4
Integração

Quando citamos trabalho em equipe, normalmente estamos falando de tarefas que são divididas entre pessoas. "Porém, trabalho em equipe não é um problema de divisão e conquista apenas. É um problema de divisão, conquista e integração" (Beck & Andres, 2004). Em XP, várias técnicas são utilizadas para facilitar e melhorar a integração do trabalho feito por cada indivíduo. Dessa forma, o resultado obtido são sistemas que parecem ter sido feitos por uma mesma pessoa.

4.1.Continuous Integration

"*Continuous Integration* é uma prática de desenvolvimento de software em que os membros de uma equipe integram o seu trabalho freqüentemente. Normalmente, cada pessoa integra pelo menos uma vez por dia, levando a múltiplas integrações em um único dia. Cada integração é verificada por um *build* automatizado (incluindo a execução dos testes). Dessa forma, os erros de integração podem ser detectados o mais rápido possível. Muitas equipes perceberam que essa abordagem leva a uma redução significativa nos problemas de integração. Além disso, permite que a equipe desenvolva sistemas coesos mais rapidamente" – Martin Fowler[22].

As maiores vantagens de fazer integração contínua são: os problemas são detectados e corrigidos continuamente; as advertências sobre código incompatível ou quebrado aparecem logo; os testes de unidade são executados para todas as alterações; há disponibilidade permanente da versão corrente para teste, demonstração e distribuição. Além disso, é possível obter informações sobre a condição do código produzido utilizando métricas e verificações oferecidas por algumas ferramentas de *build* automatizado.

[22] http://www.martinfowler.com/articles/continuousIntegration.html

Em *eXtreme Programming*, aconselha-se que a integração e o teste das mudanças sejam feitos com freqüência. Não deve demorar mais do que algumas horas entre uma integração e outra. O passo de integração é imprevisível, mas pode facilmente levar mais tempo do que a própria programação em si. Por isso, quanto mais tempo a integração demorar para ser feita, mais ela será complicada e menos previsíveis serão as dificuldades (Beck & Andres, 2004). "Uma boa equipe XP integra e testa o sistema inteiro muitas vezes por dia" (Jeffries et al., 2001).

Esta prática diminui o risco de problemas inesperados – que normalmente causam atrasos no cronograma – porque garante que não existe um grande número de mudanças para serem integradas. Esta também mantém a taxa de defeitos baixa porque, como parte da integração, os programadores devem executar os testes para garantir que suas alterações não quebraram nada (McBreen, 2002).

Existem dois estilos de integração contínua: síncrona e assíncrona. A integração síncrona é feita por um programador depois de algumas horas de desenvolvimento. Para isso, ele obtém as últimas atualizações do repositório de código compartilhado e executa um *build* completo do sistema. No estilo assíncrono, uma ferramenta específica verifica as últimas alterações e faz um *build* completo do sistema. Toda noite, o sistema é compilado e os testes de fumaça[23] são executados (Wake, 2001). Se algum erro for encontrado, a ferramenta avisa os desenvolvedores por e-mail ou por meio de publicação em algum site.

A prática de integração contínua permite que sejam feitos *deployments* diários e incrementais das aplicações. Isso possibilita uma integração maior entre os desenvolvedores e os clientes, pois estes últimos podem verificar o andamento do projeto progressivamente, ao invés de esperar por um sistema pronto para fazer críticas.

[23] Um conjunto de testes que são executados para verificar as funcionalidades básicas de um *build*. Testes de fumaça (do inglês *smoke tests*) são os primeiros testes a serem executados após cada *build*. Idealmente, estes são automatizados e podem ser executados de forma rápida e fácil. Em projetos ágeis, testes de fumaça são completamente automatizados e tipicamente incluem todos os testes de unidade. A origem do termo veio do design de hardware, onde o primeiro teste em um novo pedaço de hardware montado era o "ligue e verifique se não está saindo nenhuma fumaça" (Meszaros, 2007).

Assim como as outras práticas, é possível perceber uma interdependência da integração contínua com o restante. Só é possível fazer *Continuous Integration* em XP por causa da união do grupo, porque este é suportado por testes e porque XP fornece um design de código simples via *Refactoring* (Wake, 2001).

4.2.Coletividade do Código

Para que exista integração entre os desenvolvedores, é preciso que todo o código produzido seja compartilhado por toda a equipe. Dessa forma, todos são responsáveis pelo código e qualquer um pode fazer uma correção em alguma parte do sistema. Qualquer pessoa da equipe pode melhorar uma parte do sistema sempre que precisar. Se alguma coisa está errada com o sistema e a correção não está fora do escopo daquilo que deve ser feito, deve-se seguir em frente e corrigir o erro (Beck & Andres, 2004).

Continuous Integration é outro pré-requisito importante para que a posse do código seja compartilhada. Se a equipe está fazendo muitas mudanças, ela pode querer reduzir o intervalo entre as integrações para manter o custo de integração baixo (Beck & Andres, 2004).

Pode-se categorizar basicamente a posse do código de duas formas: posse individual e posse coletiva. Cada uma delas tem suas vantagens e desvantagens. A posse individual envolve a especialização em algumas áreas ou tecnologias específicas. Também envolve um apego emocional por parte de quem escreveu o código por aquilo que fez. A posse coletiva, por outro lado, envolve muitas pessoas trabalhando em várias áreas, sem que existam especialistas. Todo mundo sabe um pouco de tudo e, por isso, todos estão aptos a alterar qualquer parte do código. Essa prática envolve uma estima menor por aquilo que cada um faz, tornando mais difícil algum programador se sentir ofendido porque um outro alterou o seu algoritmo precioso para funcionar de maneira melhor, mais clara ou mais rápida. Em um projeto XP, no entanto, é imprescindível a posse coletiva, ou as outras práticas de XP não terão chances de funcionar (Stephens & Rosenberg, 2003).

A posse compartilhada do código ajuda a manter a simplicidade de design permitindo que qualquer um veja uma violação das regras de design e corrija o problema. Isso reduz o risco do cronograma atrasar já que um programador nunca

tem que esperar que outra pessoa faça as alterações em alguma outra classe (McBreen, 2002).

"Com a posse compartilhada do código, toda a equipe é proprietária de todo o código. Qualquer pessoa pode alterar qualquer parte que esteja precisando" (Jeffries et al., 2001).

4.3.Padrões de Desenvolvimento

Como é possível pensar em integração no meio de uma confusão? Como é possível pensar que o código é de todos se basta olhar para a organização do código e descobrir quem o escreveu? É muito fácil entender aquilo que é feito por nós mesmos. "Eu posso sempre ler meu próprio código. Mas espere, todo o código é meu código também" (Jeffries et al., 2001). Partindo dessa idéia, é preciso seguir alguns padrões para realmente agregar a equipe. Por isso é aconselhável definir padrões para formatação de código, para estruturação de projetos e criação de versões. A idéia é que não se saiba quem é o autor de um código-fonte, já que a forma como todo o código é escrito é muito parecida.

Padrões de código são importantes em qualquer projeto de programação. Em projetos XP são ainda mais importantes, pois qualquer programador pode alterar qualquer parte do código a qualquer momento (Stephens & Rosenberg, 2003). Estes também ajudam muito a manter o código limpo, bem como mais fácil de ser lido e decifrado. Encorajar programadores a dar para suas variáveis e métodos nomes com um significado ajuda a manter o código auto-documentado (Stephens & Rosenberg, 2003).

4.4.Ferramentas

Muitas das tarefas de integração podem ser automatizadas por meio de ferramentas. A integração continua pode ser feita usando Cruise Control ou Continuum. O compartilhamento de código e o controle de versões podem ser feitos com o CVS ou Subversion. Os padrões de formatação de código e de estilo podem ser garantidos com ferramentas como o Checkstyle e o Jalopy.

4.4.1.Cruise Control

Para garantir que a integração seja feita com a freqüência adequada, deve-se empregar uma ferramenta para integração contínua. O Cruise Control é uma ferramenta para *build* automatizado que, valendo-se dos *buildfiles* do Ant e do sistema de controle de versão, garante que os projetos estão sendo continuamente integrados. O Cruise Control é escrito em Java e possui os mesmos benefícios de independência de plataforma oferecidos por outras ferramentas como Ant e o JUnit (Hightower et al., 2004).

O Cruise Control é baseado em um conceito simples. Uma instância desta aplicação é configurada para observar um repositório de controle de versões e detectar mudanças nos arquivos que lá estão. Quando alguma alteração é percebida, a instância atualiza uma cópia local do projeto com as modificações e executa o roteiro de *build* para o projeto. Depois que o *build* é feito (com sucesso ou não), o Cruise Control publica vários artefatos especificados pelo usuário (incluindo um *log* do *build* que foi feito) e informa os membros do projeto sobre o sucesso ou a falha na construção.

4.4.2.Continuum

O Continuum é o servidor de *build* e integração contínua para o Maven (Massol & Van Zyl, 2006). Este funciona da mesma forma que o Cruise Control, mas se adequa trivialmente a projetos que são gerenciados pelo Maven.

4.4.3.CVS

O Sistema de Versões Concorrente (do inglês *Concurrent Versions System*, CVS) implementa um sistema de controle de versões. Ele observa todo o trabalho e todas as alterações em um conjunto de arquivos, normalmente a implementação de um projeto de software, e permite que vários desenvolvedores colaborem entre si (de forma totalmente separada). O CVS se tornou popular nos projetos livres e *open source*.

4.4.4.Subversion

Subversion é uma aplicação *open source* para controle de versões. Também conhecido como SVN, o Subversion foi feito especificamente para ser um substituto moderno para o CVS. Isso significa que o Subversion trata problemas que não são resolvidos pelo CVS e possui conceitos mais amadurecidos de controle de versões de arquivos e projetos. Por esse motivo, o uso do SVN é recomendado ao invés do CVS.

4.4.5.Checkstyle

Checkstyle é uma ferramenta feita para ajudar programadores que escrevem código em Java e que aderiram a um único padrão de codificação. Esta automatiza o processo de checagem do código Java, poupando o trabalho manual para realizar essa tarefa chata, porém muito importante. Isso torna o Checkstyle uma ferramenta ideal para equipes que desejam garantir um padrão de codificação.

Esta ferramenta é altamente configurável e pode ser personalizada para suportar qualquer tipo de padrão. Deve-se tomar cuidado, no entanto, com o grau de restrição imposto pela ferramenta. Se muitas restrições forem configuradas enquanto a equipe não está totalmente acostumada com um novo padrão de codificação, muitos erros e alertas serão gerados, tornando o relatório gerado pela ferramenta pouco útil. Uma sugestão é adicionar apenas as condições de erro mais críticas no início e aumentar as restrições com o tempo.

4.4.6.Jalopy

Jalopy é um formatador de código-fonte para a linguagem de programação Java. Este organiza qualquer código Java válido de acordo com regras altamente configuráveis. Dessa forma, conserva-se um estilo de código sem colocar a maçante tarefa de formatação como uma responsabilidade de cada desenvolvedor.

O formatador de código pode ser executado antes de cada *commit* no sistema de controle de versões. Deste modo, todo código que vai para o repositório estará seguindo a formatação padrão definida.

4.5.Precauções

Quando falamos em trabalho em equipe, integração torna-se uma parte fundamental para que o resultado final que está sendo construído seja satisfatório. Por isso, deve-se tomar cuidado com algumas situações peculiares da integração.

Às vezes, o trabalho que é feito individualmente funciona perfeitamente. O *build* de todo o sistema é feito corretamente, os testes são executados com sucesso e os devidos artefatos são produzidos. Porém, assim que a ferramenta de *Continuous Integration* obtém as alterações e realiza o *build*, deparamo-nos com um erro. Isto pode ocorrer por causa de um *commit* incompleto, por uma falha de configuração no servidor de integração ou por uma característica do ambiente de desenvolvimento que é diferente no servidor e na estação de trabalho. Esse tipo de problema vai acontecer freqüentemente e, para torná-lo menos incômodo, é conveniente possuir uma única pessoa responsável por verificar os erros de integração. O papel dessa pessoa é verificar qual tipo de erro está acontecendo: se for um erro de configuração do servidor, ela deve descobrir como corrigi-lo; se for um erro de programação, ela deve repassar a responsabilidade da correção para o programador que fez a alteração.

Apesar de o conceito de C*ontinuous Integration* poder ser quase completamente automatizado, uma parte fundamental deve ser feita manualmente por cada desenvolvedor – o *commit*. O envio de alterações para o repositório de controle de versões é fundamental para que a integração seja feita. Se os desenvolvedores só enviarem suas atualizações uma vez por semana, por exemplo, pouco importa se a ferramenta de integração está configurada para efetuar *builds* a cada hora ou uma vez por dia.

O outro extremo, por outro lado, também pode ser problemático. Levar o conceito de continuidade ao pé da letra pode ser nocivo para um projeto. Configurar o servidor de integração para integrar a cada hora pode gerar erros constantemente. Isso atrapalha o processo de desenvolvimento de software, pois interfere no trabalho do desenvolvedor que fez as alterações e também dificulta o trabalho dos outros, que podem começar a ter problemas por causa das alterações alheias realizadas de maneira incorreta e com muita freqüência. Configurar a integração para que seja feita automaticamente uma ou duas vezes por dia, dependendo da velocidade de produção dos programadores, é mais do que

suficiente para a maioria dos projetos. Caso mais de duas integrações sejam necessárias em determinado momento, estas devem ser iniciadas manualmente.

A integração contínua pode ser configurada para produzir *deployments* diários. *Deployment* diário é uma prática complementar de XP porque possui muitos pré-requisitos: a taxa de defeitos deve ser muito pequena; o ambiente de *build* deve estar definitivamente automatizado; as ferramentas de *deployment* também devem estar automatizadas, incluindo a capacidade de voltar atrás e eliminar os casos de falha; e, mais importante, a confiança na equipe e com os usuários precisa estar altamente desenvolvida (Beck & Andres, 2004). O *deployment* diário não deve ser feito se:

- ***Não conhecer o cliente***: apresentar versões diferentes todos os dias torna-se um problema caso o cliente não entenda como funciona o mecanismo de desenvolvimento incremental. A cada dia novas funcionalidades são apresentadas em detrimento de outras que não são tão importantes naquele momento e que por isso não estão presentes na versão disponibilizada. Se o cliente insistir em reclamar das partes do sistema que ainda não estão prontas, ao invés de atentar para o que é realmente importante, é melhor não fazer *deployment* diário. Ou faça, mas não deixe que o cliente saiba disso. Essa situação provoca estresse e atrapalha a produtividade de qualquer equipe de desenvolvimento, ainda mais uma equipe pequena.
- ***Não possuir um ambiente de desenvolvimento e deployment automatizado***: construir um sistema e colocá-lo em produção de forma manual não é uma tarefa trivial. Fazer isso todos os dias pode significar não fazer mais nada. A automação do *build* e do *deployment* é imprescindível para que essa prática seja utilizada.
- ***Apresentar muitos erros ou dificuldade na integração***: Se está difícil integrar uma vez por dia, não será mais fácil integrar e fazer o *deployment* com a mesma freqüência.

Assim como a integração, a tarefa de organização do código de acordo com um padrão pode ser automatizada, mas não totalmente. Uma ferramenta pode garantir a ordenação dos métodos e a forma como as chaves devem ser abertas e fechadas, mas não pode garantir que o nome das classes e seus métodos tem um significado claro. Para garantir que uma nomenclatura adequada está sendo utilizada, é preciso fazer revisões do código-fonte. *Pair Programming* é uma

prática de XP que ajuda a manter o código mais limpo, tendo em vista que este é revisado enquanto está sendo escrito (ver capítulo 5).

Algumas ferramentas permitem que um formatador seja vinculado ao processo de sincronização com o repositório de versões. Assim, todo código que vai para o repositório obrigatoriamente seguirá o padrão especificado, livrando os programadores de mais uma obrigação. Além disso, este tipo de configuração facilita a conversão de código legado para o padrão definido. Sempre que algum programa que foi escrito anteriormente precisar de uma correção, automaticamente ele será convertido para o novo estilo.

Apesar de resolver quase todo o problema, as ferramentas de formatação de código não conseguem garantir 100% de fidelidade com o padrão definido. Uma ferramenta de checagem pode ser útil neste caso. Mas se não houver um esforço de cada desenvolvedor para seguir o padrão definido, a quantidade de erros e alertas é tão desanimadora que faz com que estes não sejam corrigidos. Para evitar uma quantidade muito grande de alertas, é aconselhável que ferramentas deste tipo sejam configuradas de maneira pouco restrita no começo.

Também existem alguns potenciais problemas relacionados com a posse compartilhada de código:

- Algumas pessoas possuem orgulho do seu próprio código e não deixam que outros mexam nele. Ou seja, na verdade apenas o código é compartilhado, mas a posse ainda continua sendo individual.
- Corre-se o risco de transformar “todo mundo é responsável” em “ninguém é responsável”. Ou seja, se todos são responsáveis, ninguém tem culpa por algo que não foi feito. Isto também torna difícil a constatação de que existe um programador produzindo muito código com problema.
- Se um padrão de desenvolvimento não for definido e usado por todos, alterar o código alheio pode se tornar uma tarefa árdua. Neste caso, a principal conseqüência é a existência de uma confusão de estilos e abordagens.
- Como todo mundo altera todo o código, as chances de surgirem especialistas em determinadas áreas diminui. Esta conseqüência pode ser vista de forma positiva ou negativa. É importante ter mais de uma pessoa sabendo resolver os problemas relacionados com áreas críticas. Porém,

fazer com que todos os programadores tenham conhecimento até mesmo de áreas menos relevantes pode ser uma perda de tempo.

- Quando o grau de conhecimento dos programadores é muito desigual, um programador muito mais experiente pode ver alterações sendo feitas no código escrito por ele de forma indevida.

Como a posse coletiva do código depende muito do *Pair Programming*, é muito difícil alcançá-la sem que esta outra prática esteja sendo empregada.