Relatório de Seminário
Integração Contínua

Felipe Matos, Hugo Tavares, Welton Abreu, Yan Paiva

[1]Departamento de Ciência da Computação – Universidade Federal de Juiz de Fora (UFJF)
Rua José Lourenço Kelmer, S/n - Martelos, Juiz de Fora - MG, 36036-330

Resumo. *Este relatório faz uma breve introdução à integração contínua e discorre brevemente sobre decisões que foram feitas durante a confecção de um seminário sobre o tema.*

1. Introdução

Como parte do processo avaliativo da disciplina de Engenharia de Software do Departamento de Ciência da Computação da Universidade Federal de Juiz de Fora (UFJF), a turma foi dividida em grupos e cada grupo elaborou um seminário sobre um tema específico. Este grupo, composto por Felipe Matos, Hugo Tavares, Welton Abreu e Yan Paiva, foi responsável pelo seminário de Integração Contínua. No presente relatório, estarão informações correspondentes às decisões tomadas pelo grupo a respeito da escolha dos conteúdos abordados e sobre o conteúdo propriamente dito.

2. A Integração Contínua

A etapa de integração no desenvolvimento de um software, principalmente de um de grande escala, utilizando as metodologias tradicionais é um processo longo, complexo e imprevisível. Diversos erros de interface entre as diversas partes de um software estão fadados a acontecer, já que várias dessas partes são desenvolvidas separadamente, e frequente são feitas por desenvolvedores distintos em momentos distintos.

Mesmo que se implemente uma rotina rigorosa de testes, os problemas certamente continuarão a ocorrer, pois durante o desenvolvimento guiado pelas metodologias tradicionais o software só é integrado após todas as partes serem desenvolvidas. Não há maneiras de prever com exatidão a dificuldade que esse processo terá, já que não se pode supor como o software irá se comportar ao ter suas diversas partes conectadas.

Nesse contexto, surge a integração contínua, como parte da metodologia ágil XP, sendo uma das suas 12 práticas originais. A técnica da integração contínua se baseia no princípio de efetuar a integração do sistema durante todas as etapas do desenvolvimento. O time deve integrar ao menos uma vez por dia, fazendo commit em um repositório, que por conta disso se mantém sempre atualizado e devidamente integrado.

Como a técnica se baseia simplesmente nesse princípio, não são necessárias ferramentas complexas; são necessárias apenas uma ferramenta de gerência de configuração e ferramentas para conduzir os testes automatizados, que são importantíssimos para a integração contínua. Também há a possibilidade de se utilizar um servidor de integração contínua, tal como Jenkins, Cruise, etc. mas não é requisito básico para a aplicação da técnica, embora seja de grande utilidade em equipes distribuídas, já que é mais fácil do que ter um servidor físico de integração que seria acessado remotamente pela equipe. No caso de equipes locais, é possível fazer a integração em uma máquina local específica para esse uso de maneira ainda mais simples.

3. Requisitos

Ferramentas de gerência de configuração como Git e Bitbucket são recomendadas para qualquer processo de software, pois permitem a elaboração e manutenção de um ambiente propício para o desenvolvimento. Ao utilizar essas ferramentas, o código pode se tornar disponível em qualquer lugar que o desenvolvedor esteja; o trabalho em equipe é facilitado; é possível se ter um histórico das alterações no código e é eliminada a necessidade de se contatar o servidor para efetuar operações básicas como commit, entre diversos outros benefícios. Essas ferramentas ainda permitem aos desenvolvedores a definição de branches, que são como sub repositórios alternativos, ainda relacionados ao trunk (repositório principal) mas que são mantidos separadamente.

A gerência de configuração é a base da integração contínua. Além de se utilizar uma ferramenta de gerência de configuração, é imprescindível a adoção de algumas práticas quanto à sua utilização. É necessário que o projeto esteja completamente disponível no repositório, de preferência incluindo as ferramentas e arquivos que os desenvolvedores utilizam, tais como arquivos de configuração de IDE, etc. Isso é necessário para que seja possível sincronizar com o repositório de uma máquina nunca antes utilizada para o desenvolvimento do projeto e efetuar o processo de build (testes automatizados, compilação e todas as outras operações necessárias para gerar um executável) com uma única linha de código (build automatizada).

Além disso, não se deve utilizar branches, pois ao estabelecer branches, parte do código passa a ser mantida separadamente do trunk (onde o desenvolvimento principal ocorre), o que por consequência torna necessário fazer a integração de tal código posteriormente, se constituindo, assim, uma integração não contínua.

Para que o código seja ”auto-testável”, que é um requisito importantíssimo para a integração contínua, é necessário que haja uma suíte de testes automatizados que devem possuir uma boa cobertura do código (testar grande parte do código), tornando assim possível verificar com certo nível de precisão se o código desenvolvido a ser integrado no trunk possui ou não determinados bugs. Para isso, são utilizadas ferramentas de teste unitário, como as XUnit por exemplo (JUnit, PHPUnit, unittest, etc) e possivelmente outras, como Selenium, etc. que podem depender do domínio da aplicação a ser desenvolvida e dos requisitos e ferramentas utilizadas no desenvolvimento. Além disso, deve ser possível iniciar os testes a partir de um comando simples, para agilizar e facilitar o processo de build e de commit. O resultado da bateria de testes deve indicar se algum dos testes falhou, e se isso ocorrer, essa falha deve impedir o processo de build.

Para que os testes sejam mais efetivos, deve-se fazer com que o ambiente de teste seja o mais próximo possível do ambiente de produção. Para isso, deve-se utilizar o mesmo software de banco de dados em sua mesma versão, utilizar a mesma versão do sistema operacional, mesmas bibliotecas, mesmos IPs e portas, o mesmo hardware, etc. Uma boa abordagem é efetuar o teste em máquinas virtuais para facilitar esse processo de ”copiar”o ambiente de produção.

É importante ressaltar que tais testes, por mais que com boa cobertura de código não garantem com perfeição que o código não contém nenhum bug, mas ao menos possibilitam a garantia de que certos bugs determinados não estão presentes, o que já de extrema valia.

4. Teste de Software

Pela importância que os testes têm na integração contínua, optamos por, durante a apresentação do seminário, discorrer mais detalhadamente sobre o que é o teste de software. O teste de software pode ser definido como a ação de executar um código com a intenção de encontrar erros [MYERS 2004]. É um dos três pilares da qualidade de software (verificação, validação e teste) e se difere da verificação por envolver a execução do código, e da validação pelo fato desta ser referente à aceitação do cliente e não ao funcionamento propriamente dito [NEVES 2015].

4.1. Tipos de Teste

Na abordagem dos testes de software, optou-se por adotar a terminologia padrão da área de teste, definindo os principais tipos de teste como sendo o teste funcional, teste estrutural e teste baseado em defeitos [BARESI 2006].

4.1.1. Teste Funcional

O teste funcional diz respeito ao funcionamento do sistema. Verifica se um requisito específico (que pode ser funcional ou não funcional) foi implementado corretamente. Tal verificação pode ser feita diretamente através de ferramentas que verificam as funções "na prática", ou a partir da elaboração de testes unitários (utilizando ferramentas como as XUnit) para testar suas unidades (métodos ou procedimentos) verificando se tais métodos têm o comportamento esperado.

4.1.2. Teste Estrutural

O teste estrutural é um complemento ao teste funcional. É basicamente a combinação do teste funcional com uma ferramenta que avalie quanto do código é coberto pelos testes já escritos. Normalmente essas ferramentas geram a visualização de um grafo de fluxo de controle (GFC) que oferece uma visualização do código a partir de nós, que representam blocos de código, e arestas, que representam as interações entre os blocos de código. Em seguida, essas ferramentas colorem os nós e as arestas para simbolizar partes do código cobertas ou não pelos testes.

4.1.3. Teste Baseado em Defeitos

O teste baseado em defeitos é, como o nome diz, baseado nos erros já encontrados anteriormente pelos desenvolvedores, tanto na aplicação em questão, quanto em aplicações similares, etc. Também existem técnicas avançadas envolvendo algoritmos genéticos e similares, mas essa variação não costuma ser amplamente aplicada na indústria por conta do custo elevado.

4.2. Fases de Teste

Ainda nas definições correntes de teste de software, as fases de teste são definidas como: teste unitário, teste de integração, teste de sistema, teste de aceitação e teste de regressão [DELAMARO 2016].

4.2.1. Testes Unitários

Testes unitários visam analisar o comportamento de uma unidade (método ou função) e compará-lo com o comportamento esperado. São importantes para atestar a qualidade do código em suas unidades lógicas mínimas, portanto, é o teste mais básico para aferir o funcionamento do código. Como o teste é feito em cada método separadamente, e frequentemente os métodos interagem uns com os outros, são criados drivers, que são responsáveis por fornecer a entrada e saída para os testes, e os stubs, que simulam o funcionamento de outras unidades das quais a unidade sendo testada depende. Para este tipo de teste, são utilizadas ferramentas próprias para testes unitários, específicas para cada linguagem. Uma ”família”importante para testes unitários é a família XUnit (JUnit, PHPUnit, etc.).

Figura 1. Teste unitário

4.2.2. Testes de Integração

Testes de integração testam a interação entre duas ou mais unidades. São extremamente importantes para verificar que duas unidades que interagem entre si o fazem em harmonia, pois diversos erros podem ocorrer durante essa interação, tais como a presença de variáveis não globais sendo tratadas como se fossem, etc.

4.2.3. Testes de Sistema

Testes de sistema são os teste de integração de mais alto nível. Se ocupam da interação entre todo o sistema e frequentemente demoram dias para serem executados. Neles, além dos códigos do software e do funcionamento em geral, também são testadas as interações com o sistema operacional, com o banco de dados, além de a compatibilidade do sistema com o manual do usuário, o treinamento, etc.

4.2.4. Testes de Aceitação

Testes de aceitação se situam na fronteira entre teste e validação (componentes da qualidade de software). Avaliam, mediante testes, se o sistema obtido é correspondente às expectativas do cliente.

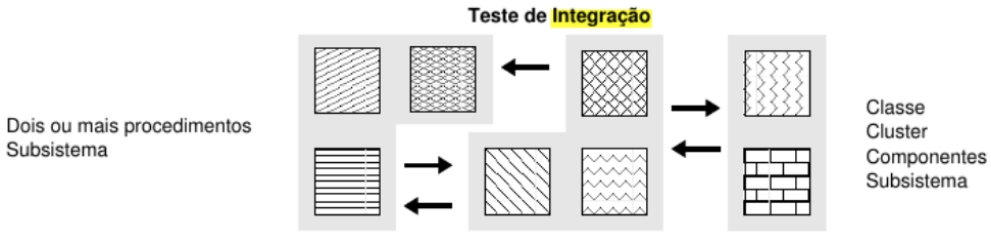

Figura 2. Teste de Integração

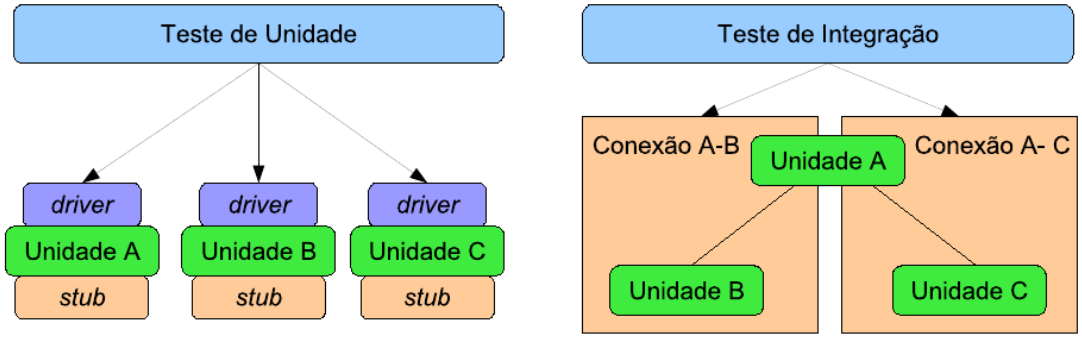

Figura 3. Teste Unitário e Teste de Integração

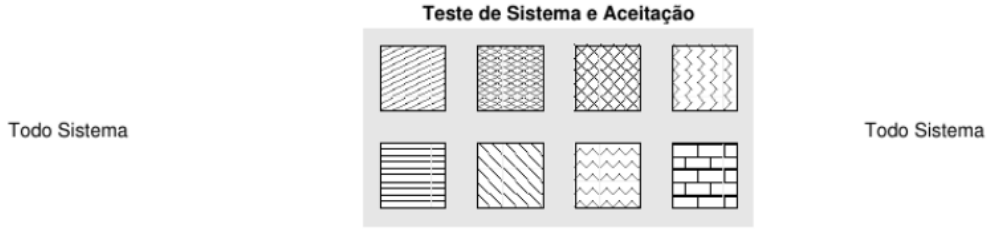

Figura 4. Teste de Sistema e Teste de Aceitação

4.2.5. Testes de Regressão

Testes de regressão são responsáveis por verificar se nenhum efeito colateral foi introduzido no código. No contexto da integração contínua, esse tipo de teste é ainda mais importante, já que cada commit gera uma versão nova do código. Para esses fins, os testes unitários e de integração já descritos acima são também caracterizados como testes de regressão, já que são executados a cada commit, cumprindo esse papel a cada alteração no código.

5. Princípios da Integração Contínua

5.1. Esperar os testes

O desenvolvedor deve esperar o sucesso dos testes para começar uma nova tarefa. Isso é necessário pois se os testes falharem, todo código novo que for gerado terá que ser adaptado posteriormente, já que a base dele estaria com erros. Isso pode gerar muito trabalho e possívelmente a perda de todo código desenvolvido nesse período.

5.2. Não comentar testes

É uma prática comum (embora perigosa) comentar testes que falham em caráter provisório para ir elaborando outras partes do código e deixando para consertar depois. Essa é uma manobra extremamente perigosa e que põe em risco a estabilidade do projeto, já que em um contexto de integração contínua, além dos problemas gerados pela inserção de erros no projeto, ainda se teria os problemas gerados pelo fato de todo o código que produzirem em seguida usar esse código com erro como base.

5.3. Limite de tempo para correção

Pelo motivo mencionado acima, todos os problemas devem ser corrigidos o mais rápido possível, já que todo erro vai gerar mais erros a cada commit.

5.4. Responsabilidade

Os desenvolvedores devem assumir responsabilidade pelas quebras causadas por suas mudanças. Além de ser uma prática ética, fazê-lo agiliza o processo de descoberta do que está causando o erro e de como consertá-lo.

5.5. Desapego e Coragem

Além da refatoração pregada pela XP e outras metodologias ágeis, o desenvolvedor deve estar sempre disposto a modificar seu código. Em uma escala maior, a equipe deve estar sempre disposta a voltar a uma versão anterior caso seja necessário. Muito tempo pode ser perdido tentando encontrar os problemas e resolvê-los; frequentemente é mais prático descartar as alterações feitas e voltar a uma versão estável. Essa mentalidade é extremamente importante pois, apesar de todas as precauções, é bem provável que em algum momento a build do trunk vá quebrar.

6. Práticas

6.1. Comunicação

Integração diz respeito à comunicação. A integração permite aos desenvolvedores se comunicarem sobre as mudanças que eles efetuaram. Comunicação frequente permite aos desenvolvedores saberem rapidamente quando as mudanças são efetuadas.

6.2. Procedimento para o Commit

Para que se possa efetuar o commit para o trunk (repositório principal atualizado), o código deve ser capaz de passar pelo processo de build sem erros, ou seja, deve ser capaz de passar por todos os testes sem que se encontre nenhuma falha, já que um erro em um teste cancela o processo de build. Por conta disso, antes de cada commit, o desenvolvedor deve primeiro fazer a atualização de sua cópia de trabalho do repositório para que ela seja igual ao do trunk, ou seja, a versão mais atual do código. Após isso, o desenvolvedor resolve quaisquer conflitos com o trunk e então efetua o processo de build em sua máquina local. Caso esse passo se dê sem erros, o desenvolvedor pode enfim efetuar seu commit para o trunk.

Ao fazer isso frequentemente, a equipe é capaz de descobrir rapidamente se há algum conflito entre dois desenvolvedores e resolvê-lo rapidamente. Como os desenvolvedores farão commit várias vezes por dia, um conflito é detectado em poucas horas depois

de ocorrerem, e por ter se passado tão pouco tempo, não ocorreram tantas alterações e então torna-se muito mais fácil resolver o conflito. Quando um conflito passa semanas ou meses sem ser sem ser detectado, diversas alterações no sistema como um todo podem ter sido feitas e se torna uma tarefa árdua adaptar todo o sistema desenvolvido posteriormente ao erro para que funcione corretamente após a correção dele. Quanto mais frequentes os commits forem, em menos lugares será necessário procurar para que se encontre a causa dos erros.

Além dos benefícios já citados, o commit frequente faz com que os desenvolvedores fragmentem seu trabalho em blocos de poucas horas. Isso facilita a avaliação do progresso do projeto e traz consigo a sensação de não estar estagnado - a percepção de que o projeto está avançando é imediata, não são necessários vários dias para que se possa notar o progresso. A concepção popular é de que não é possível concluir algo realmente relevante em poucas horas, mas ao se manter essa prática, nota-se facilmente o contrário.

Após um desenvolvedor efetuar com sucesso o processo de build em sua máquina, antes do commit ser efetuado, é necessário que se efetue a build em uma máquina própria para a integração. Isso é necessário para barrar algum commit que não tenha passado pelo processo de build corretamente ou para evitar que o programa rode apenas da máquina do desenvolvedor e não na máquina ”neutra”de integração. Esse processo pode ser efetuado manualmente (o desenvolvedor efetua o processo de build na máquina de integração) ou utilizando um servidor de integração contínua. Um servidor de integração contínua funciona como uma maneira de monitorar o repositório. Toda vez que um commit é efetuado, o servidor automaticamente atualiza o código em uma máquina de integração, inicia um processo de build e notifica a pessoa que efetuou o commit sobre o resultado da build.

A parte principal de efetuar builds continuamente é que, se a build do trunk falha, ela precisa ser consertada imediatamente. O objetivo de se trabalhar com integração contínua é sempre estar desenvolvendo em uma base estável. Se a build do trunk dá erro frequentemente, isso pode significar que a equipe de desenvolvimento não está sendo cuidadosa o suficiente ao se certificar de sincronizar seus arquivos com o trunk e efetuar a build em sua própria máquina antes de fazer o commit. Se for o caso, medidas devem ser tomadas para certificar que os desenvolvedores cumpram essas etapas antes dos commits.

A intenção da Integração Contínua é prover feedback rapidamente. Nada atrapalha mais esse objetivo do que um processo de build que demora muito tempo. Para a maior parte dos projetos, a XP indica que a build deve demorar no máximo 10 minutos, já que a cada minuto que você reduz do processo de build, é um minuto a mais que cada desenvolvedor dispõe a cada vez que eles efetuam commit. Já que a integração contínua envolve commits frequentes, isso acaba somando bastante tempo. O processo de agilizar a build normalmente requer alterações radicais no código para quebrar as dependências em partes menores ou modificar o conjunto de testes. É necessário que se encontre uma proporção razoável entre a cobertura dos testes e o tempo que eles demoram para serem executados. Suites de testes extremamente demoradas não servem ao propósito da integração contínua.

7. Implementando a Integração Contínua [FOWLER 2006]:

7.1. Automatizar o processo de build

Tornar possível efetuar a build do projeto com apenas uma linha de código.

7.2. Introduzir testes automatizados a esse processo

Aplicar testes automatizados durante o próprio processo de build.

7.3. Agilizar o processo de build

Agilizar o processo de build que é efetuado a cada commit, preferencialmente até a meta de 10 minutos estabelecida pela XP. Para fazê-lo, é possível reestruturar o código ou a bateria de testes.

8. Entrega Contínua

A entrega contínua é um conjunto de práticas que visa produzir código em ciclos curtos, garantindo que seja possível entregar software frequentemente e em sua versão mais recente [HUMBLE 2013]. Para que isso seja possível, é imperativa a aplicação da integração contínua, pois como abordado ao longo deste relatório, a integração de um código pelo método tradicional é trabalhosa e longa, o que impossibilitaria entregas frequentes. Logo, a integração contínua é um requisito básico para que se possa implantar a entrega contínua.

Referências

BARESI, L., P. M. (2006). An introduction to software testing. In *Electronic Notes in Theoretical Computer Science, Vol. 148, No. 1*, pages 89–111.

DELAMARO, Márcio Eduardo; MALDONADO, J. C. (2016). *Introdução ao Teste de Software*. Elsevier - Campus, 2nd edition.

FOWLER, M. (2006). Continuous integration. `https://martinfowler.com/articles/continuousIntegration.html`. Accessed: 2017-07-01.

HUMBLE, J.; FARLEY, D. (2013). *Entrega contínua: como entregar software*. Porto Alegre: Bookman, 1st edition.

MYERS, G. (2004). *The art of software testing*. John Wiley and Sons.

NEVES, V. d. O. (2015). *Automatização do teste estrutural de software de veículos autônomos para apoio ao teste de campo*. São Carlos : Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo - Tese de Doutorado em Ciências de Computação e Matemática Computacional.