Integração Contínua

Boas práticas para facilitar um modelo de trabalho mais fluido



Fone: (21) 93174772

MSN: me@juanlopes.net

Login DevMedia: juanplopes@gmail.com

**Juan Lopes**

me@juanlopes.net

Arquiteto de soluções pela Living Consultoria. Iniciou a carreira desenvolvendo aplicações biométricas em C++ e Java. Programa principalmente em C# desde 2007, mas desenvolve também em Ruby e Python. É entusiasta da comunidade Microsoft e participa de projetos open source.

*Sugestão de Ilustração: Equipes esportivas, interseção de rios, acessos de entrada em rodovias, bifurcações em ferrovias.*

|  |
| --- |
| **De que se trata o artigo** |
| Este artigo trata de integração contínua. Um termo cunhado por Kent Beck em seu livro sobre Extreme Programming, que define uma prática aplicada na indústria há muito tempo. Consiste, basicamente, em diminuir o ciclo de integração e efetuá-lo tão freqüentemente quanto possível. |
| **Para que serve** |
| Numa equipe com muitas pessoas, gerenciar as mudanças de forma razoável pode ser muito difícil, visto que as diversas mudanças podem conflitar entre si. Quanto mais tempo se leva entre o desenvolvimento e a integração, mais difícil ela se torna. A integração contínua visa resolver esse problema, introduzindo práticas e ferramentas que facilitem a verificação e correção de erros de integração. |
| **Em que situação o tema é útil** |
| Este tema é especialmente útil ao se trabalhar com equipes com muitas pessoas, onde frequentemente as mudanças feitas por um membro impactam todos os outros. Em muitas empresas, definem-se esquemas antiquados para evitar erros de integração, como a aplicação de locks exclusivos no controle de versão. A integração contínua tenta resolver esse problema sem tirar a fluidez necessária a um processo de desenvolvimento saudável. |

[COLOQUE AQUI O MESMO TÍTULO DO ARTIGO]

Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo Corpo

Em um projeto de software, desenvolvedores trabalham gerando código-fonte e outros artefatos que compõem um sistema maior. Cada desenvolvedor realiza um conjunto de tarefas que normalmente tem relação com as tarefas que outros desenvolvedores estão realizando. Apesar de um software ser um sistema determinístico, o resultado da relação entre cada um dos componentes de um programa pode ser imprevisível.

Sabendo isso, fica claro o quão complicado pode ser trabalhar num software em equipe. Quanto maior a equipe, maior é o desafio. E quanto mais tempo um desenvolvedor trabalha sem integrar seu código com o de seus colegas, mais difícil será essa integração quando acontecer. Para diminuir esse problema, é necessário que a integração seja um processo freqüente.

A metodologia XP cunhou o termo “Integração Contínua”, como um dos 12 princípios de desenvolvimento de software. Desde então, muito tem se falado sobre este princípio nos mais diversos contextos. Isso acontece porque a prática da Integração Contínua permeia quase todos os processos do desenvolvimento de software, alimentando ou recebendo feedback de outras práticas.

**Nota do DevMan**

XP é Extreme Programming, uma metodologia ágil de desenvolvimento de software. Ela é baseada em uma série de princípios que se complementam, e.g., pair programming, TDD e iterações curtas. Foi criada por Kent Back, que escreveu um livro sobre o assunto em 1999. Desde então, XP tornou-se uma das metodologias ágeis mais famosas. Apesar de relativamente nova, seus princípios vem sido discutidos há muito tempo na indústria e na comunidade científica.

Muitos desenvolvedores argumentam que essa é uma prática muito custosa no processo de desenvolvimento e que, para aplicá-la, muitas mudanças precisam ser feitas. De fato, muitas vezes isso é verdade. Mas aplicar a Integração Contínua, assim como o TDD, não é o motivo de tal dificuldade, mas sim a negligência de outras boas práticas complementares.

A integração contínua não é o tipo de prática que se implante. É uma prática que se alcança através da implantação de outras. Várias práticas que tornam a integração contínua possível serão discutidas neste artigo.

Um ciclo de integração sustentável

É comum acreditar que a integração contínua acontece num servidor de build. Entretanto, a maior parte das práticas deve acontecer no dia-a-dia do desenvolvedor. E para que tudo isso funcione, deve existir um ciclo comum que defina como integrar o código.

Deve existir um repositório comum, acessado por todos os membros da equipe, que contenha todo o código já integrado até certo momento da vida de um software. Esse repositório central é chamado de “mainline”. Existem muitas estratégias sobre como gerenciar a mainline e as várias versões que ela guarda. Iremos discutir sobre isso mais a frente.

Para desenvolver uma funcionalidade, um membro da equipe deve copiar a última versão disponível na mainline, desenvolver em cima dela. Depois, deve integrar as modificações de seus colegas, e então devolver as modificações para a mainline. O ideal é que o desenvolvedor passe o mínimo tempo possível desconectado, pois quanto maior o tempo, maior a dificuldade para realizar a integração.

Parece um ciclo simples, a primeira vista. A grande dificuldade está em garantir que as modificações feitas se integrem com a versão já existente e com as modificações feitas em paralelo. Existem algumas práticas que facilitam esse trabalho. Muitas delas envolvem, na verdade, automatizar tarefas manuais.

O controle de versão deve ajudar, não atrapalhar

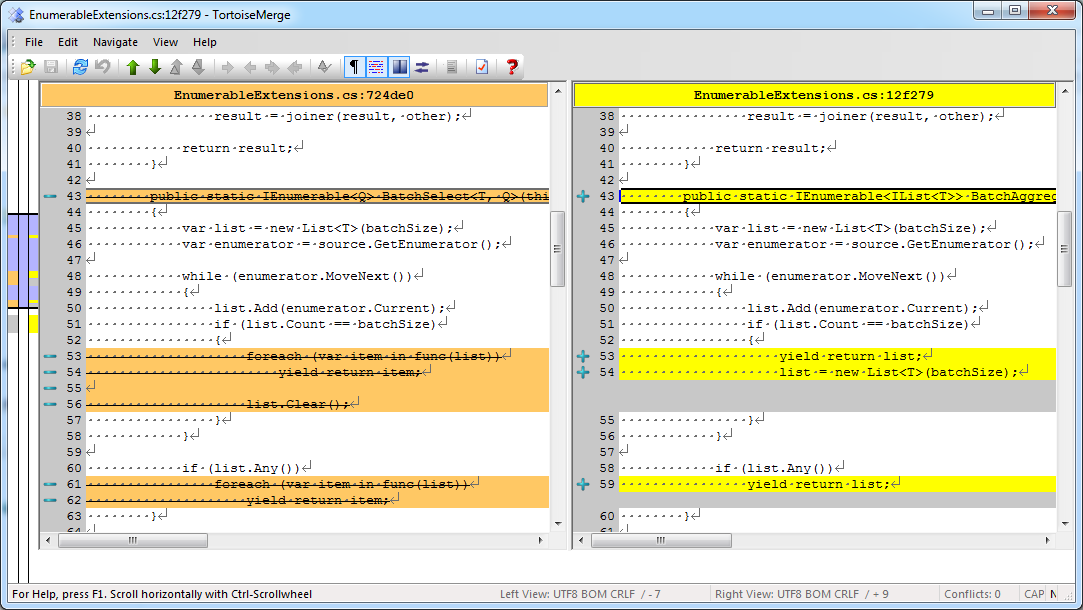
Por mais que pareça óbvio para a maioria dos leitores, utilizar um software de controle de versão é uma prática essencial para o ciclo de vida do desenvolvimento de software. Ele irá não só controlar o histórico de desenvolvimento como também irá ajudar a integrar as modificações mais facilmente.

Ferramentas como o Subversion, Mercurial e Git são amplamente utilizadas pelo mercado. Cada uma delas tem prós e contras que devem ser avaliados, pois impactam diretamente o ciclo de desenvolvimento da sua equipe. Escolher uma ferramenta como o Subversion, por exemplo, pode ser um problema quando muitos merges precisam ser efetuados.

Isso evidencia o primeiro problema ser resolvido em uma equipe: como gerenciar os conflitos? É fato que em certo ponto do desenvolvimento de software, duas ou mais pessoas poderão concorrer pela modificação de um mesmo artefato de código. Nesse ponto, existem primariamente duas opções: lock pessimista ou lock otimista.

Ao utilizar lock pessimista, assume-se que as mudanças que duas pessoas fizerem no código vão gerar conflitos entre si e, para garantir que isso não aconteça, permite-se que apenas uma pessoa modifique o código por vez. Essa opção é bastante famosa entre usuários de SCMs como Source Safe e TFS, e é conhecida como lock exclusivo. A outra opção (lock otimista) seria permitir as modificações, assumindo que as mudanças podem ser mescladas no final do desenvolvimento, seja de forma automática, seja de forma manual.

Na maior parte das vezes, a segunda opção torna o trabalho mais fluido. E graças aos mecanismos de merge atuais, são necessários menos merges manuais por parte do desenvolvedor.



**Figura 1.** Comparando dois commits (manualmente) com o TortoiseMerge

A segunda grande decisão com relação ao controle de versão é: o que versionar? Existe muita discussão nesse ponto. A regra é: tudo o que for necessário para transformar seu código em um software precisa estar no seu repositório. Isso pode incluir dependências, scripts de build, ferramentas de análise, entre outros. Um dos problemas mais comuns é assumir que controle de versão deve ser usado apenas para guardar código fonte. Isso torna a integração bem mais difícil, pois exige que cada membro da equipe configure um ambiente, muitas vezes complexo, apenas para começar a trabalhar.

Build automatizado

Durante o ciclo de desenvolvimento, uma das tarefas para verificar a integridade de certo ponto da mainline é transformá-lo em software executável. Para linguagens como o C#, isso geralmente significa compilar o código. Mas o processo de build não envolve só isso. Envolve também preparar um pacote completo que permita que o software seja distribuído. Esse processo pode se tornar muito complexo, conforme o sistema cresce. O objetivo é automatizar as tarefas envolvidas.

Tudo aquilo que é necessário para o software funcionar precisa estar incluído no build automatizado. Se o seu sistema utiliza um banco de dados, o seu build deve ser capaz de executar a última versão dos scripts. Também pode incluir chamadas a ferramentas de análise de código que irão emitir relatórios úteis sobre o progresso do projeto.

É preciso tomar cuidado, entretanto, pois um build com muitos passos pode demorar muito e acabar desanimado os desenvolvedores em executá-lo com freqüência. Pode ser importante quebrá-lo em partes menores e fazê-lo mais inteligente, de forma a somente executar para os artefatos que forem modificados.

Existem várias ferramentas para ajudar nessa automatização. A principal delas é o MsBuild, que vem incluído na maior parte das distribuições do .Net Framework. Mas além dela, existem outras, e.g., o NAnt (que é baseado no Ant, do Java), o PSake (escrito em PowerShell).

Uma das vantagens do MsBuild é que todo projeto C# (.csproj) é também um script nesta linguagem. Isso faz com que chamar um script do projeto seja mais fácil, muitas vezes. Mas, de uma forma geral, são ferramentas que diferem na sintaxe e sua escolha está mais relacionada à preferência pessoal do que a utilidade em si.

A **Listagem 1** mostra um exemplo de script de build em MsBuild. Ele possui 4 tasks: Clean, Build, Rebuild e Test. As três primeiras basicamente compilam o código usando o MsBuild em cima da solução original. A última executa o NUnit contra o assembly de testes.

**Listagem 1.** Script em MsBuild

<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" DefaultTargets="Test">

<PropertyGroup>

<BuildDir>$(MSBuildProjectDirectory)\build\</BuildDir>

<TestAssembly>$(BuildDir)HelloWorld.Tests.dll</TestAssembly>

<SourceDir>src\</SourceDir>

<Configuration>Release</Configuration>

<Platform>Any CPU</Platform>

</PropertyGroup>

<ItemGroup>

<SolutionFile Include="$(SourceDir)\HelloWorld.sln" />

</ItemGroup>

<Import Project="util\msbuild\MSBuild.Community.Tasks.Targets"/>

<Target Name="Clean">

<MSBuild Projects="@(SolutionFile)" Targets="Clean"

Properties="Configuration=$(Configuration);Platform=$(Platform);OutputPath=$(BuildDir)"/>

<DeleteTree Directories="$(BuildDir)" ContinueOnError="true"/>

</Target>

<Target Name="Build">

<MSBuild Projects="@(SolutionFile)"

Properties="Configuration=$(Configuration);Platform=$(Platform);OutputPath=$(BuildDir)"/>

</Target>

<Target Name="Rebuild">

<CallTarget Targets="Clean; Build"></CallTarget>

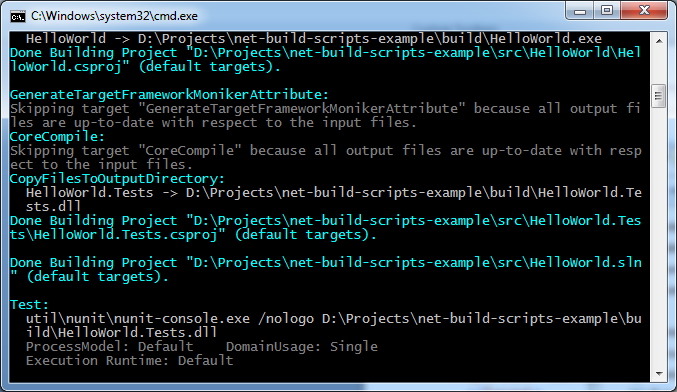
</Target>

<Target Name="Test" DependsOnTargets="Build">

<Nunit Assemblies="$(TestAssembly)" ToolPath="util\nunit"/>

</Target>

</Project>



**Figura 2.** MsBuild executando o script

Testes Automatizados

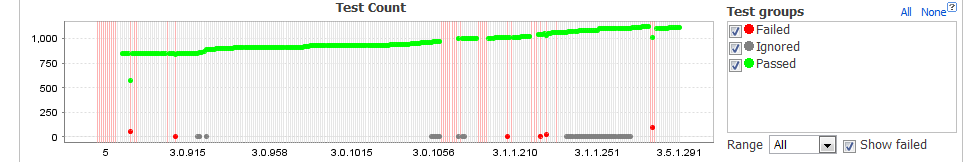
Apesar de ajudar, o build por si só não é capaz de garantir a integridade do software. Somente é possível verificar erros em um software exercitando-o. Essa é a função dos testes automatizados. É importante ressaltar que isso não necessariamente significa usar TDD, apesar de ser recomendado.

Os testes ajudam a descobrir muitos bugs no código. Apesar de não garantirem que o código não possui erros, são essenciais para verificar se uma mudança uma parte do software acabará afetando outras que dependem dela indiretamente.

Muita discussão existe sobre a escrita e manutenção de testes automatizados, e apesar da maioria dos desenvolvedores concordarem que os testes são úteis para aumentar a qualidade do código, ainda é não é consenso o seu real custo no processo de desenvolvimento.

Fato é que todo desenvolvedor testa de alguma forma o código que escreve. Muito desse teste é manual. Escrever testes automatizados facilita o processo de integração, pois permite que mesmo quem não conheça todo o fluxo de todos os componentes do sistema possa testá-los de forma rápida e intuitiva.

É importante que executar os testes seja parte da rotina de build. Assim você permite que seu código seja testado em qualquer ambiente automaticamente, garantindo que a última versão na mainline sempre passa por todos os testes escritos sem o programador sequer precisar se preocupar em rodá-los. A **Figura 3** mostra um exemplo de relatório emitido por um servidor de build TeamCity, mostrando a evolução da quantidade de testes através das versões.



**Figura 3.** Gráfico de quantidade de testes automatizados

Essa informação pode ser útil para relatórios de impacto também, visto que os testes exercitam partes fundamentais do seu código. A partir da execução deles, é possível analisar quais partes do seu sistema tem maior importância em alguns cenários de execução, prevendo, assim, onde os bugs seriam mais críticos ao aparecer.

Preparação de ambiente automatizada

Como é possível perceber, grande parte das práticas da integração contínua envolve basicamente automatizar tarefas outrora manuais. Não poderia ser diferente com a preparação do ambiente necessário para o sistema funcionar.

A pergunta que deve reger o nível de automatização é “quantos passos são necessários entre obter o código do repositório e o sistema funcionar?”, levando em consideração que isso seja feito em uma máquina apenas com o .Net e o Windows SDK instalados. Isso serve para facilitar o deploy automático em ambientes sem ter que instalar todas as ferramentas de desenvolvimento no servidor.

Preparar o ambiente geralmente envolve tarefas como: rodar scripts de banco, registrar aplicação no servidor de aplicação, criação de usuários, inserção de dados de teste, compilação do programa, entre outras coisas. Todas essas tarefas devem poder ser executadas da forma mais fácil possível. Um bom exercício é imaginar que se trata do desenvolvimento de uma ferramenta open-source, onde a facilidade de subir um ambiente de desenvolvimento é crucial para a evolução do projeto.

Tudo isso deve fazer parte do seu processo de build, e será crucial no momento de fazer setup de build e deploy automáticos em servidor pois, no processo de automatização, ficam bastante claras quais são as dependências da aplicação.

Suporte a um modelo evolutivo

O grande problema e a grande vantagem de automatizar os processos é perceber o quão mal funcionavam os processos antes. Muitas das práticas habituais em equipes de desenvolvimento são amarradas e não facilitam muito a integração.

Uma das práticas não-integráveis mais comuns é o uso de um banco de dados compartilhado. Apesar de apresentar vantagens em curto prazo para a equipe de desenvolvimento, como não ter que gerenciar scripts, em longo prazo isso se mostra uma péssima prática para manter a evolução da aplicação.

O uso de banco de dados compartilhado é um deles.

Tudo isso será crucial no momento de fazer setup de build e deploy automáticos em servidor, pois no processo de automatização, ficam bastante claras quais são as dependências da aplicação.

[CHECKPOINT]

[Abaixo segue exemplo de sessão Links]

**Links**

How to create a generic List of anonymous types?

<http://kirillosenkov.blogspot.com/2008/01/how-to-create-generic-list-of-anonymous.html>

Exportando DataSets para XML, TXT, Word, Excel, HTML

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=13629>

Página do autor na Embarcadero com inúmeros projetos em Delphi e Turbo Pascal

<http://cc.embarcadero.com/Author/222668>

CLS

<http://msdn.microsoft.com/en-us/library/12a7a7h3(VS.71).aspx>

IL / MSIL

<http://en.wikipedia.org/wiki/Common_Intermediate_Language>

Blogs do autor

[http://guintherpauli.blogspot.com](http://guintherpauli.blogspot.com/)

<http://twitter.com/guintherpauli>

[Quando estiver criando o artigo, indique no BOX Saiba Mais (exemplo abaixo) algum conteúdo relacionado ao seu artigo que esteja no site da DevMedia (tem que ser do site [www.devmedia.com.br](http://www.devmedia.com.br/), caso não seja, esse deve ser no Box Links). É válido qualquer tipo de material (vídeo-aulas, cursos, artigos, mesmo de outros canais, por exemplo, um artigo sobre OO com Delphi poderia ter links saiba mais para vídeos de UML do canal SQL Magazine ou ES Magazine (pois são temas relacionados). Quanto mais links, melhor, não existem limites, pois esses links serão exibidos somente na versão digital da revista. Para incluir os links, basta entrar no site da DevMedia e usar a função de procura, copiar o link e o título e colar no BOX ]

**Saiba Mais**

Artigo - ClubeDelphi 98 - Aplicação ASP.NET multicamadas com Firebird – Guinther Pauli

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=10443>

Artigo – Revista Web Mobile 14 – ASP.NET Profiles – Guinther Pauli

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=7527>

[...]

[O mínimo de links aqui neste Box é 5]

**Observação sobre o processo editorial**

Após recebido, seu artigo passará por uma série de avaliações, incluindo avaliação inicial, edição técnica, edição geral, revisão externa, diagramação, revisão interna e impressão. O prazo estimado de publicação, a partir da entrega do artigo, é de cerca de 45 dias. O artigo pode ser publicado na edição impressa, na digital, ou ambas. Qualquer dúvida sobre o status de publicação do seu artigo, sinta-se a vontade para entrar em contato com o editor guinther.pauli@gmail.com