Integração Contínua

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. Em um projeto grande, o processo de integração do trabalho de diversas pessoas em várias equipes pode levar dias, semanas ou até meses.

Quanto mais tempo um desenvolvedor ou equipe trabalha sem integrar seu trabalho com o de seus colegas, mais difícil será essa integração quando acontecer. Visando diminuir esse problema, tenta-se realizar essa tarefa da maneira mais freqüente possível. Mas nem sempre é uma prática simples.

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 ele nos mais diversos contextos. Isso acontece porque a prática deste princípio 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, iterações curtas, etc. 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 aplicar integração contínua se revela muito custoso no processo de desenvolvimento e que, para ter sucesso, 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. Ela deve ser alcançada através da implantação de outras práticas. Várias delas 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 armazena. 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. Idealmente, este ciclo deve se repetir várias vezes ao dia.

Parece um ciclo simples, à 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

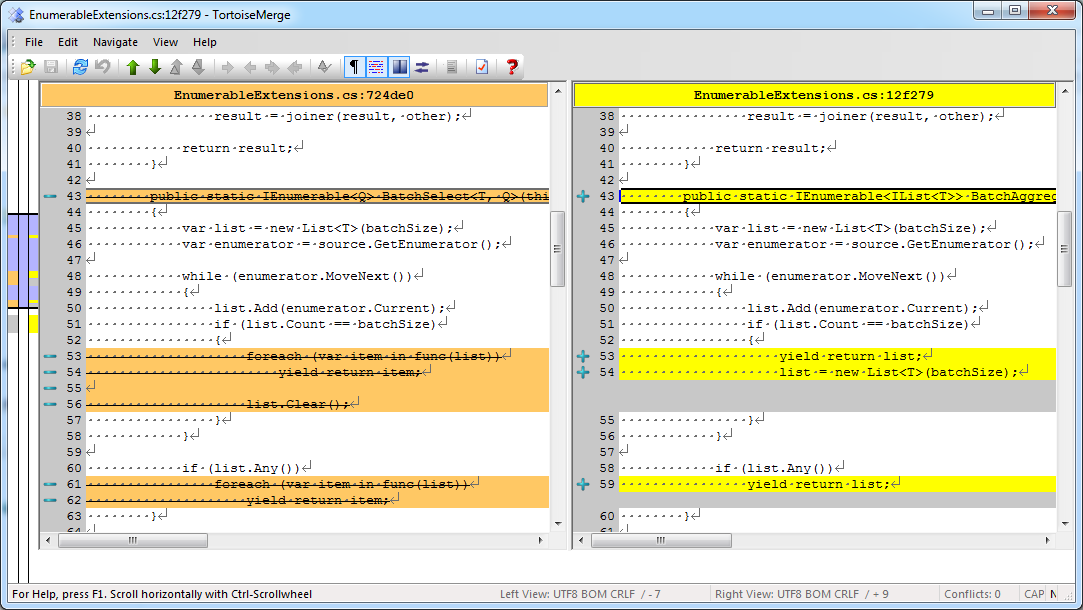
Por mais que pareça óbvio para a maioria dos leitores, utilizar um software de controle de versão é uma decisão essencial para o ciclo de vida do desenvolvimento de software. Ele não só irá controlar o histórico de desenvolvimento como também irá ajudar a integrar as modificações mais facilmente.

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. Cada um tem uma forma diferente de lidar com trabalho descnectado, histórico de versões e tolerância a conflitos.

Isso evidencia o primeiro problema ser resolvido em um projeto: 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 comum entre usuários de controles de versão 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 é: o que versionar? Existe muita discussão nesse ponto. A regra é: tudo o que for necessário para transformar o 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 bastante complexo, apenas para começar a trabalhar.

Durante o desenvolvimento, nem sempre o projeto segue uma história linear. Em certos momentos, a mainline será transformada em uma versão que irá para homologação e/ou produção. Esses pontos são importantes na história do projeto, pois são altamente passíveis de receber feedback. Isso significa que certas vezes poderá haver modificações no código que está em um ponto no passado da mainline. Essas modificações criam um branch na mainline.

Por isso, a terceira pergunta a se fazer é: como gerenciar os branches? Existem várias opções. Certa linha de pensamento defende que toda feature deve ser desenvolvida em branches e a mainline deve estar imaculada. Outra defende que a mainline é o ponto de integração e todos devem integrar sobre ela com a maior freqüência possível. Não existe uma única resposta. Decidir o esquema de branching de um projeto tem muito mais a ver com o ciclo de vida de entregas, do que escolher qual esquema é universalmente melhor.

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 empacotar e até 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 desanimar 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, como o NAnt (que é baseado no Ant, do Java) ou 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.** Exemplo de 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>

<BuildProperties>

Configuration=$(Configuration);Platform=$(Platform);OutputPath=$(BuildDir)

</BuildProperties>

</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="$(BuildProperties)"/>

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

</Target>

<Target Name="Build">

<MSBuild Projects="@(SolutionFile)" Properties="$(BuildProperties)"/>

</Target>

<Target Name="Rebuild">

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

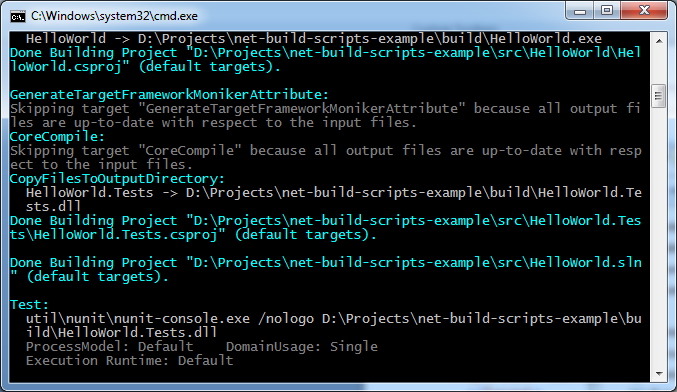
</Target>

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

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

</Target>

</Project>



**Figura 2.** MsBuild executando o script

Testes Automatizados

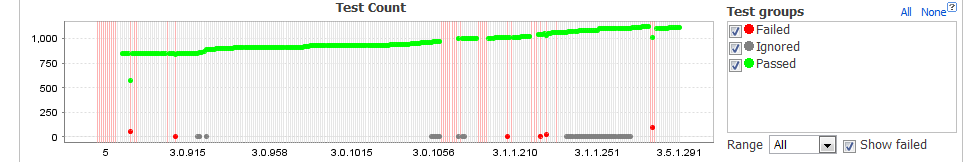
Apesar de ajudar, o build por si só não é capaz de garantir a integridade do software. Somente é possível verificar erros em um software exercitando-o. Essa é a função dos testes automatizados. Os testes ajudam a descobrir muitos bugs no código. Apesar de não garantirem que o código não possui erros, são essenciais para verificar se uma mudança uma parte do software acabará afetando outras que dependem dela indiretamente.

É importante ressaltar que isso não necessariamente significa usar TDD, apesar de ser recomendado.

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.

A rotina de build deve executar os testes automatizados. 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 executá-los. A **Figura 3** mostra um exemplo de gráfico 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 mínimo de pré-requisitos possíveis. Lembre-se que deve ser possível fazer deploy automático em ambientes sem ter que instalar neles todas as ferramentas de desenvolvimento.

Preparar o ambiente geralmente envolve tarefas como: executar 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 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 precariamente 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, por exemplo, 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 grande problema, muitas vezes, é tratar o deployment como um processo sagrado, que acontecerá uma vez somente, e até lá, não se deve ter muita preocupação com a evolução do sistema. Mas quando estamos falando de integração contínua, esse processo deve acontecer muito mais freqüentemente, e o sistema deve estar preparado para isso. Não deve ser necessário recriar o banco a cada instalação do sistema, por exemplo.

É importante deixar claro que a gerência da estrutura do banco de dados é apenas um exemplo. Pode abranger muitas outras coisas, como a configuração de uma ferramenta externa, que precisa poder ser tratada como um mecanismo versionável.

Migrations

Ainda sobre a manutenção de um modelo evolutivo, é importante falar também sobre migrations, que é um tipo de ferramenta para lidar com schema do banco de dados de uma forma bem mais elegante. É um mecanismo que ficou famoso com o framework Ruby on Rails, e se mostrou uma excelente forma de gerenciar a complexidade dos scripts de DDL.

Uma migration consiste num script, que roda transacionalmente, cuja execução é gerenciada pela ferramenta ferramenta. Por exemplo, suponha a existência de dois scripts, A e B. Em uma certa instância do banco de dados, apenas o A foi executado. A ferramenta deve saber que somente o B deve ser executado na próxima vez que for solicitada. Além disso, se a execução de um script falhar, todas as mudanças feitas por ele até então devem ser desfeitas. Isso permite que cada versão da mainline tenha um conjunto de scripts que a fazem funcionar, que descrevem como o schema do banco era quando a integração daquela versão foi feita.

As migrations podem ser implementadas em forma de DSL, que é um dos grandes atrativos que fizeram as migrations do Ruby on Rails ficarem tão famosas. Veja, na **Listagem 2** um exemplo de migration do Rails.

**Listagem 2.** Exemplo de migration usando ActiveRecord::Migration

class CreateProducts < ActiveRecord::Migration

  def self.up

    create\_table :products do |t|

      t.string :name

      t.text :description

      t.timestamps

    end

  end

  def self.down

    drop\_table :products

  end

end

Muitos desenvolvedores usam as migrations do Rails em projetos .Net, visto que sua infraestrutura é independente de tecnologia. Entretanto, existem outras ferramentas análogas no mundo .Net, como as Subsonic Migrations, o Migrator.Net, entre outros. Na verdade, nada impede que scripts SQL puros sejam usados como migrations, desde que seja possível automatizar sua execução.

Sandboxing

Quando se permite que o modelo evolua naturalmente, o mecanismo de instanciação do ambiente torna-se muito mais intuitivo. E sendo ele incluído no processo de build automático, o ambiente de desenvolvimento torna-se muito naturalmente transportável.

O objetivo é que cada desenvolvedor tenha uma cópia completa do ambiente de desenvolvimento, sem depender de qualquer infra-estrutura. Um ambiente confinado para testes e desenvolvimento de funcionalidades. Uma caixa de areia. Preferencialmente, isso deve poder ser feito em passos simples, como executando um script em que, dada certa versão da mainline, instancie todo o ambiente, com banco de dados e serviços auxiliares, sem maiores preocupações.

Isso às vezes é difícil quando se trata de sistemas com alto nível de integração com outros sistemas ou ferramentas. E que nem sempre é possível ter uma cópia desses artefatos rodando na máquina local. Para esses casos, é importante desenvolver mecanismos de stub, para permitir o sandboxing. Mas sempre com a opção de se conectar com o serviço externo, quando necessário.

Mais uma vez é importante fazer a analogia com um projeto open-source. Deve-se perguntar: “qual seria a dificuldade de colocar uma versão do sistema no ar, mesmo executando em uma máquina completamente desconectada?”, pois será exatamente essa a dificuldade de instanciar o ambiente em um servidor de build futuramente.

Servidor de integração

Com todas estas práticas, o desenvolvedor é capaz de integrar e testar cada ponto da mainline para verificar sua integridade. A integração se torna um processo mais simples de ser executado e pode ser executado com mais freqüência. Mas qual é o próximo passo? Por que não fazer com que a integração aconteça automaticamente?

Uma das principais vantagens de todas as práticas citadas anteriormente é a possibilidade de, a partir de sua implantação, executar todos os passos da integração automaticamente em um servidor dedicado a isso. Esse servidor agiria como sentinela, sempre analisando a mainline para verificar sua integridade. Ele deve ser capaz de executar todos os processos necessários para garantir que o produto estaria pronto para ser entregue, se necessário fosse.

Haverá situações onde a mainline estará corrompida, seja porque os fontes não compilam, seja porque os testes não passam. O servidor de integração deve ser capaz de verificar isso e ainda avisar pessoas responsáveis sobre o incidente.

Existem muitas ferramentas que ajudam a alcançar esse objetivo, mas antes delas, é preciso analisar o que isto realmente significa. Qual deve ser o fluxo para integrar um projeto transformando-o em um produto potencialmente entregável? Quais tipos de teste devem ser executados? Qual o ambiente que esse servidor precisa comportar?

É preciso ter em mente que integração contínua é uma prática que reforça a comunicação, pois permite que todos os envolvidos, a qualquer momento, possam ter informações precisas e automáticas sober o progresso de um projeto. É preciso, então, projetar o fluxo de integração de uma forma que o feedback sobre alguma eventualidade possa ser recebido o mais rápido possível. Isto é, num projeto muito grande, a integração, mesmo automática, pode demorar muito tempo. E isso diminuir a fluidez da informação. Muitas vezes é apenas um teste unitário que está quebrado, mas por causa de um ciclo complicado que, que pode envolver instanciação de ambiente complexo, essa informação demora a chegar a quem precisa.

Por esse motivo é importante planejar sua integração. Talvez quebrá-la em pedaços menores e executá-la em servidores diferentes. É comum ter um servidor apenas para compilar e rodar os testes unitários (que são mais rápidos) e outro servidor para rodar os testes de integração e aceitação, que demoram mais. Assim, espera-se que os erros mais triviais sejam percebidos mais rapidamente.

Neste ponto entram os servidores de integração. Eles irão se responsabilizar pela integração automática do código na mainline. Entretanto, depois de toda a reformulação apresentada aqui, essa automatização já está toda pronta, e o servidor de integração irá servir mais como uma ferramenta de comunicação e relatório do que de orquestração, de fato. Quem irá orquestrar o processo de integração, na verdade, é o build automatizado.

A variedade de ferramentas para esse propósito é grande. Só o artigo da Wikipédia sobre o assunto cita quase 30. Desse universo, pode-se dizer que três delas se destacam no mundo .Net: o TeamCity, o Hudson e o CruiseControl.Net.

|  |  |  |
| --- | --- | --- |
| **Ferramenta** | **Empresa** | **Licença** |
| TeamCity | JetBrains | Gratis até 20 configurações |
| Hudson | Oracle | Open-source |
| CruiseControl.Net | Thoughtworks | Open-source |

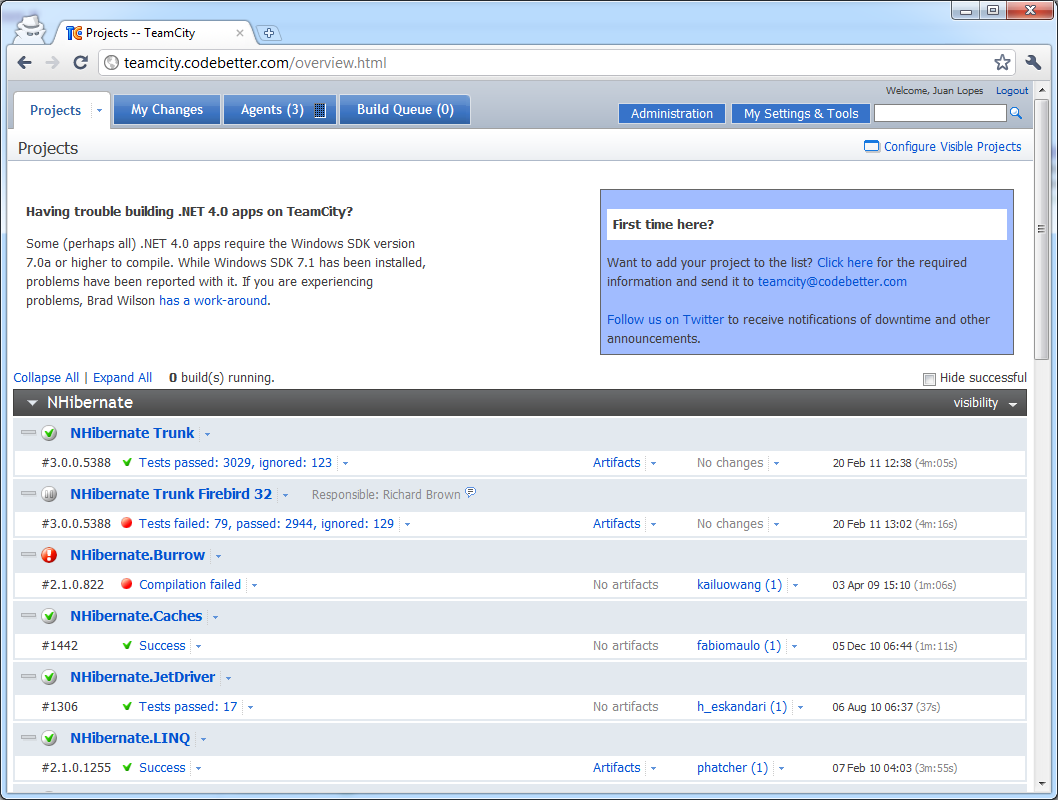
**Tabela 1.** Ferramentas de integração contínua.

Servidor de Integração – TeamCity

TeamCity é uma das ferramentas mais intuitivas nesse ramo. Toda a sua configuração é feita via interface e o balanceamento de carga é extremamente simples, baseado em build agents. Ela foi desenvolvida em Java, mas isso não impede que se integre com o MsBuild e o NUnit de forma impecável. Sua instalação é bastante intuitiva e deixa o sistema já preparado para rodar builds em .Net.

Seu maior problema talvez seja a licença. É grátis até 20 configurações de build e 3 build agents. A partir daí é necessária uma licença Enterprise, que custa US$1999, e inclua-se US$ 299 por build agent adicional.

A JetBrains cedeu licenças para a CodeBetter colocar no ar um servidor aberto para projetos open-source. Muitos projetos grandes são colocados lá. Qualquer pessoa com um projeto open-source pode pedir para seu projeto ser incluído. Na seção de links é possível encontrar o endereço para a página com instruções. A **Figura X** mostra as configurações de build do NHibernate com dois projetos falhando.



**Figura X.** NHibernate sendo compilado no servidor da CodeBetter

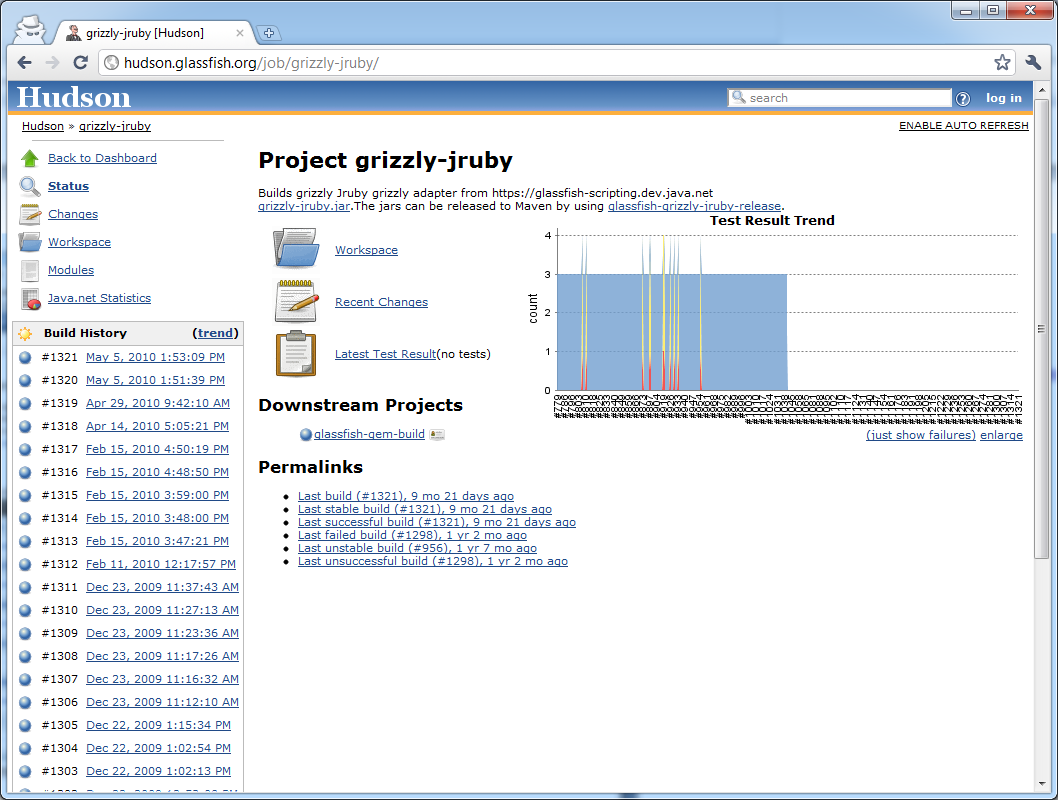
Servidor de Integração – Hudson

O Hudson é um projeto open-source, sustentado pela Oracle atualmente. Sua configuração é também bastante simples. Ele pode ser executado tanto dentro de um container J2EE, como o Tomcat ou o Glassfish quanto standalone. A segunda opção geralmente é a mais comum, para instalação rápida.

Sua configuração é quase tão intuitiva quanto a do TeamCity. Existem alguns pontos onde ele poderia ser melhor, como o esquema de plugins, onde a maior parte dos plugins necessários para compilar projetos em .Net precisam ser instalados antes de começar a usar, diferentemente do TeamCity. Em compensação, o sistema de builds distribuídos via máster/slave é talvez o mais simples entre os três produtos.

A interface gráfica do Hudson é um pouco menos menos polida que a do TeamCity, como pode ser percebido na **Figura X**.

No final do ano passado, a comunidade entrou em uma disputa com a Oracle com relação ao controle do projeto e aos direitos sobre o nome Hudson. Desde então, houve uma dissidência do projeto, agora chamado Jenkins. Agora, Hudson (ainda com a Oracle) e Jenkins são dois projetos independentes, que representam o mesmo software, desenvolvido por frentes diferentes.



**Figura X.** Build de um dos subprojetos do GlassFish

Servidor de Integração – CruiseControl.Net

Dos três produtos apresentados, o CruiseControl.Net é talvez o menos atrativo. Sua interface é bruta, seus controles confusos. Sua configuração caótica. Mesmo assim, talvez por ter sido escrito em .Net, é uma das ferramentas mais populares para integração contínua no mundo Microsoft.

Apesar disso, uma das suas grandes vantagens é seu formato de feed e configuração XML ser compatível com o CruiseControl para Java, uma das ferramentas mais utilizadas ainda hoje. Isso torna o CruiseControl.Net compatível com a maior parte das ferramentas já desenvolvidas para sua versão em Java.

Seu grande ponto fraco é a configuração em XML, que demora um grande tempo para ser entendida e dominada. Mas uma vantagem desse aspecto é que as configurações podem ser facilmente versionadas.

Fazendo a informação irradiar

[CHECKPOINT]

<http://codebetter.com/jameskovacs/2009/02/24/announcing-teamcity-codebetter-com/>