Perquirere, 11(2): 141-159, dez. 2014

http://perquirere.unipam.edu.br

XP e Integração Contínua: um estudo de caso de sua adoção no desenvolvimento de *software*

XP and Continuous integration: a case study of its adoption in software development

Leandro da Costa Gonçalves
Pós-graduado em Engenharia de Software (UNIPAM).
E-mail: leandrocgsi@gmail.com

William Chaves de Souza Carvalho
Mestre e Doutorando em Ciência da Computação pela Universidade Federal de Uberlândia (UFU).
E-mail: william@facom.ufu.br

Resumo: O desenvolvimento de software, assim como qualquer outra atividade, tem sofrido constantes mudanças ao longo dos últimos anos. Dentre elas, vale destacar o surgimento de *frameworks ágeis*. Esses *frameworks* trouxeram consigo uma série de novas práticas e uma das mais importantes é a Integração Contínua. Trata-se de uma prática ágil que pode ser adotada inclusive como suporte a metodologias convencionais. Este artigo está centrado no uso do *framework* ágil *XP* e de Integração Contínua, que foram utilizados para o desenvolvimento de uma aplicação web, com fins de controle do setor bibliotecário, e que serviu como estudo de caso.
Palavras-chave: Integração Contínua. TDD. Desenvolvimento Ágil. XP.

Abstract: Software development, as any kind of activity, has suffered constant changes over the past few years. Among them, it is worth mentioning the on set of *agile frameworks*. These *frameworks* brought many new practices and one of the most important is the *Continuous Integration*. It is an agile practice which can be adopted including as a support to conventional methodologies. This paper is centered in the use of *agile framework XP* and Continuous Integration, which were used in development of a web application, for purposes of sector control in a library, and served as a case study.
Keywords: Continuous Integration. TDD. Agile Development. XP.

---

1 INTRODUÇÃO

Nos últimos anos, o desenvolvimento de software cresceu, abrangendo diferentes áreas de trabalho, visando atender às mais variadas necessidades expostas pela sociedade. Neste período ocorreram mudanças significativas na forma de se desenvolver softwares, dentre as quais merecem destaque os frameworks ágeis. Essas mudanças impactaram diretamente nas ferramentas de suporte e apoio ao

desenvolvimento de software. Nesse contexto surgiram o TDD, o ALM, a Integração Contínua dentre outros.

O ritmo frenético de mudanças impactou diretamente na forma que as equipes de desenvolvimento se organizam e interagem entre si. Paralelamente cresceu a necessidade de que se automatizem tarefas repetitivas, economizando, com isso, tempo e dinheiro. A evolução do mercado de software associado aos prazos cada vez menores enfatizou ainda mais a necessidade de agilizar os processos de desenvolvimento de software. E foi nesse cenário que a Integração Contínua veio crescendo e ganhando cada vez mais destaque.

Este artigo visa descrever a experiência adquirida no desenvolvimento de um software para trabalho de conclusão de pós graduação[1] usando o *framework* ágil *eXtremeProgramming(XP)*e Integração Contínua. Ambos utilizam abordagens diferentes da convencional, para o desenvolvimento de software. Este artigo está organizado da seguinte forma: primeiro faz-se a revisão de literatura, depois aborda conceitos básicos da Integração Contínua, descreve como princípios do *XP* e a Integração Contínua foram adotados no desenvolvimento e, por fim, detalha o processo de desenvolvimento durante os s*prints*.

2 REVISÃO DA LITERATURA

Esta seção descreve a revisão da literatura e conceitos referente à engenharia de *software*, frameworks tradicionais e frameworks ágeis de desenvolvimento de software.

2.1 ENGENHARIA DE SOFTWARE

A engenharia de software é uma área da engenharia que se propõe a fornecer parâmetros para o desenvolvimento de softwares. Ela está relacionada a todos os aspectos do desenvolvimento de software, abrangendo desde aspectos iniciais como a especificação de requisitos até processos de manutenção (SOMMERVILLE, 2008). A engenharia de software engloba três elementos – *métodos*, *ferramentas* e *procedimentos* – que permitem controlar o processo de desenvolvimento e oferecem uma base sólida para a implementação de softwares de forma produtiva e com qualidade. Os *métodos* fornecem os detalhes do que fazer para se construir o software, as *ferramentas* fornecem apoio automatizado ou semiautomatizado aos métodos e, por fim, os *procedimentos* formam um elo que conecta os *métodos* e as *ferramentas*, permitindo, assim, o desenvolvimento do software de forma racional e oportuna (PRESSMAN, 2006).

O termo engenharia de software surgiu no final dos anos 60 durante uma conferência em que se discutia a "crise do software". A crise do software, por sua vez, era um resultado direto da evolução tecnológica empregada na fabricação do hardware

[1] O software foi desenvolvido pelos alunos Joelber Flávio dos Santos Garcia, Kéllyson Gonçalves da Silva e Leandro da Costa Gonçalves e seu código está disponível no Github através do link: https://github.com/leandrocgsi/ErudioProjetoTCCPos

de computador baseado em circuitos integrados. Essa evolução viabilizou a implementação de softwares antes considerados impossíveis de serem desenvolvidos. Os softwares resultantes tornavam-se cada vez maiores e o desenvolvimento informal mostrava-se cada vez mais inviável. Projetos de grande porte apresentavam, muitas vezes, anos de atraso. Os custos frequentemente superavam as previsões, o software resultante não era confiável, além de ser difícil de manter e de desempenho insatisfatório (SOMMERVILLE, 2008).

Esse quadro tornou evidente a necessidade de se criarem novos processos de gestão e desenvolvimento de software. Inicialmente os processos de desenvolvimento de software mantinham conceitos típicos da Engenharia. Eles ajudaram a sistematizar o processo de desenvolvimento de software e mais tarde deu origem a Engenharia de Software. Desses processos surgiram as primeiras metodologias de desenvolvimento de software, como a metodologia cascata, a prototipação, o espiral e outros.

2.2 FRAMEWORKS TRADICIONAIS DE DESENVOLVIMENTO DE SOFTWARE

As metodologias tradicionais são também chamadas de pesadas ou orientadas a documentação. Elas foram muito utilizadas no passado em um contexto de desenvolvimento de software muito diferente do atual, baseado apenas em um *mainframe* e *terminais burros.* Naquela época, o custo de fazer alterações e correções era muito alto, uma vez que o acesso aos computadores era limitado e não existiam modernas ferramentas de apoio ao desenvolvimento do software, como depuradores e analisadores de código. Por isso o software era todo planejado e documentado antes de ser implementado. Uma das metodologias tradicionais mais utilizadas até hoje é o modelo Clássico ou Cascata. (SOARES, 2004).

2.3 FRAMEWORKS ÁGEIS DE DESENVOLVIMENTO DE SOFTWARE

A maioria dos conceitos adotados pelos frameworks ágeis nada possuem de novo. A principal diferença entre esses frameworks e as metodologias tradicionais são o enfoque e os valores. Os frameworks ágeis enfocam as pessoas e não os processos ou algoritmos como as metodologias tradicionais. Além disso, existe a preocupação de gastar menos tempo com documentação e mais com a implementação, propiciando, assim, maior interação entre desenvolvedores e clientes (ALVES e ALVES, 2009).

2.3.1 O Manifesto Ágil

Percebendo que a indústria de software apresentava um grande número de casos de fracasso, alguns líderes experientes adotaram modos de trabalho opostos às metodologias tradicionais. Nesse sentido, em 2001, durante uma reunião realizada por 17 desses líderes, concluiu-se que desenvolver software é algo complexo demais para ser definido por um único processo. O desenvolvimento de software depende de muitas variáveis e principalmente é realizado por pessoas em praticamente todas as

etapas do processo (BASSI FILHO, 2008). Nesse encontro chegaram a um consenso quanto a alguns princípios que levavam a bons resultados. Entretanto, concluíram que uma metodologia unificada seria incapaz de atender a todas as particularidades. Desse trabalho surgiu o Manifesto Ágil, cujo foco era o cliente e a agilidade no desenvolvimento de softwares.

O Manifesto Ágil valoriza quatro princípios centrais, que resumem bem o foco do novo processo (BEEDLE, 2001).

- Indivíduos e interação entre eles mais que processos e ferramentas;
- Software em funcionamento mais que documentação abrangente;
- Colaboração com o cliente mais que negociação de contratos;
- Responder a mudanças mais que seguir um plano.

Após a divulgação do Manifesto Ágil, surgiu e/ou ganhou destaque uma ampla gama de novos frameworks denominados Ágeis, dentre as quais as mais conhecidas são eXtreme Programming (XP), a Scrum, a AMDD e a TDD. Esses frameworks mantêm, entre si, muitas características em comum e geralmente diferenças sutis. De acordo com Pressman (2006), esses frameworks ressaltam quatro tópicos-chave: são equipes de desenvolvimento pequenas, entre 2 e 10 membros, que se auto organizam; priorizam mais o desenvolvimento em detrimento da documentação; reconhecem aceitam a mudança além de valorizarem e estimularem a comunicação tanto entre os membros da equipe quanto entre a equipe e o cliente.

Outras características não citadas por Pressmam, mas que merecem destaque, são o fato de serem mais utilizadas em projetos pequenos, embora possam ser aplicadas em grandes projetos. Além disso, assim como no PU, os agilistas adotam o desenvolvimento iterativo a fim de maximizar o feedback e minimizar riscos.

2.3.2 Extreme Programming (XP)

O Extreme Programming é uma framework de desenvolvimento de software criada por Kent Beck, nos Estados Unidos, no final da década de 1990. É uma das frameworks ágeis mais conhecidas e utilizadas no mundo.

2.3.2.1 As origens do XP

As ideias originais do que viria a ser o XP foram definidas por Kent Beck em 1996, por meio do livro *"Smalltalk: best pratice pattners"*, no qual apresentava pontos positivos e negativos de projetos de software. No mesmo ano, Beck foi chamado para conduzir, juntamente com Martin Fowler e Ron Jeffries, um projeto de alto risco na Crysler, o projeto C3. Beck selecionou um conjunto de práticas que haviam se mostrado eficientes em outros projetos e aplicou-as de forma intensiva. A equipe gerenciada por Beck não só conseguiu entregar o software antes do tempo estimado como também criou o framework XP (BASSI FILHO, 2008).

Em alinhamento com as ideias do Manifesto Ágil, o framework XP se baseia em cinco valores, que são:

- Comunicação – para que um projeto atinja seu objetivo com sucesso a comunicação deve ser intensa entre membros da equipe e os *stakeholders*;
- Feedback – as respostas as decisões tomadas e ou mudanças no projeto devem ser rápidas, eficientes e perceptíveis;
- Coragem – é necessário muita coragem para aceitar erros, mudar pontos de vista, se desfazer de antigas idéias;
- Simplicidade – o software, resultante do projeto, deve ser tão simples quanto possível. Além disso deve-se levar em conta que muitas vezes o que o cliente quer é bem mais simples do que o desenvolvedor imagina;
- Respeito – todos tem seu valor dentro da equipe e as individualidades não só devem ser respeitadas como também ser valorizadas. (BASSI FILHO, 2008)

2.3.2.2 A documentação no XP

No framework XP, a documentação é minimalista. Na maior parte dos casos apenas o código fonte e os testes a compõem. Para a XP, um código claro, simples e bem estruturado facilita a compreensão e mudanças no futuro. Com o auxílio de comentários relevantes, o código é a melhor documentação que um software pode ter, além de não se desatualizar. Embora não seja muito indicada, extrair a documentação a partir do código é uma opção utilizada pela XP. Além do código, pouco material é produzido, apenas os *radiadores de informação* e os *cartões de história.*

Os *cartões de história* são feitos de papel e servem para que os clientes e usuários descrevam funcionalidades que desejam no sistema. Os progaramadores utilizam-nos para direcionar a implementação. Os *radiadores de informação*, por sua vez, são gráficos e cartazes que demonstram a produtividade da equipe. Estes devem ficar expostos onde todos os membros da equipe e os clientes possam vê-los.

2.3.2.3 A equipe e os papéis do XP

Uma equipe XP deve reunir o máximo possível de habilidades técnicas e de negócio possíveis para desenvolver o software. A hierarquia entre os desenvolvedores deve ser rasa e não é recomendável estabelecer uma divisão de tarefas. Inicialmente as responsabilidades são distribuidas de acordo com as especialidades de cada um, mas, gradualmente, espera-se que essas especialidades sejam disseminada entre os membros da equipe para evitar a concentração de conhecimento e contribuir para o crescimento profissional de todos os membros da equipe. Apesar disso, existem papéis que determinados membros da equipe podem assumir. Nesse sentido, os papéis mais importantes do XP são:

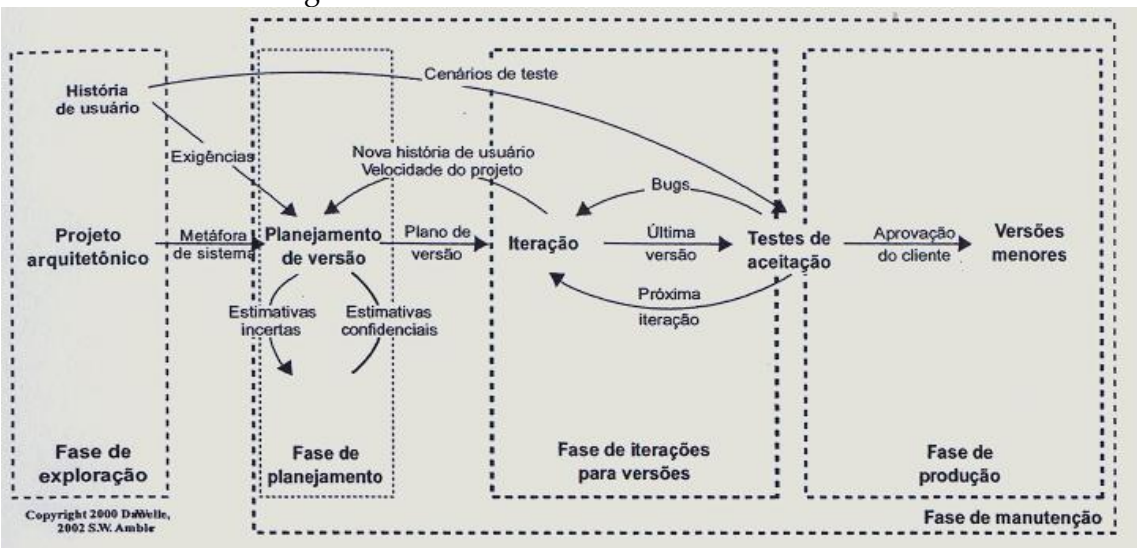
- Os *programadores,* que são maioria dos membros da equipe;
- O *coach,* que geralmente é o programador mais experiente da equipe e deve assegurar que seus membros estejam executando as práticas propostas e garantir que o framework esteja sendo seguido;

- O *tracker* é o desenvolvedor responsável por prover informações referentes ao progresso do projeto e por mostrar pontos que devem ser melhorados. É da responsabilidade do *tracker* elaborar os *radiadores de informação.*
- No framework XP, o *cliente* é considerado parte da equipe, visto que ele conhece as regras do negócio, consegue definir prioridades funcionais do software e além de prover *feedback* do processo de desenvolvimento. Recomenda-se que o cliente esteja presente o tempo todo. Quando isto não for possível, o *coach* assume o papel de *cliente proxy*, responsabilizando-se por repassar informações ao cliente real.

2.3.2.4 O ciclo de vida do XP

A Figura 1 representa o ciclo de vida de um projeto utilizando o framework XP. Como se pode ver, o primeiro retângulo à esquerda representa a *fase de exploração*. É nessa fase que as primeiras *histórias de usuário* são levantadas e o projeto arquitetônico da aplicação é iniciado. As *histórias de usuário* são frases curtas escritas pelo cliente que explica uma funcionalidade que o software deve ter (HENRAJANI, 2007). Ainda na *fase de exploração* são levantados os requisitos e implementados os cenários de teste. Os cenários de testes serão utilizados na *fase de manutenção* e antes da fase de produção. Já os requisitos, por sua vez, serão abordados durante os *planos de versão*.

Figura 1– O Ciclo de Vida do Framework XP



Fonte: HENRAJANI, 2007

Como se pode ver no segundo retângulo, é no *plano de versão* que são definidas as estimativas e quais *histórias de usuário* serão implementadas em uma iteração. A iteração é uma pequena etapa de desenvolvimento ao final da qual será entregue uma *pequena versão* a ser testada. Através dos cenários de testes, as *pequenas versões* são testadas e, se não ocorrerem erros e o cliente aprovar, ele entra na *fase de produção,*

ficando a equipe livre para iniciar uma nova iteração. Já se o cliente desaprovar algo ou ocorrerem bugs terá origem novas *histórias de usuário* que serão reconsiderados no *plano de versão* e adicionados à próxima iteração.

2.3.2.5 As práticas e valores do XP

O XP recomenda um conjunto de práticas que traduzem os valores do XP em ações do dia a dia. As principais são:

- Testes – os testes são muito importantes no XP e devem ser implementados preferêncialamente antes do desenvolvimento;
- Refatoração – sempre que possível o código deve ser simplificado e melhorado;
- Programação Pareada – o XP recomenda que os programadores trabalhem em duplas: assim enquanto um programador digita, o outro observa, pensa em melhorias e alternativas;
- Propriedade Coletiva – o código fonte não pertence a um único programador qualquer um pode modificá-lo e aperfeiçoá-lo.
- Integração Contínua – depois de testada, cada nova funcionalidade deve ser imediatamente sincronizada entre todos os desenvolvedores;
- Semana de 40 horas – a programação simplesmente não irá render se o programador não estiver descansado e disposto;
- Cliente Sempre Presente – o cliente não é alguém de fora, mas sim um membro da equipe;
- Padronizações – se todo o time seguir padrões pré-acordados de codificação, será mais fácil manter e entender o que já está feito. O uso de padrões é uma das formas de reforçar o valor comunicação.

Além dessas peculiaridades, o XP possui uma série de características similares a outros frameworks ágeis. Isso se deve ao fato de todas elas terem ganhado destaque após a divulgação do *Manifesto Ágil* e por seguirem suas recomendações. Além disso, muitas vezes, o XP é utilizado simultaneamente outro(s) framework(s) também ágeis que o complementa como o Scrum e o Lean (HENRAJANI, 2007). Uma das práticas que ganhou força no mundo ágil e consequentemente também no XP foi a Integração Contínua que será abordada no próximo tópico.

3 INTEGRAÇÃO CONTÍNUA

Nos frameworks ágeis, a Integração Contínua possibilita um monitoramento contínuo e uma melhor percepção do andamento e resultados do projeto (GARCIA, 2013), visto que cada nova funcionalidade e alteração que ocorre no projeto é percebido com maior rapidez pela equipe, o que contribui para aumentar a confiança coletiva durante a implementação e também do cliente quando além da Integração Contínua faz se ainda a Entrega Contínua (*Continuous Delivery*) (HUMBLE& FARLEY, 2011).

A Integração Contínua traz consigo uma série de vantagens como o refatory constante, a evolução do código pela equipe de desenvolvimento, de forma paralela e gradativa enquanto adiciona novas funcionalidades na versão corrente (COFFIN &RADY, 2011). Além disso, possibilita um melhor gerenciamento dessa evolução do código. Quando utilizada juntamente com práticas como o TDD garante que o código esteja funcionando corretamente após cada refactory ou adição de uma nova funcionalidade visto que essas práticas facilitam, quando bem aplicadas, a detecção imediata da inclusão de bugs no código (MARTIN, 2011).

A descoberta precoce de bugs pode ser crucial para o bom andamento do projeto, já que quando são detectados em fases avançadas do desenvolvimento pode comprometer o prazo de entrega bem como o orçamento do projeto. Além disso, a correção destes bugs pode gerar novos bugs ocasionando uma reação em cadeia. E isto abala a confiança e credibilidade de todos os membros da equipe envolvidos no projeto (GARCIA, 2013).

A Integração Contínua contribui para melhorar a *visibilidade do projeto*, proporcionando aos *stakeholders* envolvidos ou não com a implementação uma melhor percepção dos resultados obtidos diariamente e se estes estão de acordo com as expectativas. Isso é muito importante pois pode evitar que problemas não detectados durante o desenvolvimento sejam descobertos tardiamente, impedindo uma reação mais adequada por parte da equipe (COHN, 2010). Com a adoção de técnicas e práticas da Integração Contínua, conseguimos agir em todos esses pontos.

3.1 O QUE É MESMO A INTEGRAÇÃO CONTÍNUA

A Integração Contínua surgiu como uma das práticas da metodologia ágil XP, e seu foco é o desenvolvimento de software em ciclos mais curtos, possibilitando uma melhor resposta às alterações e inclusão de novos requisitos ao software. Mas não se limita apenas a equipes que adotam a metodologia XP ou frameworks ágeis, trata-se de um conjunto de boas práticas que podem perfeitamente serem adotadas em metodologias de desenvolvimento convencionais ou híbridas (GARCIA, 2013).

A ideia central por traz da Integração Contínua é a diminuição dos riscos por meio de um melhor monitoramento das alterações e de uma integração frequente do código. Ela é encarada como parte do processo de desenvolvimento, sendo considerada como um procedimento normal e corriqueiro onde a integração do código deve ocorrer frequentemente, se possível a cada nova alteração. Caso ocorram falhas, a Integração Contínua possibilita que seu impacto seja pouco significativo, facilitando a identificação e correção do(s) erro(s).

Para que a integração seja eficaz, precisa-se verificar o funcionamento do código após as modificações, e, para atender a esta necessidade, o desenvolvimento em conjunto com a realização de testes é de suma importância e sempre que possível devem ser feitos utilizando-se da metodologia do TDD (*Test Driven Development*), o que possibilita testar pequenas partes do código de forma rápida e barata (MARTIN, 2011).

A execução de uma integração bem sucedida traz alguns benefícios. Dentre eles podemos destacar a garantia de que o código compila e que as funcionalidades testadas mantêm sua integridade e funcionamento esperados. Além disso, pode eliminar ou minimizar a necessidade de testes manuais, visto que a maioria dos testes pode ser feita de forma automática.

3.2 AS FERRAMENTAS DISPONÍVEIS

Para se garantir a eficácia da Integração Contínua em um ambiente de desenvolvimento são necessárias certas ferramentas que proporcionem funcionalidades fundamentais, como o controle de versão, builds automatizados e execução das integrações. Além disso, quando a equipe faz também Entrega Contínua, é necessário uma ferramenta de deploy contínuo (GARCIA, 2013).

O Sistema de Controle de Versões (SCM) é o responsável por gerenciar as mudanças no código durante o desenvolvimento. Nele, um projeto é representado por uma estrutura de diretórios chamada de repositório. Durante o processo de implementação, o repositório do projeto vai sofrendo alterações, que são divididas em revisões. Cada uma delas possui um identificador único, a data de criação, o conjunto de modificações efetuadas e o responsável. Isso permite que a evolução do código seja rastreada, indicando detalhes do que, quando e por quem as modificações foram realizadas (SHORE & WARDEN, 2008).

As revisões constituem um recurso crucial em sistemas SCM, pois mantém o histórico de alterações e possibilitam a restauração de versões anteriores do repositório. O SCM age como um ponto central no que diz respeito às alterações efetuadas em um projeto, além de facilitar aos membros da equipe o acesso ao conteúdo do projeto (GARCIA,2013).

Por meio dos recursos oferecidos pela ferramenta de SCM, o Servidor de Integração Contínua monitora a criação de novas revisões no repositório. Quando uma nova revisão é criada, o servidor faz um update do código com a versão mais atual no repositório e executa a integração. Dentre as principais ferramentas de SCM disponíveis atualmente podemos destacar o *Concurrent Versions System (CVS)*, *Subversion (SVN)* e o *Git*, sendo o último uma evolução dos dois primeiros.

3.3 BUILDS AUTOMATIZADOS

O *build* de um projeto é o um processo composto por várias etapas, como compilação, execução de testes, empacotamento, geração dos artefatos, e em alguns casos geração automática de documentação e realização do *deploy*. Além dessas etapas principais, podem-se adicionar novas de acordo com as necessidades específicas de cada projeto, bastando, para isso, a adição de scripts adicionais ao processo padrão de build (SHORE& WARDEN, 2008).

O processo de build é repetido todas as vezes em que for necessária a disponibilização de uma nova versão, ou em cada commit, se for o caso. Para a criação

dos *scripts* de *build,* pode se optar por soluções "caseiras" ou por ferramentas de automatização de *build*, como o Apache Ant, que é bastante flexível, que permite uma customização de todas as etapas e/ou da sequência de execução do build. Entretanto, uma das opções mais usadas é o Apache Maven, que, apesar de não ser tão flexível quanto o *Ant,* oferece uma gama de recursos que facilitam a criação do projeto e do *script*, além de padronizar a estrutura de diretórios e o ciclo de *build* (GARCIA,2013).

Entretanto escolher entre scripts caseiros, Ant, Maven ou qualquer outra ferramenta, ainda não é suficiente para se implementar a Integração Contínua. O que realmente importa é poder iniciar o build do projeto de forma simples e rápida. Neste ponto é que entra em cena o Servidor de Integração Contínua, que é o responsável por manter a integração e o processo de build frequente.

3.4 O SERVIDOR DE INTEGRAÇÃO CONTÍNUA

O Servidor de Integração Contínua tem como objetivo principal integrar as alterações ocorridas no repositório através do processo de *build*. Ele normalmente permite a configuração de procedimentos de *build*, para que o processo de integração possa ser executado. Após executar um procedimento de *build*, o Servidor de Integração Contínua disponibiliza o resultado do processo de integração, como *log* de execução, o resultado dos testes e claro se o build foi bem sucedido ou não (HUMBLE& FARLEY, 2011).

Além disso, essas ferramentas também possuem mecanismos de notificação que divulgam os resultados da integração, o que é importante caso seja necessária uma ação rápida para fixar uma falha na integração. Normalmente enviam e-mail, mas podem ser configurados para enviar SMS, além de se integrar às principais IDE's através de *plugins*. O Servidor de Integração Contínua irá manter um ciclo de funcionamento, que consiste no monitoramento e recuperação constante das modificações, executando a integração sempre que houver mudanças no repositório, disponibilizando os resultados na sua própria interface e nos mecanismos de notificação configurados.

As principais opções de Servidores de Integração Contínua existentes no mercado apresentam as funcionalidades citadas anteriormente, dentre eles o *Jenkins* e o *Hudson* do qual o primeiro deriva. O Jenkins é gratuito, multiplataforma, open source e pode ser customizado através da instalação de *plug-ins*. Podemos destacar ainda os plugins, com destaque para os Servidores de Integração *JetBrains Team City, o Circle CI* e o *Atlassian Bamboo*.

3.5 AS PRÁTICAS

Além das soluções de infraestrutura, também é necessária a adoção de certas práticas de integração que devem ser executadas para que os benefícios da Integração Contínua sejam colhidos (GARCIA, 2013). Tais práticas podem afetar a forma habitual

de trabalho, exigindo esforço de toda a equipe, resultando em um ambiente mais controlado e seguro para a mesma. Tais práticas transformam a Integração Contínua um processo natural da implementação de código e reduz o risco de *bugs* e problemas passarem despercebidos ou se acumularem, facilitando o processo de correção (RASMUSSON, 2010).

3.5.1 Commit frequente

O principal objetivo da Integração Contínua é integrar as alterações da forma mais frequente possível, identificando e solucionando os problemas de forma rápida. Manter código sem commitar por muito tempo diminui os benefícios da Integração Contínua e cria ainda alguns inconvenientes como os conflitos durante a atualização com o código atual do repositório. Neste caso, tarefas simples acabam se tornando difíceis, pois o impacto das alterações é maior (RASMUSSON, 2010).

Quando ocorrem conflitos, eles devem ser fixados localmente, antes do commit. Em alguns casos, o desenvolvedor pode esquecer ou propositalmente deixar de atualizar seu código com o repositório e isso acaba por retardar a ocorrência de conflitos até o commit das modificações. Quando isso ocorre, a situação pode ficar ainda mais complicada, prejudicando o processo de Integração Contínua. Idealmente os *commits* deveriam acontecer no mínimo diariamente, possibilitando a toda a equipe conhecer e, se necessário, reutilizar o novo código implementado.

Para aumentar a frequência de *commits*, é necessário subdividir as tarefas a serem desenvolvidas em partes menores e implementáveis separadamente, de preferência utilizando-se do TDD .As micro implementações vão se encaixando ao código existente podendo ser commitado e integrado pelo Servidor de Integração Contínua de forma menos traumática. Entretanto, mesmo commitando frequentemente, os conflitos continuarão a ocorrer. Todavia, como as porções de código são menores, a resolução de conflitos torna-se mais simples.

3.5.2 Faça TDD

O TDD é uma técnica de desenvolvimento orientado a testes que ganhou força com os frameworks ágeis. Ele se baseia em três etapas básicas, a que Martin (2009) denomina de "As três leis do TDD":

1. Você não pode iniciar a codificação de algo que irá para produção sem antes escrever um teste unitário que a princípio irá falhar.
2. Você não pode escrever mais testes até que a implementação permita que o primeiro passe.
3. Você não poderá escrever mais código de produção enquanto seu teste estiver falhando.

Além desses três etapas você pode e deve refatorar seu código, cuidando para que seus testes continuem passando. A principal vantagem do TDD, é que ele garante que para cada funcionalidade desenvolvida existe uma série de testes que asseguram

seu funcionamento correto. Através desses testes, o servidor de Integração Contínua verifica automaticamente se novas funcionalidades inseridas não comprometem o funcionamento esperado de outros módulos da aplicação (MARTIN, 2011).

Existe uma série de frameworks disponíveis com soluções para as mais diferentes necessidades existentes na criação de testes dentre os quais podemos destacar:

1- *JUnit*, *NUnit* e *TestNG*: Para o desenvolvimento de testes unitários;
2- *PowerMock*, *EasyMock* e *Mockito*: Que São utilizados juntamente com o framework de testes unitários. Estes frameworks são usados para simular a referências às dependências externas o que é feito através da criação de *Mocks* (objetos falsos). A principal característica de frameworks de mock é o isolamento do comportamento das dependentes em relação à funcionalidade testada.

Existem outros frameworks que permitem testar acesso a banco como o *DB Unit*. Além disso, a implementação dos testes pode incluir também os testes funcionais que verificam o comportamento das funcionalidades de maneira integrada. Um bom exemplo desse tipo de framework é o Selenium, que testa as funcionalidades através da interface com o usuário, simulando a utilização real da aplicação. Isso possibilita que o teste avalie o resultado de uma operação, verificando a integração de todos componentes que fazem parte de sua implementação (GARCIA, 2013). Obviamente não se pode testar todos os aspectos de um software, entretanto uma boa abrangência nos testes pode minimizar problemas, facilitar correções e minimizar os custos com testes de qualidade.

3.5.3 Não commite código que não funciona

Depois de concluir a implementação de uma nova funcionalidade e antes de commitar, é importante verificar se os testes continuam executando sem falhas. Além disso, é preciso assegurar que todos os arquivos que fazem parte das modificações sejam commitados para o repositório. Não prestar a devida atenção a estas questões provocam falhas na Integração Contínua, que podem ser perfeitamente evitados através de simples precauções (SHORE & WARDEN, 2008).

3.5.4 Faça builds locais antes de commitar

Falhas na Integração Contínua prejudicam a evolução do desenvolvimento, pois dificultam a integração, tornando-a inoperante e interrompendo o versionamento do sistema, exigindo a intervenção imediata da equipe de desenvolvimento. Quando elas acontecem, um integrante da equipe, geralmente o responsável pela falha, deve investigar e corrigir a falha o quanto antes. Enquanto o problema não for resolvido, a Integração Contínua fica indisponível para toda a equipe (MARTIN, 2011).

A fim de minimizar essas falhas na Integração Contínua é recomendável executar o build localmente antes de cada *commit*, de forma que simule a execução que

ocorre no Servidor de Integração Contínua. Essa prática possibilita que falhas sejam percebidas antes do *commit*, minimizando as chances de ocorrência de falhas no servidor (RASMUSSON, 2010).

3.5.5 Mantenha o build rápido

O resultado do processo de *build* é o principal indicador de sucesso da evolução do código, e é fundamental manter o tempo de execução do mesmo mais rápido possível (SHORE& WARDEN, 2008). Entretanto, dependendo do tamanho do projeto e da quantidade e tipos de testes inclusos na execução do *build*, pode dificultar o alcance dessa meta. Uma solução muito comum para esse problema é a criação de tipos específicos de planos de *build*. Pode-se, por exemplo, criar um procedimento de build mais rápido que executa testes leves e outro mais demorado que execute os testes mais pesados ou todos os testes. Desse modo, a cada *commit*, o procedimento mais leve é executado e em menor frequência o procedimento de *build* pesado pode ser executado uma vez ao dia.

3.5.6 Disponibilize os artefatos gerados pelo build

A Integração Contínua possibilita a disponibilização de potenciais versões do sistema a cada integração bem sucedida. Essas versões podem ser utilizadas para diversos fins, como a execução de testes ou demonstrações. Além de disponibilizar novas versões a cada build bem sucedido, os servidores de Integração Contínua permitem a geração de artefatos auxiliares, como relatórios de inspeção de código e documentação. Alguns exemplos de configurações incluem: relatório de cobertura de testes, relatório de violações de padrões de codificação e claro relatório de bugs. Esses artefatos devem ser acessíveis à equipe, pois dão visibilidade em relação à qualidade e maturidade na evolução do projeto (GARCIA, 2013).

4 METODOLOGIA

O ciclo de vida de desenvolvimento do software se baseou no *framework ágil eXtreme Programming*, em que o desenvolvimento foi dividido em sprints de duas semanas e, a cada dia, novas funcionalidades foram sendo continuamente integradas ao sistema. As reuniões de time foram realizadas pela equipe diariamente via *Skype*. Como foram implementadas poucas histórias, não se utilizou do quadro *kanban*, todas as discussões necessárias e esclarecimento de dúvidas foram feitos durante essas reuniões.

A tecnologia de desenvolvimento adotada foi a Plataforma Java, mais especificamente o *framework* de desenvolvimento *Web JavaServer Faces 2.0* e as bibliotecas de componentes *Primefaces 2.2* e *Facelets*. Foram utilizados ainda os *frameworks Spring Security* para prover segurança e autenticação de usuários e o *Hibernate* para gerenciar as transações com o banco de dados. Para formatação de

conteúdo foi utilizado CSS. Todo o tratamento de informações de interface com o usuário foi provido pelo Primefaces e/ou por validadores no lado servidor.

Para facilitar o processo de build foram utilizados os frameworks *Maven* e JUnit, o Servidor de Integração Contínua *Jenkins* oferecido pelo serviço de *Cloud Computing Cloud Bees* e o servidor de SCM Github. Durante esse período, o software foi continuamente deployado em ambiente de desenvolvimento nos servidores da *Cloud Bees*. Desse modo, o time pode implementar o software independentemente da localização geográfica dos membros do time.

O software foi feito a partir de uma reimplementação da versão desenvolvida para a conclusão do curso de graduação dos membros do time. Entretanto não houve um cliente real para essa versão e o software foi desenvolvido apenas com objetivos acadêmicos. Visou-se com isso melhorar as principais funcionalidades do software bem como seus mecanismos de segurança, além de usar uma gama de ferramentas que viabilizasse a integração contínua.

Pelos motivos acima citados o time definiu as sete estórias apresentadas nos quadros 1 e 2. As duas primeiras a serem implementadas estavam relacionadas à definição da estrutura do banco de dados, criação do banco físico e popular com dados iniciais dessas tabelas.

Quadro 1 – Primeiro Sprint Concepção e Construção do Banco de Dados

História	Tarefa	Ferramenta(s)	Artefato
Elaboração do modelo conceitual do banco de dados	Elaborar o modelo conceitual	Case Studio	Modelo conceitual do banco de dados
	Gerar o script SQL	Case Studio	Script SQL para a construção do banco físico
Construção do banco físico	Rodar o script SQL	MySQL Query Browser e MySQL	Banco de dados físico
	Criar usuários	MySQL Query Browser e MySQL	Banco de dados atualizado com usuários e permissões

Fonte: Dados do Trabalho

Após modelar o banco de dados, iniciou-se o processo de codificação propriamente dito. A antiga arquitetura foi revisada para suportar o framework *Maven* o que era essencial para a adoção da integração contínua. Posteriormente foi removido o antigo mecanismo de segurança, que foi substituído pelo *Spring Security*.

O Quadro 2 representa um ciclo padrão pelo qual passaram as demais estórias durante o desenvolvimento. As tarefas apresentadas nele se repetiram sucessivas vezes até que todas as estórias fossem implementadas. A única exceção foi a estória "Prover

Segurança", que não necessitou de testes unitários, uma vez que foi necessário apenas configurar alguns arquivos XML.

Quadro 2 – Segundo Sprint

História(s)	Tarefa	Ferramenta(s)	Artefato
Prover segurança Implementar cadastros de usuários Implementar cadastros de materiais Implementar relatórios Implementar operações da biblioteca	Programação	Netbeans 7.4 MySQL Browsers Apache Tomcat	Releases diários deployados em ambiente de desenvolvimento
	Testes com mocks	Netbeans 7.0 MySQL Browsers Apache Tomcat	Conjunto de testes com dados falsos que simulem a execução real do sistema para garantir a integridade do build
	Testes unitários	Netbeans 7.0 MySQL Browsers Apache Tomcat	Conjunto de testes para garantir a integridade do build

Fonte: Dados do Trabalho

5 ANÁLISE DOS RESULTADOS

A avaliação dos resultados do projeto desenvolvido, que resultou neste artigo, se baseia em históricos de commits no github em logs de builds no Jenkins e no tempo dispendido para solucionar os problemas que aconteceram ao longo do desenvolvimento.

Todo o planejamento das histórias a serem desenvolvidas foram feitos via Skype, o que era bom, pois reduzia a necessidade de deslocamento entre os membros do time. Além disso, permitia que cada membro do time se concentrasse ao máximo nas tarefas que precisava realizar. Isso possibilitou também a realização de "*Pair Programming* remoto" visto que bastava que um dos membros do time, que precisasse de ajuda, compartilhasse sua tela com um dos outros membros do grupo. Por outro lado, isso acabou gerando uma grande dependência do time em relação a essas reuniões em que se definia quem trabalharia em qual tarefa.

A Figura 2 representa a tela de logs de commits do Github usado como servidor de versionamento no projeto. Pelo lado positivo ele permitiu ao time medir sua velocidade através dos logs de *commits*, além de possibilitar reverter alterações que causassem falhas, caso fosse necessário. O lado negativo é que nem todos os membros do time estavam familiarizados com o GIT, o que dificultou a resolução de conflitos de versionamento nos primeiros *sprints*.

Figura 2 – Os logs de commit no GitHub

Fonte: Dados do Projeto

O uso dos frameworks *Maven* e *JUnit* também contribuiu bastante para a produtividade do grupo e garantia do código produzido. O *Maven* permitiu um melhor gerenciamento das dependências (bibliotecas) usadas no projeto, eliminando a necessidade de baixar uma série de bibliotecas manualmente. Já o *JUnit* possibilitou a implementação de uma série de testes que asseguravam que as novas funcionalidades ou correções no software não causavam impactos em outras funcionalidades. Isso deu maior segurança para todo o time codificar. Entretanto, mais uma vez vale destacar que a pouca experiência de alguns dos membros do time com essas tecnologias dificultou um pouco o trabalho no *sprints* iniciais.

Figura 3 – Os builds do Jenkins

Fonte: Dados do Projeto

A Figura 3 representa a tela com os *builds* do *Jenkins* e o resultado de cada integração executada. Ele trouxe uma série de vantagens, pois, além de executar os *builds* do projeto, ainda executava toda a bateria de testes. Sempre que o processo de *build* falhava, o *Jenkins* assinalava o mesmo de vermelho tornando visível para todo o

time que algo de errado estava acontecendo. Vale destacar também a realização do *deploy* do projeto de forma automática a cada *build* bem sucedido, que era assinalado de azul.

Além dos pontos evidentes, o uso da Integração Contínua trouxe outras vantagens indiretas como uma melhoria gradual da qualidade do código, uma vez que o medo de refatorar foi reduzido com a adoção dos testes. O processo de manutenção do software também fica mais fácil com a introdução de testes unitários. Vale destacar também a facilidade de realização de *deploys* do projeto, já que, a cada commit, o build era executado e, se bem sucedido, os artefatos deployados no ambiente de desenvolvimento da Cloud Bees. Com um pouco mais de esforço poderia realizar inclusive o Continuous Delivery.

6 CONCLUSÃO

Durante o desenvolvimento, as reuniões diárias entre os membros do time foram realizadas através do Skype, devido à impossibilidade de todos os membros da equipe se reunirem no mesmo local diariamente. Como era de se esperar, no início do projeto, alguns membros do time enfrentaram dificuldades com algumas ferramentas ou tecnologias, mas essas dificuldades foram diminuindo na medida em que a equipe adquiria familiaridade com as mesmas.

Como o *XP* valoriza software funcionando mais do que documentação, destaca-se como ponto positivo sua utilização. Reduziu se com ele o tempo que gasto com documentação, dando ao time maior tempo para o desenvolvimento, pesquisa e troca de experiências.

A Integração Contínua, por sua vez, trouxe grandes vantagens, uma vez que permitiu ao time codificar e integrar funcionalidades ao software rapidamente. O uso de TDD deu maior segurança ao time. O Jenkins, por sua vez, tornou possível automatizar o build, a execução dos testes unitários e o processo de deploy, o que permitiu ao time desenvolver o software em menos de dois meses.

Conclui-se que quando o time consegue agregar, de forma efetiva, o uso da Integração Contínua, a possibilidade de divergências entre os membros do time sobre o software diminui, visto que cada um está integrando constantemente o código recém criado pelos colegas. Por outro lado, o tempo entre a codificação e a integração das novas funcionalidades ao projeto são minimizadas, o que contribui para que os princípios do *framework* sejam cumpridos, levando o projeto ao êxito.

REFERÊNCIAS

ALVES, Sérgio de Rezende; ALVES, André Luiz. *Engenharia de Requisitos em Metodologias Ágeis.* Goiânia: Universidade Católica de Goiás (PUC – Goiás), 2009. Disponível em: < http://www.cpgls.ucg.br/ArquivosUpload/1/File/V%20MOSTRA%20DE%20PRODUO%20CIENTIFICA/EXATAS/10-.PDF > Acesso em: 04 abr. de 2011.

BASSI FILHO, Dairton Luiz. *Experiências com desenvolvimento ágil.* São Paulo: USP – Istituto de Matemática e Estatística da Universidade de São Paulo, 2008.

BEEDLE, Mike *et al*. *Manifesto para o desenvolvimento ágil de software.* 2001. Disponível em: <http://manifestoagil.com.br/>. Acesso em: 28 fev. 2011.

COFFIN, Rod; RADY, Bem. *Continuous Testing:* With Ruby, Rails, and JavaScript. Dallas, The Pragmatic Bookshelf, 2011.

COHN, Mike. *Succeeding with Agile:* software development using Scrum.Boston, Addison-Wesley, 2010.

GARCIA, Francisco A. *Integração Contínua:* da teoria à prática. Java Magazine. DevMedia. Rio de Janeiro, 2013.

HENRAJANI, Anil. *Desenvolvimento ágil em Java com Spring, Hibernate e Eclipse*. São Paulo: Pearson Prentice Hall, 2007.

HUMBLE, Jez; FARLEY. David. *Continuous Delivery:* reliable software releases through build, test, and deployment automation. Boston, Addison-Wesley, 2011.

MARTIN, Robert C. *Clean Code:* a handbook of agile software craftsmanship. Prentice-Hall Boston, 2009.

MARTIN, Robert C. *The Clean Coder:* a code of conduct for professional programmers. Prentice-Hall Boston, 2011.

PRESSMAN, Roger S. *Engenharia de software*. 6 ed. São Paulo: McGraw Hill/Nacional, 2006.

RASMUSSON, Jonathan. *The Agile Samurai:* how agile masters deliver great software. Dal

SHORE, James; WARDEN, Shane. *The art of agile development***.** O'Reilly. Sebastopol, 2008.

SOARES, Michel dos Santos. *Comparação entre metodologias ágeis e tradicionais para o desenvolvimento de software.* 2004. Disponível em: <http://www.dcc.ufla.br/infocomp/artigos/v3.2/art02.pdf > acesso em: 04 abr. 2011.

SOMMERVILLE, Ian. *Engenharia de software*: 8 ed. Rio de Janeiro: Prentice-Hall, 2008.