conteúdo de dois servidores, considerando configurações, conteúdo e permissões. O resultado será a garantia de um ambiente totalmente replicado.

Comparação de versões entre servidores

"Consigo verificar o que será alterado antes de efetuar a sincronização?"

O Web Deploy possui alguns comandos que permitem verificar dependências e o que será sincronizado antes de efetuar a sincronização. Por exemplo:

```
msdeploy
verb:sync
source:webServer ,computername=[FQDN servidor de origem]
dest:webServer,computername=[FQDN servidor destinatário]
whatif
```

O comando whatif retornará uma lista de todo conteúdo que será sincronizado, como neste seguinte exemplo:

```
Info: Using ID 'cd49337e 1930 4698 b96d ecba3eefb43a' for connections to the remote server.
Info: Using ID '3a11f11f adfa 4264 aa62 eeb45dbc7c7b' for connections to the remote server.
Info: Adding child site (MSDeploy.webServer/webServer/appHostConfig[@path='']/location[@path='']/section[@name='system.applicationHo
```

```

st/sites']/sites/site[@name='Default Web Site']/application[@path=
'/_labs']).  

Info: Adding child application (MSDeploy.webServer/webServer/appHo
stConfig[@path='']/location[@path='']/section[@name='system.applic
ationHost/sites']/sites/site[@name='Default Web Site']/application
[@path='/_labs']/virtualDirectory[@path='']).  

Info: Adding directory (C:\inetpub\wwwroot\labs).  

Info: Adding directory (C:\inetpub\wwwroot\labs\bin).  

Info: Adding file (C:\inetpub\wwwroot\labs\bin\healthcheck.txt).  

Info: Adding file (C:\inetpub\wwwroot\labs\bin\labs01.dll).  

Info: Adding file (C:\inetpub\wwwroot\labs\Default.aspx).  

Info: Adding file (C:\inetpub\wwwroot\labs\Lab1.aspx).  

Info: Adding file (C:\inetpub\wwwroot\labs\Site.Master).  

Info: Adding file (C:\inetpub\wwwroot\labs\Web.config).  

Info: Adding child sites (MSDeploy.webServer/webServer/appHostConf
ig[@path='']/location[@path='']/section[@name='system.applicationH
ost/sites']/sites/site[@name='labs']).  

Info: Adding child bindings (MSDeploy.webServer/webServer/appHostC
onfig[@path='']/location[@path='']/section[@name='system.applicati
onHost/sites']/sites/site[@name='labs']/bindings/binding).  

Info: Adding child bindings (MSDeploy.webServer/webServer/appHostC
onfig[@path='']/location[@path='']/section[@name='system.applicati
onHost/sites']/sites/site[@name='labs']/bindings/binding).  

Info: Updating logFile (MSDeploy.webServer/webServer/appHostConfig
[@path='']/location[@path='']/section[@name='system.applicationHos
t/sites']/sites/site[@name='labs']/logFile).  

Info: Adding child site (MSDeploy.webServer/webServer/appHostConfig
[@path='']/location[@path='']/section[@name='system.applicationHo
st/sites']/sites/site[@name='labs']/application[@path='']).  

Info: Adding child application (MSDeploy.webServer/webServer/appHo
stConfig[@path='']/location[@path='']/section[@name='system.applic
ationHost/sites']/sites/site[@name='labs']/application[@path='']/
virtualDirectory[@path='/']). Info: Adding appPoolConfig (labs).  

...
Total changes: 796 (403 added, 309 deleted, 84 updated, 0 paramete
rs changed, 636318 bytes copied)

```

Com o comando `getDependencies`, é possível verificar todos os componentes necessários para que a aplicação funcione em outro servidor. Ou seja, uma lista de pré-requisitos a serem configurados antes da migração.

```

msdeploy
verb:getDependencies
source:webServer ,computername=[FQDN servidor destinatário]

```

O resultado será semelhante a este:

```

Info: Using ID 'dce2747e 228c 4aa1 961f 0abf17a3c9c0' for connections to the remote server.
<output>
    <dependencyInfo>
        <dependencies>
            <dependency name="HttpCompressionStatic" />
            <dependency name="DefaultDocument" />
            <dependency name="DirectoryBrowse" />
            <dependency name="StaticContent" />
            <dependency name="AnonymousAuthentication" />
            <dependency name="RequestFiltering" />
            <dependency name="HttpErrors" />
            <dependency name="HttpLogging" />
            <dependency name="RequestMonitor" />
            <dependency name="ISAPIExtensions" />
            <dependency name="ISAPIFilter" />
            <dependency name="NetFxExtensibility" />
            <dependency name="AspNet4.0" />
            <dependency name="HttpTracing" />
            <dependency name="AspNet2.0" />
        </dependencies>
        <apppoolsInUse>
            <apppoolInUse name="DefaultAppPool" definitionIncluded
="True" />
        </apppoolsInUse>
        <managedTypes>
            <managedType type="System.ServiceModel.Activation.ServiceHttpModule,
System.ServiceModel.Activation, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
            <managedType type="System.Web.Routing.UrlRoutingModule" />
            <managedType type="System.Web.Handlers.ScriptModule,
System.Web.Extensions, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
            <managedType type="System.Web.Script.Services.ScriptHandlerFactory,
System.Web.Extensions, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
            <managedType type="System.ServiceModel.Activation.ServiceHttpHandlerFactory,
System.ServiceModel.Activation, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
            <managedType type="System.Web.HttpForbiddenHandler" />
            <managedType type="System.Web.Handlers.ScriptResourceHandler,
System.Web.Extensions, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
            <managedType type="System.Web.Handlers.TransferRequestHandler" />
        </managedTypes>

```

```
</dependencyInfo>  
</output>
```

Migração entre Servidor com IIS6 e IIS7+

"E se quisermos migrar entre ambientes com versões de IIS diferentes?"

Este é mais um dos cenários comuns para o uso do web deploy, principalmente com o final do suporte ao Windows Server 2003 e o IIS6. Para este cenário, é possível efetuar a migração com o seguinte comando:

```
msdeploy  
verb:sync  
source:webServer60,  
computername=[FQDN servidor destinatário],  
username=[usuário administrador servidor origem],  
password=[senha usuário]  
dest:webServer60,  
computername=[FQDN servidor destinatário],  
username=[usuário administrador servidor destino],  
password=[senha usuário]
```

É possível notar que o novo servidor também usa o provider `webServer60`. Isto não significa que o modo de compatibilidade IIS6 no IIS7+ precisa ser habilitado, pelo contrário, o Web Deploy trata a compatibilidade com o metabase internamente e permite o uso da nova arquitetura do IIS.

Resumindo

O Web Deploy possui muitas outras funcionalidades, como a sincronização de *assemblies* no GAC, suporte a criptografia de seções dos arquivos de configuração, backup e restore. Essa ferramenta pode ser muito útil, além de poder auxiliar na automação e adesão aos processos, como na gestão de aplicações web de forma simples e prática.

REFERÊNCIAS

Event Tracing -	https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx
IIS Metabase -	https://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/43a51d34-7c81-413b-9727-ec9a19d0b428.mspx?mfr=true
Internet Information Services (IIS) -	https://msdn.microsoft.com/en-us/library/ee532514(v=vs.90).aspx
Walkthrough: Using MSBuild -	https://msdn.microsoft.com/en-us/library/dd393573.aspx
Web Deploy webServer60 Provider -	https://technet.microsoft.com/en-us/library/dd569007(v=ws.10).aspx

6.11 MELHORES PRÁTICAS AO ESCREVER EXPRESSÕES REGULARES

Por Vinícius dos Santos Martins

Expressões regulares promovem uma maneira poderosa e eficiente para processamento textual, permitindo que grandes quantidades de texto sejam analisadas, encontrar caracteres que respeitam um certo padrão, validar um texto e garantir que este esteja de acordo um formato pré-definido (por exemplo, endereço de e-mail), ou substituir/inserir/remover uma parcela de texto dentro de um outro texto maior.

Na maioria dos casos, a avaliação desses padrões acontece de maneira muito eficiente. Entretanto, em casos extremos, a análise de um padrão pode levar um tempo considerável para terminar, ou até mesmo deixa de responder enquanto processa um texto relativamente pequeno.

Levando a performance em consideração, a seguir estão compiladas algumas das melhores práticas que podem ser aplicadas para garantir que a execução das expressões regulares nas aplicações desempenhe com performance.

Considere o tipo de entrada

A expressão regular pode aceitar entradas que já obedecem uma formatação pré-definida ou entradas que não respeitam qualquer estrutura. Uma expressão projetada para validar entradas pré-formatadas podem não ser tão performáticas quando se deparam com entradas nas quais essa formatação não está presente.

A expressão `^[0-9A-Z]([-.\w]*[0-9A-Z])*$`, por exemplo, é comumente utilizada para validar endereços de e-mail. Uma entrada válida de acordo com o padrão deve conter um caractere alfanumérico seguido de zero ou mais caracteres que podem ser alfanuméricos, pontos ou hifens, e deve terminar com um caractere alfanumérico.

```
Stopwatch sw;
```

```
string[] listaDeEmail = { "AAAAAAA@contoso.com",
    "AAAAAAAaaa@contoso.com"};
string pattern = @"^[\w][\w]*[\w]$";
string input;

StringBuilder builder = new StringBuilder();
foreach (var email in listaDeEmail) {

    string mailbox = email.Substring(0, email.IndexOf("@"));
    int index = 0;
```

```

    for (int ctr = mailBox.Length - 1; ctr >= 0; ctr--) {
        index++;

        input = mailBox.Substring(ctr, index);
        sw = Stopwatch.StartNew();
        Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
        sw.Stop();
        if (m.Success)
            builder.AppendFormat("{0,2}. Encontrado &#39;{1,25}&#39; em {2}",
9; em {2}", index, m.Value, sw.Elapsed);

        else
            builder.AppendFormat("{0,2}. Falhou &#39;{1,25}&#39; e m {2}",
index, input, sw.Elapsed);
    }
}

Console.WriteLine(builder.ToString());
Console.Read();

```

Como é possível notar nesse código, o endereço que concorda com o padrão (`AAAAAAAAAAAA@contoso.com`) é processado muito rapidamente, não importando o tamanho da string. Porém, quando o segundo endereço (`AAAAAAAAAaaaaaaaa!@contoso.com`) é avaliado, o tempo de processamento começa a aumentar consideravelmente, mostrando que a performance diminui conforme o tamanho da string, e atestando a ineficiência daquela expressão “`^[\0-9A-Z]([-. \w]*[\0-9A-Z])*$`” sendo utilizada.

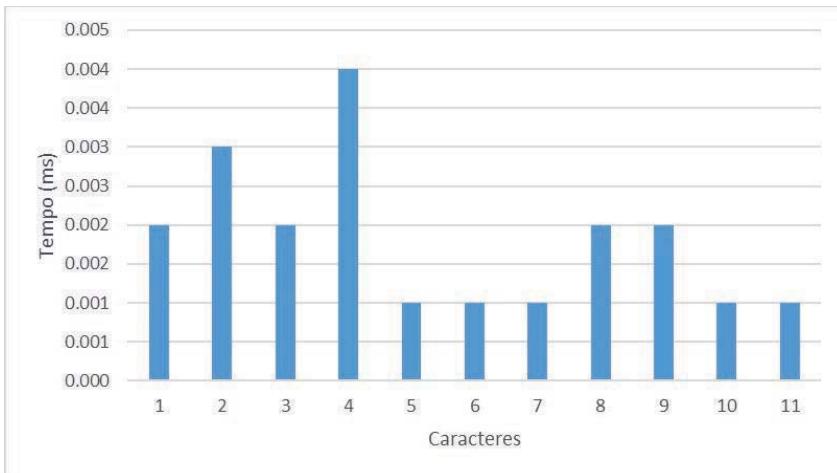


Figura 6.45: Tempo decorrido no email "AAAAAAAAAAAA@contoso.com"

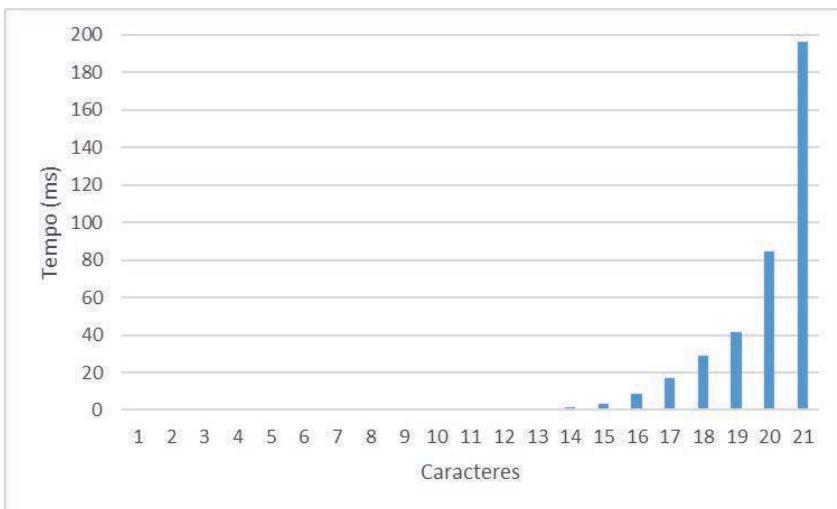


Figura 6.46: Tempo decorrido no email "AAAAAAAAAAAaaaaaaa!@contoso.com"

Para endereçar esse tipo de problema, é recomendado que se avalie como o *backtracking* pode afetar a performance, principalmente para aquelas entradas sem uma pré-formatação. Outra recomendação é testar rigorosamente usando entradas tanto válidas quanto quase válidas e, de acordo com os resultados, ajustar

a expressão.

Lide corretamente com a engine do Regex

A forma como o método `Match` da classe `Regex` é chamado pode impactar (e muito) na performance da aplicação. A seguir, serão discutidas chamadas estáticas ao método `Match`, a utilização de expressões interpretadas e a aplicação de expressões compiladas bem como os cenários em que cada abordagem provê um ganho ótimo de performance.

Expressões regulares estáticas

Fazer chamadas ao método estático `Match` da classe `Regex` é recomendado para situações nas quais a mesma expressão regular será usada diversas vezes. A expressão é compilada e guardada em um cache para rápido acesso, eliminando a necessidade de recompilação. Ao contrário do que acontece ao instanciar um objeto `Regex` diversas vezes, pois a expressão precisa ser recompilada a cada execução do método `Match` da instância criada.

Um cenário em que o uso dessa abordagem traria ganhos de performance seria, por exemplo, um *event handler* que é chamado para validar uma entrada vindo do usuário, como é mostrado no código:

Chamada ao método estático Match

```
public void button_Click(object sender, EventArgs e){  
    if (!string.IsNullOrEmpty("ALGUM TEXTO")) {  
        if (Regex.IsMatch("ALGUM TEXTO", "ALGUM PADRAO")) {  
            // TODO  
        }  
        else {  
            // TODO  
        }  
    }  
}
```

Expressões interpretadas e expressões compiladas

As expressões regulares, na plataforma .NET, podem ser construídas de três formas diferentes, podem serem interpretadas, compiladas e também compiladas para um *assembly* diferente.

1. Expressões interpretadas

Quando um objeto `Regex` é instanciado ou quando o método estático `Match` é chamado, e a expressão regular não existe no cache, a engine converte a expressão para `OPCODES`. Quando o método `Match` é chamado, esses `OPCODES` são convertidos para `IL` (linguagem intermediária) e, então, o compilador `JIT` (Just-in-time) executa o resultado. Esse é o tipo padrão ao criar um objeto `Regex`.

Essa opção tem como vantagem o tempo reduzido de inicialização. Porém, o tempo geral de execução é mais devagar, sendo ideal para casos nos quais se espera uma baixa frequência de chamadas aos métodos que executarão as avaliações para uma expressão regular específica.

2. Expressões compiladas

São expressões semelhantes ao modo interpretado, porém, a diferença é que a expressão regular é convertida diretamente para `IL`, e então executada pelo `JIT`. Esse tipo de expressão pode ser definida usando o atributo `RegexOptions.Compiled`, como demonstram os códigos::

Atributo Compiled na instância de um objeto Regex

```
Regex objetoRegex = new Regex("ALGUM PADRAO", RegexOptions.Compiled);
```

Atributo Compiled na chamada estática do método Match

```
var matches = Regex.Match("ALGUM TEXTO", "ALGUM PADRAO", Re
```

```
gex.Options.Compiled);
```

Expressões compiladas proveem um tempo de execução mais rápido em troca de um tempo de inicialização um pouco mais demorado, se mostrando ideal para cenários em que se sabe de antemão que a quantidade de chamadas pode variar substancialmente, podendo ser desde algumas até milhares.

3. Expressões compiladas para um assembly diferente

O método estático `Regex.CompileToAssembly` compila uma ou mais expressões regulares para outro assembly, podendo ser usadas somente por meio de instanciação de um objeto da classe gerada. A seguir, é demonstrada a criação bem como utilização dessa abordagem.

```
RegexCompilationInfo SentencePattern =
    new RegexCompilationInfo("ALGUM PADRAO",
        RegexOptions.Multiline,
        "NomeDoPadrao",
        "NomeDaClasse",
        true);
RegexCompilationInfo[] regexes = { SentencePattern };
AssemblyName assemName = new AssemblyName("RegExLib, Version=1.0.0.1001, Culture=neutral, PublicKeyToken=null");
Regex.CompileToAssembly(regexes, assemName);
```

Figura 6.47: Criação do Assembly com a expressão regular compilada

! [Utilização da expressão regular compilada em outro assembly] (images/Images6/cont_image24.png)

Apesar de oferecer uma performance consistente tanto no tempo de inicialização quanto no tempo de execução, essa alternativa de construção de expressões regulares se mostra inflexível em alterações nas configurações que foram definidas em tempo de desenvolvimento. Sem contar que todas as expressões precisam ter sido definidas antecipadamente, invalidando construções dinâmicas.

Controlando o backtracking

Uma das características das expressões regulares na plataforma .NET é a habilidade de executar de uma maneira não linear. Ou seja, em vez de avançar um caractere após o outro, é possível retornar para um estado passado a fim de avaliar a combinação de um

padrão. Essa habilidade é conhecida como *backtracking*.

Embora o backtracking traga poder e flexibilidade, caso seja usado excessivamente, pode impactar seriamente a performance da aplicação, deixando toda a responsabilidade de design nas mãos do desenvolvedor.

O backtracking acontece nas situações em que:

- Um *alternate constructor* (por exemplo, `(\w|\d)`) falha na primeira condição;
- Quantificadores indeterminados (`*`, `?`, `+`) são encontrados no texto;
- Expressões condicionais falham na primeira condição.

Quando um dos três cenários está presente, o estado atual é salvo para que seja possível voltar a ele caso haja necessidade. Geralmente, aplicações sofrem uma penalidade na performance por usar backtracking, apesar do fato de que ele não é essencial em alguns casos, para que haja uma combinação bem-sucedida.

Exemplificando, a expressão `\b\p{Lu}\w*\b` identifica palavras que começam com uma letra maiúscula seguida por zero ou mais caracteres (por exemplo, *Contoso*). Por conta que o quantificador `*` está presente, para cada caractere válido, o estado é salvo para caso seja necessário abandonar um estado inválido (causado por um caractere inválido) e voltar para o último estado válido salvo.

O padrão `\b` identifica os limites de uma palavra e, por conta disso, não faz parte dos caracteres que identificam uma palavra (identificados pelo padrão `\w`). Portanto, não há como se deparar com um `\b` ao avaliar o padrão `\w`. Ou seja, neste caso, o backtracking não contribui na identificação de uma combinação válida.

Nos cenários onde o backtracking não é necessário, é possível desligá-lo usando a construção `?>SUBEXPRESSÃO` onde não deve ser aplicado o backtracking na SUBEXPRESSÃO . Assim, alterando a expressão anterior, obtemos `\b\p{Lu}(?>\w*)\b` .

Em muitos casos, entretanto, o backtracking é necessário para obter uma combinação positiva. Porém, seu uso excessivo é prejudicial para a performance. Em particular, isso acontece quando quantificadores são aninhados, e o texto que está de acordo com a subexpressão externa também é um subconjunto do texto que combina com a subexpressão interna.

Por exemplo, a expressão `^[\0-9A-Z]([-.\w]*[\0-9A-Z])*`$``, na qual o caractere que combina com `[\0-9A-Z]` também é um subconjunto da expressão `[-.\w]*` .

Código para validar a expressão `^[\0-9A-Z]([-.\w][\0-9A-Z])*`$``*

```
String pattern = @"^[\0-9A-Z]([-.\w]*[\0-9A-Z])*`$`";
Regex rgx = new Regex(pattern, RegexOptions.IgnoreCase);
Stopwatch sw;

string[] items = { "A163.1523C$",
    "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa#" };
string toValidate, result;
StringBuilder builder = new StringBuilder();

foreach (var item in items) {
    for (int ctr = 1; ctr <= item.Length; ctr++) {
        toValidate = item.Substring(0, ctr);
        sw = Stopwatch.StartNew();

        if (rgx.IsMatch(toValidate))
            result = "Valido";
        else
            result = "Nao valido";

        sw.Stop();
        builder.AppendFormat("{0} com {1} caracteres {2}",
            result,
            ctr,
```

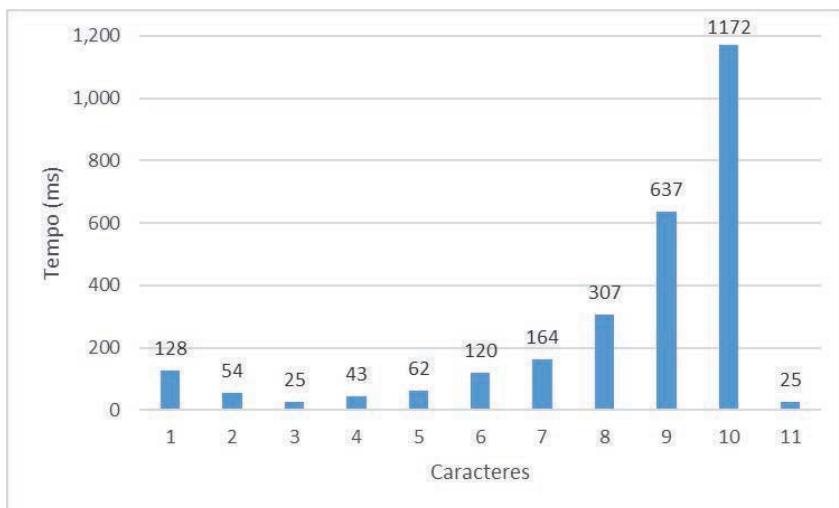
```

        sw.Elapsed);
    }
}
Console.WriteLine(builder.ToString());

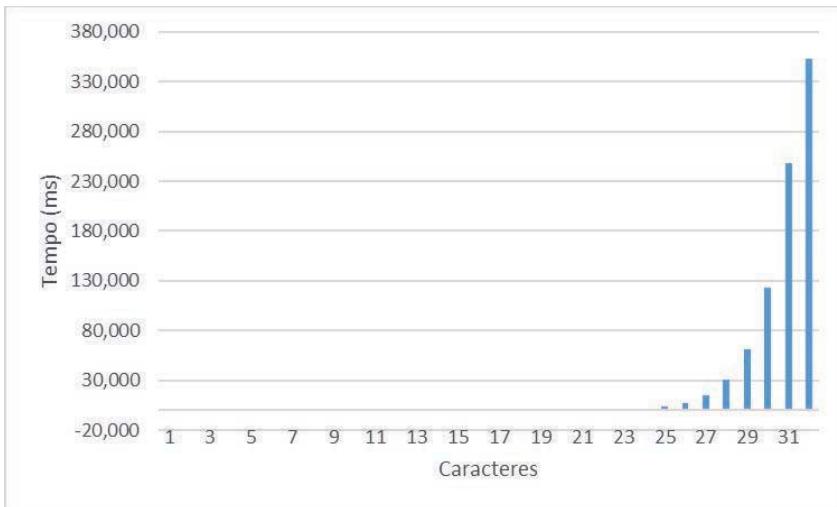
```

Como é possível perceber nos gráficos a seguir, a expressão é capaz de desempenhar bem quando encontra um valor válido. Porém, a performance se degrada rapidamente se um valor inválido é encontrado.

Desempenho da expressão `^[0-9A-Z]([-.\w]*[0-9A-Z])*$$` ao avaliar a entrada `A163.1523C$"]([-.\w]*[0-9A-Z])*$$`:



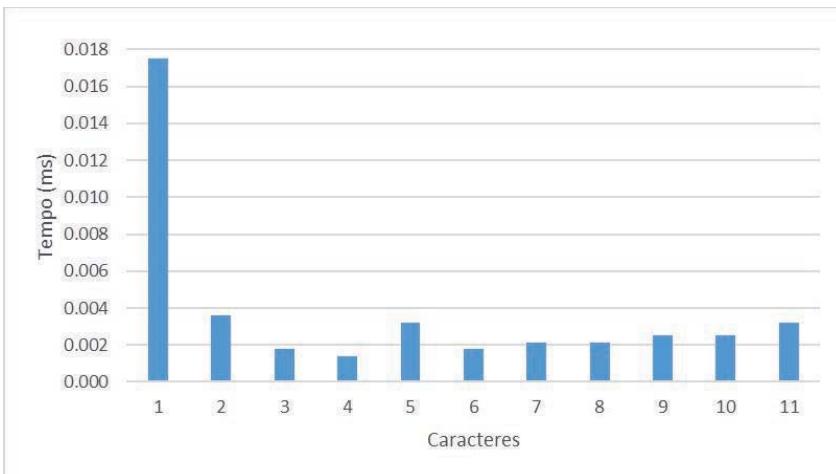
Desempenho da expressão `^[0-9A-Z]([-.\w]*[0-9A-Z])*$$` ao avaliar a entrada `aaaaaaaaaaaaaaaaaaaaaaaaaaa#`:



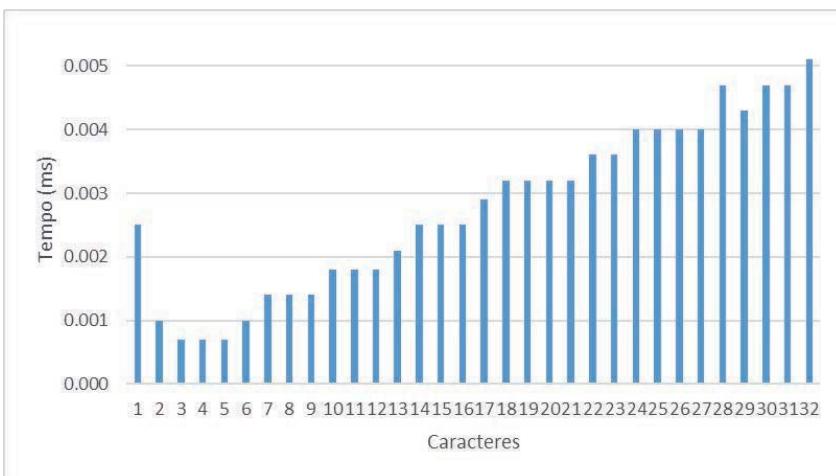
Para otimizar esse tipo de situação, é possível substituir os quantificadores aninhados por asserções *look-ahead* ou *look-behind*. Essas asserções não movem de posição o ponteiro na string, mas, em vez disso, olham para a frente ou imediatamente para atrás da posição atual para determinar se o padrão definido é verdadeiro ou não.

A expressão anterior pode, então, ser modificada, para `^[0-9A-Z][- .\w]*(?<=[0-9A-Z])\$$.` Nela, é usada uma asserção *look-behind* positiva quando o símbolo `$` é validado para garantir que o caractere imediatamente antes do `$` é, ou numérico, ou alfanumérico. Após aplicada a modificação na expressão, como mostram os gráficos a seguir, é visível o ganho de performance.

Desempenho da expressão `^[0-9A-Z][- .\w]*(?<=[0-9A-Z])\$$.` ao avaliar a entrada `A163.1523C$` :



Desempenho da expressão regular $^{\text{[0-9A-Z]}}[\text{ - . \w}]*(\text{?}<=[\text{0-9A-Z}])\$\$$ ao avaliar a entrada aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa# :



REFERÊNCIAS

- Backtracking in Regular Expressions -
[https://msdn.microsoft.com/en-us/library/dsy130b4\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dsy130b4(v=vs.110).aspx)
- Best Practices for Regular Expressions in the .NET Framework -
[https://msdn.microsoft.com/en-us/library/gg578045\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/gg578045(v=vs.110).aspx)
- Details of Regular Expression Behavior -
<https://msdn.microsoft.com/en-us/library/e347654k.aspx>
- Optimizing Regular Expression Performance, Part I -
<https://blogs.msdn.microsoft.com/bclteam/2010/06/25/optimizing-regular-expression-performance-part-i-working-with-the-regex-class-and-regex-objects-ron-petrusha/>
- Optimizing Regular Expression Performance, Part II -
<https://blogs.msdn.microsoft.com/bclteam/2010/08/03/optimizing-regular-expression-performance-part-ii-taking-charge-of-backtracking-ron-petrusha/>
- GOYVAERTS, J.; LEVITHAN, S. *Regular Expressions Cookbook*. 2. ed. O'Reilly Media, 2012.

6.12 CONCLUSÃO

Este capítulo apresentou práticas adotadas mundialmente pelos times de desenvolvimento. Dentre elas, notamos ações voltadas para melhorar o desempenho de aplicações, assim como práticas para aperfeiçoar as entregas das equipes. As práticas aqui discutidas objetivam a evolução contínua das aplicações e o aumento da

maturidade dos times de desenvolvimento.

CAPÍTULO 7

BIBLIOGRAFIA

CWALINA, K.; ABRAMS, B. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries.* ed. 2. Boston: Addison Wesley, 2009.

EVANS, Eric. *Domain-Driven Design.* ed. 1. Addison-Wesley Professional, 2003.

FOWLER, Martin. *Data Transfer Object.* 2013. Disponível em: <http://martinfowler.com/eaaCatalog/dataTransferObject.html>. Acesso em: 19 jan. 2016.

GOYVAERTS, J.; LEVITHAN, S. *Regular Expressions Cookbook.* 2. ed. O'Reilly Media, 2012.

HUMBLE, Jez; FARLEY, David. *Continuous Delivery:* reliable software releases through build, test, and deployment automation. Addison-Wesley, 2010.

LIPPERT, Eric. *The Truth About Value Types.* 2010. Disponível em: <https://blogs.msdn.microsoft.com/ericlippert/2010/09/30/the-truth-about-value-types/>. Acesso em: 19 jan. 2016.

MARTIN, Robert C. *SOLID: Principles of Object Oriented Design.* 2000. Disponível em: <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>. Acesso em: 19 jan. 2016.

MYERS, Glenford. *The Art of Software Testing.* New York:

Wiley, 2015.

POPENDIECK, M; POPPENDIECK, T. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley, 2006.

RICHTER, J. *CLR via C# - Developer Reference*. 4. ed. Microsoft Press, 2012.

SKEET, Jon. *C# in Depth*. 3. ed. New York: Manning, 2013.

SKEET, Jon. *Memory in .NET: what goes where*. 2014. Disponível em: <http://jonskeet.uk/csharp/memory.html>. Acesso em: 19 jan. 2016.

SPOSITO, Dino; SALTARELLO, Andrea. *Microsoft .NET:- Architecting Applications for the Enterprise*. Microsoft Press, 2008.

YOURDON, Edward; CONSTANTINE, Larry. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press, 1976.

CAPÍTULO 8

SOBRE OS AUTORES

Microsoft Modern Apps Premier Field Engineers



Adilson Coutrin possui bacharelado em Ciência da Computação e MBA em Cloud Computing pela FIAP, além de certificações em Cloud Computing e suporte. Possui 13 anos de experiência em gerenciamento de infraestrutura baseadas em tecnologias Microsoft. Adilson aproveita seu tempo livre com sua esposa assistindo séries, vendo filmes e lendo. Além disso, gosta de acompanhar seu time, o São Paulo F.C.



Alexandre Campos iniciou essa jornada em ALM em 2006, ao iniciar o processo de adoção de XP em uma pequena equipe de 12 programadores. Foi quando começou a utilizar técnicas como Integração Contínua e Teste Unitário. A partir de 2007, passou a adotar o TFS, então na versão 2005, como a ferramenta preferida para suportar o ALM. Em 2009, passou a atuar como Scrum Master e em projetos de Coaching e Implementação de Scrum. Em 2013, aceitou a oferta para atuar como Premier Field Engineer na Microsoft, onde ajuda os clientes Premier a entregar mais valor de negócio na forma de software. Nas horas vagas, Alexandre brinca de Kanban para seu filho Thiago de 3 anos, e joga jogos de tabuleiro nas madrugadas.



Alexandre Teoi é formado em Engenharia de Computação pelo Instituto Tecnológico de Aeronáutica, e trabalha com

desenvolvimento de software há mais de 20 anos. Nesse período, teve a oportunidade de trabalhar com várias linguagens (C, C++, C#), criando software para rodar nos mais diversos sistemas operacionais (MS-DOS, Windows, Unix, Windows NT). Atualmente, trabalha como Premier Field Engineer especializado em Modern Apps na Microsoft. Nas horas livres, gosta de conhecer novos restaurantes com a esposa e dois filhos, e jogar Street Fighter.



Bruno Lins de Oliveira atua com desenvolvimento e suporte ao desenvolvimento de software, em diferentes ramos, há mais de 10 anos. Possui diversas certificações Microsoft voltadas para desenvolvimento. Bruno aproveita seu tempo livre com sua família e assistindo séries de TV.



Christiano Donke é especialista em arquitetura e construção de software. Começou a desenvolver aos 12 anos, e há 10 anos atua

profissionalmente na área, sempre focado e baseado em tecnologias Microsoft. Em seu tempo livre, Christiano aproveita o tempo com sua esposa Fernanda, saindo com amigos, fazendo algo em família, ou desenvolvendo algo para estudar.



Demetrio Rafael Neri Costa é um profissional certificado com mais de 10 anos de experiência no desenvolvimento e suporte de soluções. Gosta muito de cinema, bons livros, e passar o tempo com os amigos e familiares.



Felipe Fujiy Pessoto possui bacharelado em Ciência da Computação, atua com desenvolvimento de software há mais de 10 anos, em sua maioria com tecnologias Microsoft, possuindo diversas certificações. Participa de comunidades online, contribuindo em projetos Open Source, fóruns, artigos em blog e revistas.



Fernando Filiputti possui bacharelado em Ciência da Computação. Possui 12 anos de experiência em desenvolvimento de soluções baseadas em tecnologia Microsoft, e atualmente tem ajudado os clientes na adoção das melhores práticas de Application Lifecycle Management e DevOps, usando Team Foundation Server e Visual Studio Team Services. Fernando aproveita seu tempo livre com sua esposa Tania e seu filho João Pedro, ouvindo Pink Floyd e acompanhando o Palmeiras.



Fernando Henrique Inocêncio Borba Ferreira possui bacharelado e mestrado em Sistemas de Informação. Atuando com desenvolvimento de software há mais de 10 anos, ele também foi Microsoft MVP em Data Platform Development e Visual C#. Já trabalhou em expedições científicas na floresta Amazônica e foi campeão de competições de desenvolvimento, como o Microsoft

Imagine Cup. Fernando usa seu tempo livre lendo, tocando bateria e trabalhando em pesquisas acadêmicas.



Henrique Rezende da Silva possui bacharelado em Computer Systems Engineering pela City University de Londres. Atua com foco em IIS desde 2007. Atualmente, além de suporte técnico, ele também ministra workshops para os clientes Premier da Microsoft. Em seu tempo livre, gosta de curtir com a sua esposa Fernanda e seus filhos Nina e Gabriel, além de também ter o hobby de mestre cervejeiro.

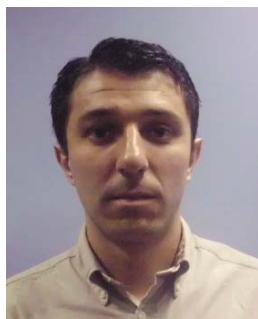


Iury Oliveira possui bacharelado em Administração de Empresas e Ciência da Computação. Atua com desenvolvimento e suporte ao desenvolvimento de software há quase 15 anos. Foi Microsoft Certified Trainer e ministra palestras em diversos

eventos. Em seu tempo livre, gosta de ler, assistir filmes, ir ao Parque do Ibirapuera e passar seu tempo com o pequeno João, seu primeiro filho.



Leandro de Almeida Santos é bacharel em Sistemas de Informação. Atuando no suporte a produtos Microsoft há mais de 10 anos, participou de diversos projetos de migração em grandes empresas. Hoje, ele agradece a Deus e a todos que o ajudaram chegar onde está. Em seu tempo livre, gosta de sair para jantar com sua esposa e sua família, assistir filmes e encontrar os amigos.



Leandro Prado (vulgo “Pia”) trabalhou com projetos de desenvolvimento desde o inicio do .NET Framework. Desde então, acompanha a evolução da plataforma. Leandro aproveita seu tempo livre tocando bateria, jogando futebol e participando de corridas com seu carro.



Luís Henrique Demetrio possui bacharelado em Análise de Sistemas e pós-graduação em Engenharia de Software. Possui 15 anos de experiência na profissão. Atuou 5 anos no mercado financeiro com desenvolvimento através do maior banco de atacado da América Latina antes de iniciar a carreira na Microsoft. Trabalhando como engenheiro de suporte para clientes Premier da Microsoft, é responsável por diversas iniciativas de educação como workshop, webcasts e palestras, principalmente envolvendo Visual Studio e técnicas de troubleshooting.



Luiz Fernando de Macedo atua como Premier Field Engineer em Modern Apps, especializado em Visual Studio, ALM e DevOps. Atua com desenvolvimento de software há mais de 10 anos, possui certificações Microsoft voltadas para Desenvolvimento, ALM e Cloud Computing e foi MCT. Orgulha-se a cada dia do que faz e de

onde está, além de ter o maior orgulho por ter uma linda esposa Gabi, a qual o acompanha em todos os momentos felizes. Passa boa parte do tempo lendo, jogando Xbox One e, quando pode, pratica mergulho, conhecendo assim um mundo novo.



Rafael Teixeira possui bacharelado em Engenharia da Computação. Atua com desenvolvimento de software há mais de 12 anos, se especializando em Application Performance e Troubleshooting, e atuando em projetos para automatização de análises, como extensões para Windbg e DebugDiag. Rafael gosta de aproveitar seu tempo livre praticando esportes, como futebol e kart, lendo e jogando videogame.



Ricardo Serradas possui bacharelado em Ciência da Computação. Trabalha na área de tecnologia há mais de 12 anos. Seus primeiros passos foram na área de infraestrutura. Entretanto,

quando descobriu o poder que o desenvolvimento de software traz, decidiu especializar-se. Serradas ocupa seu tempo vago contribuindo com a comunidade técnica, tocando guitarra, assistindo seriados ou correndo no parque, sempre na companhia da sua noiva Patrícia.



Robson Rocha de Araújo atua com desenvolvimento e arquitetura de software a mais de 15 anos. Ele possui mais de dezoito certificações Microsoft voltadas para Cloud, Web e banco de dados, e atuou por cinco anos como MCT na região de Campinas, interior de São Paulo. Orgulhoso pai de três filhos, Robson passa seu tempo livre jogando videogame, assistindo desenhos, brincando com seus dois meninos, e assistindo séries e filmes com a esposa quando a bebê está dormindo.



Sérgio Ramos é funcionário da Microsoft desde maio de 1999,

sempre envolvido nas atividades de desenvolvimento de código, tendo como paixão as tecnologias de Sistema Operacional. Sergin, como é conhecido pelos amigos, usa a maior parte do tempo livre tentando sobreviver a algum tipo de aventura que sua esposa Lília sempre descobre para eles.



Thiago Camargos Lopes possui bacharelado em Ciência da Computação. Iniciou sua vida profissional como desenvolvedor em ferramentas e tecnologias Open Source. Após conhecer o Visual Studio, se apaixonou pelas ferramentas e tecnologias Microsoft. É especializado em Visual Studio, Team Foundation Server e práticas de ALM e DevOps.



Tiago Soczek possui bacharelado em Sistemas de Informação e atua com desenvolvimento de software há mais de 10 anos. É apaixonado por desenvolvimento web e código bem escrito. Em seu

tempo livre, gosta de correr, estudar e viajar com sua adorável esposa, Camila.



Vinícius Martins possui bacharelado em Ciência da Computação e 8 anos de experiência em desenvolvimento de aplicações com base nas tecnologias da Microsoft com foco em desktop e mobile. Além disso, possui diversas certificações Microsoft voltadas ao desenvolvimento. Em seus momentos de descanso, Vinícius aproveita para assistir filmes e séries, tocar violão e, ocasionalmente, jogar videogame.

Microsoft Azure Premier Field Engineer



Andreia Otto possui experiência em desenvolvimento de software e gestão de ciclo de vida de aplicações baseadas em tecnologia Microsoft há 8 anos. Atualmente, Andreia apoia clientes

na adoção e utilização de Azure, plataforma de nuvem da Microsoft. Baseada em Portugal, ela atende clientes em toda Europa Ocidental, entregando treinamentos de Azure e aproveita as viagens para conhecer novos lugares, culturas, pessoas e comidas. Andreia aproveita seu tempo livre na água, se dividindo entre natação de águas abertas e surf.

Microsoft Technical Solution Professional



Wanderson Lima atua como Technical Solution Professional (TSP) no time de Development Tools da Microsoft. Graduado em Ciência da Computação pela Universidade Federal de Uberlândia (UFU), trabalha a mais de 10 anos no mercado de TI, desempenhando vários papéis através das disciplinas da indústria de software, culminando assim na atuação com ALM e DevOps (Consultor, Premier Field Engineer e agora no time de Developer Experience (DX) da Microsoft). No tempo livre, gosta de se divertir com a noiva e retornar para Minas para rever família, amigos e o velho e bom pão de queijo mineiro.

Microsoft Support Engineer



Rodrigo Gallazzi S. Leite é bacharel em Ciência da Computação e atua com desenvolvimento de software com tecnologias Microsoft há mais de 14 anos. Atuou como PFE em Modern Apps, e suportou os clientes na adoção e execução de testes de carga, utilizando Microsoft Azure. Atualmente, Rodrigo atua como Support Engineer - Developer Tools. Ele aproveita seu tempo livre com sua esposa Priscila e com seus filhos Gabriel e Felipe, em pescarias e jogos de PC.

Microsoft Technical Account Manager



Ricardo de Almeida possui bacharelado em Sistemas de Informação e 9 anos de experiência em desenvolvimento de software com foco em tecnologias Microsoft. Já atuou como Premier Field Engineer em Modern Apps ajudando os clientes na

adoção das melhores práticas de Application Lifecycle Management e DevOps. Ricardo aproveita seu tempo livre com sua noiva Andressa, lendo, indo ao tetro ou praticando Kung Fu, corrida, entre outros esportes.