PROJETO*: práticas ágeis em um projeto de software livre

Autores

[1]Instituições

Abstract. *Agile and free software development models were born in different contexts, for different reasons. Nonetheless, both models present many common practices and values. In this paper we present PROJETO, a free software project as a case of how agile practices are adopted by free software projects. We will analyse which Extreme Programming practices are adopted by PROJETO, and which ones are not. In either case, we will discuss the reasons for that. Therefore, our contributions is presenting empirical evidences about the relationship between agile practices and free software development using a real example.*

Resumo. *Embora surgidas em contextos diferentes, e com motivações diferentes, há vários aspectos em comum entre as práticas ágeis e as práticas tradicionalmente adotadas em projetos de software livre. Neste artigo apresentaremos o projeto PROJETO como um estudo de caso sobre a aplicabilidade de práticas ágeis por comunidades desenvolvedoras de software livre. Iremos adotar como referência as práticas da Programação Extrema (XP) e iremos explorar quais são as práticas da XP utilizadas e as não utilizadas no projeto, assim como as motivações para a utilização ou não de cada prática. Com isso, nossa contribuição é a apresentação de evidências empíricas sobre a relação entre práticas ágeis e projetos de software livre (SL) em um caso concreto.*

1. Introdução

Embora surgidas em contextos diferentes, e com motivações diferentes, há vários aspectos em comum entre as práticas ágeis e as práticas tradicionalmente adotadas em projetos de software livre (SL). Essa relação entre software livre e práticas ágeis tem sido estudada por pesquisadores [Corbucci and Goldman 2010, During 2006].

Logo no prefácio da segunda edição do livro "Programação Extrema" [Beck 2004], Erich Gamma já aborda as práticas ágeis utilizadas em um projeto de software livre, o projeto Eclipse. Essa menção sugere uma proximidade entre esses dois mundos. No entanto, ainda nesse prefácio, é declarado que nem todas as práticas da XP são usadas no projeto Eclipse. Seriam as práticas da XP não utilizadas nesse caso inadequadas para um projeto SL?

Os métodos ágeis nasceram com um foco bem grande na estruturação de pequenos times co-localizados que se dedicam em tempo integral a um projeto. Já projetos SL, por outro lado, podem ter dezenas ou centenas de colaboradores dispersos geograficamente e que colaboram de forma voluntária e muitas vezes pontual. Essas diferenças põem em cheque a aplicabilidade das práticas ágeis em projetos SL.

*Devido ao sistema de *blind review* do WSL 2015, o nome do projeto utilizado neste estudo de caso será rasurado com "PROJETO". Outros contextos que possibilitem a fácil identificação do projeto serão rasurados com "@@@" .

Para colaborar com a discussão sobre a aplicabilidade das práticas ágeis em projetos SL, neste artigo apresentaremos um projeto de software livre, o PROJETO, como estudo de caso. Adotaremos como referência as práticas da Programação Extrema como representantes das práticas ágeis, e iremos explorar:

- Quais as práticas da XP adotadas no PROJETO e os benefícios que elas trazem no contexto específico de um projeto SL.
- Quais as práticas ágeis não adotadas no PROJETO e os motivos disso.

Nossa intenção é que essas perguntas sejam respondidas com alguma profundidade, uma vez que os autores deste artigo são também desenvolvedores do PROJETO. Assim, com este artigo, esperamos apresentar evidências empíricas sobre a relação entre práticas ágeis e projetos de software livre.

Em nosso contexto, as análises, interpretações e conclusões se restringem a um grupo restrito de projetos de SL: aqueles que não possuem um apoio financeiro regular de alguma instituição e que, portanto, são mantidos por voluntários. Também entendemos como projetos SL *típicos* aqueles que possuem as características apresentadas por Raymond como o modelo Bazar de desenvolvimento [Raymond 1999]. O modelo Bazar procura explicar o funcionamento e o sucesso do modelo de desenvolvimento do kernel do Linux, no qual toda evolução do software e seu código correspondente são imediatamente liberados aos usuários e a outros desenvolverdores. Esse modelo se contrapõe ao modelo de Catedral, no qual novas mudanças são liberadas somente depois que os principais desenvolvedores considerarem que essa evolução já esteja madura o suficiente.

Esperamos que as discussões aqui apresentadas possam ser úteis principalmente para comunidades de SL que estejam adotando ou discutindo a adoção de práticas ágeis. Além das comunidades de SL, acreditamos que possam se beneficiar da discussão desenvolvedores atuando em equipes geograficamente distribuídas, que muito podem aprender com a dinâmica das comunidades de SL.

2. Sobre o projeto PROJETO e seus modos de desenvolvimento

O PROJETO é um aplicativo que @@@.

Características gerais do projeto:

- Iniciado em 2012.
- Possui cerca de 8500 linhas de código Python, 1800 linhas de Java Script e 2700 linhas de HTML.
- Possui cerca de cinco desenvolvedores principais (*core developers*).
- Já recebeu contribuições de cerca de 30 colaboradores diferentes.
- Conta com 451 curtidas em sua página no Facebook.
- Apresentou, em média, pouco mais de 100 visitas mensais nos últimos meses (porém em época @@@, ocorrem picos de acessos maiores do que essa média).

Um projeto SL pode receber contribuições em contextos variados. Cada um desses contextos pode impor um *modo de desenvolvimento* específico, que apresentará forças favoráveis ou contrárias à adoção de determinadas práticas ágeis. No PROJETO, identificamos três modos de desenvolvimento, que podem inclusive coexistir no tempo. São eles:

Desenvolvimento distribuído com contribuição voluntária: esse é o modo clássico de desenvolvimento de software livre, que Raymond descreveu como o modelo Bazar, adotado no desenvolvimento do kernel Linux [Raymond 1999]. Colaboradores geograficamente dispersos contribuem voluntariamente por meios virtuais, que no caso do PROJETO são principalmente a lista de e-mail e o *issue tracker*. O aspecto *voluntário* das contribuições implica a impossibilidade de gerenciar os membros como em um projeto tradicional. Uma consequência é a impossibilidade de se prever ou estimar quando uma determinada tarefa será concluída. Embora a comunidade possa eleger prioridades, também não é possível controlar o que cada colaborador irá fazer, pois são livres para escolher as tarefas que acharem mais atraentes. Esse é o modo predominante no desenvolvimento do PROJETO que, por brevidade, será referido como *modo distribuído*.

Desenvolvimento com equipe co-localizada em *hackathons*: eventualmente membros da comunidade se encontram por alguns dias para um esforço intensivo de desenvolvimento. Normalmente esse esforço se insere no contexto de uma competição, para a qual a equipe participante do *hackathon* estipula metas a serem atingidas nesse curto espaço de tempo (normalmente de 1 a 4 dias). O PROJETO já participou de pelo menos três *hackathons* competitivos promovidos por organizações externas ao projeto. A participação nesses eventos parece ser uma prática comum em projetos SL. O projeto PyPy, por exemplo, organiza periodicamente *hackathons* não-competitivos, denominadas de *sprints* [During 2006]. Esse modo de desenvolvimento será referido como *modo hackathon*.

Desenvolvimento no contexto de uma disciplina universitária: em cursos superiores relacionados ao desenvolvimento de software é comum a existência de disciplinas que demandem o desenvolvimento de projetos de software por estudantes. Alguns cursos já têm incentivado seus alunos a desenvolverem esses projetos criando ou contribuindo com SL [Santos et al. 2012]. Em particular, o primeiro semestre de 2015 corresponde ao quinto semestre consecutivo em que o PROJETO recebe contribuições de alunos do curso de Engenharia de Software da Universidade @@@. Nesse modo de desenvolvimento, um grupo de estudantes com cerca de cinco membros contribui com o projeto ao longo de um semestre letivo, dedicando-se em média 8 horas por semana, das quais pelo menos 4 horas de forma co-localizada. Nesse contexto, é possível imprimir aos estudantes uma disciplina maior no desenvolvimento, exigindo-se o uso de certas práticas ágeis, bem como oferecer uma revisão mais criteriosa sobre o código entregue. Por outro lado, os estudantes são livres para contribuir com as funcionalidades que acharem mais atraentes. Nos referiremos a esse modo de desenvolvimento como *modo disciplina*.

Cabe esclarecer algumas peculiaridades da disciplina @@@, que têm contribuído com o PROJETO. A disciplina tem a intenção de proporcionar um ambiente real de desenvolvimento de software baseado na metodologia da XP, assim, nela, os alunos interagem diretamente com comunidades de SL. Embora o professor possa interferir na formação dos grupos para garantir algum balanceamento técnico entre eles, a princípio os alunos escolhem os projetos com que querem trabalhar. Dessa forma, os grupos são formados com base no interesse pessoal de cada pessoa, estabelecendo em paralelo com o universo real de trabalho do SL.

Vale ainda ressaltar que o trabalho em grupo no contexto da disciplina é importante, mas as contribuições individuais têm grande peso na nota final dos alunos. O conceito de meritocracia está normalmente presente nas comunidades de SL, inclusive no próprio PROJETO. Ao contribuir, os estudantes começam atacando tarefas simples, tanto pelo fato de não conhecerem o projeto a fundo, tanto por não terem a confiança da comunidade para assumir demandas mais complexas.

3. Métodos ágeis e a Programação Extrema

Os métodos ágeis nasceram como uma alternativa aos chamados métodos tradicionais de desenvolvimento de software. Nos chamados métodos tradicionais, há muita ênfase em processos, ferramentas, documentação, contratos e planejamento. Já os métodos ágeis procuram valorizar mais as interações entre indivíduos, software funcionando, colaboração com o cliente e a aceitação de mudanças [Beck et al. 2001].

A Programação Extrema é uma metodologia ágil introduzida por Kent Beck, e possui como valores a comunicação, a coragem, o *feedback*, o respeito e a simplicidade. Apresentaremos agora brevemente as práticas da XP descritas no livro *Extreme Programming Explained* [Beck 2004]:

Sentar-se junto: a equipe deve trabalhar em um mesmo espaço de trabalho para promover a comunicação. Espaços abertos são preferíveis aos cubículos. A comunicação rápida e direta facilita a resolução de conflitos entre os desenvolvedores, o que muitas vezes surge de visões diferentes sobre possíveis soluções para um determinado problema do projeto.

Equipe integral: a equipe deve incluir pessoas de forma a conter todas as habilidades necessárias para o sucesso do projeto. Exemplos de perfis possivelmente necessários: desenvolvedores, analistas de negócios, administradores de sistemas, testadores, etc. Todos devem estar comprometidos com o sucesso do projeto.

Ambiente informativo: algum interessado deve ser capaz de entrar no ambiente de trabalho e obter em pouco tempo uma ideia do andamento do projeto. Com mais uma olhada, ele deve ser capaz de captar possíveis problemas que estejam acontecendo. Essas informações também ajudam o time a relembrar alguns problemas no projeto. Como exemplo, destacar a quantidade de entregas diárias de software em ambiente de produção ajuda o time a relembrar sobre a qualidade e o tempo gasto nas entregas [Humble and Farley 2010].

Trabalho energizado: trabalhe apenas o tempo em que conseguir manter um ritmo produtivo. O desenvolvimento de software é um trabalho criativo, por isso trabalhar mais horas não é o melhor caminho para a entrega de um produto melhor.

Programação em par: dois membros da equipe trabalham juntos codificando e revisando o código em revezamento. Essa prática tenta resolver um problema comum da solidão do desenvolvedor com o código. A troca de experiência ajuda no aprendizado e na inovação das práticas de programação. Além disso, o código (junto de seu *design* e testes) já nascem de forma muito mais criteriosa e, portanto, com maior qualidade.

Histórias: o cliente deve escrever pequenas histórias que ilustrem novas funcionalidades ou capacidades do sistema. O esforço de implementação dessas histórias é estimado pela equipe, e assim elas podem ser priorizadas pelo cliente.

Ciclo semanal: a equipe e o cliente devem realizar um planejamento semanal do que a equipe irá entregar na próxima semana. Isso promove o desenvolvimento incremental e acelera o ciclo de *feedback*.

Ciclo trimestral: além do planejamento semanal, um planejamento de maior prazo é realizado trimestralmente, no qual é possível haver mais reflexão sobre o projeto, além de discussões sobre os gargalos enfrentados.

Folga: deixe sempre uma folga no planejamento. Se sobrar tempo, é possível entregar algo a mais ou utilizar o tempo extra para aprendizado. Mas se comprometer com o que não pode ser entregue pode gerar muitos desgastes nos relacionamentos com o cliente e com a organização.

***Build* de dez minutos:** o sistema deve ser construído (compilado, testado e empacotado) em até 10 minutos. Um tempo maior é um incentivo para que o desenvolvedor não execute o *build* com frequência, o que reduz o *feedback* e deteriora a qualidade do produto.

Integração contínua: suas mudanças no código devem ser continuamente integradas com o trabalho de outros colegas. Quanto mais tempo se espera para integrar diferentes fluxos de trabalho, maior será o esforço e a dificuldade em se completar com sucesso a integração. Para cada integração, é necessário que uma verificação de erros seja feita automaticamente através de um *build* automatizado.

Desenvolvimento Orientado a Testes: escreva primeiro um teste falhando para especificar o comportamento do próximo trecho de código a ser escrito. Assim o desenvolvedor codifica apenas o código realmente necessário. A detecção de defeitos no software é facilitada, diferentemente da situação na qual o teste é feito muito depois do código de produção, quando a depuração pode consumir muito tempo. Além disso, a escrita de testes automatizados fornece constante *feedback* sobre a qualidade do *design* do código.

Design incremental: investimento em um bom *design* deve ser realizado constantemente ao longo do desenvolvimento do sistema. Um bom *design* promoverá a capacidade de introduzir mudanças no sistema posteriormente, o que é o grande objetivo da XP (*abrace a mudança*). A melhoria contínua do *design* é preferível a uma grande fase inicial de *design*.

4. Trabalhos relacionados

Outros pesquisadores têm estudado a relação entre práticas ágeis e projetos de SL. Corbucci e Goldman aplicaram questionários nas comunidades de software livre e de métodos ágeis para descobrir que princípios e práticas eram compartilhados por ambas as comunidades [Corbucci and Goldman 2010]. As duas comunidades apresentaram respostas razoavelmente similares sobre ferramentas preferidas, tais como *issue tracker* ou alerta de *build* quebrada, assim como sobre a eficiência de diferentes meios de comunicação, tais como canais de IRC ou troca de e-mails.

During relata a experiência do projeto PyPy [During 2006], um projeto SL que adotou práticas ágeis desde seu início com bastante naturalidade. O artigo discute a dificuldade em se conciliar a mentalidade de software livre / projeto ágil com as necessidades burocráticas do patrocínio da União Europeia. Ou seja, para o autor a mentalidade de projetos SL converge sem problemas com a mentalidade ágil, enquanto que ambas apresentam problemas para se encaixar nos processos burocráticos de gerenciamento e documentação de um "projeto europeu".

Ahalt et al. apresentam um processo de desenvolvimento de software livre com a adoção de práticas ágeis no contexto de pesquisas científicas [Ahalt et al. 2014]. O grande foco do processo proposto está na maior colaboração entre pesquisadores e desenvolvedores de softwares científicos. Essa colaboração é proposta dentro de uma perspectiva ágil. Já o aspecto de software livre se apresenta como um meio para um maior reaproveitamento e compartilhamento de soluções em uma comunidade científica, o que leva inclusive ao desenvolvimento de software de melhor qualidade, contrastando com diferentes softwares com o mesmo propósito sendo desenvolvidos por pesquisadores isolados. Novamente, esses autores encaram projetos SL e práticas ágeis como aspectos convergentes.

Tsirakidis et al. tentam encontrar fatores de sucesso e de falha em projetos ágeis executados em organizações desenvolvedoras de software livre [Tsirakidis et al. 2009]. Baseando-se em entrevistas com membros de dois desses projetos ágeis e na análise de perfis de seus participantes, os autores concluem como fatores de sucesso: 1) comunicação síncrona e constante, em contraste com conversas espalhadas por diferentes listas de e-mails; 2) consistência na metodologia de desenvolvimento; 3) cultura de testes; e 4) aceitação e tratamento de limitações do ambiente. O mais importante desse artigo, em nosso contexto, é que ele apresenta uma evidência de que uma "organização desenvolvedora de SL" adota oficialmente práticas ágeis de desenvolvimento de software.

Temos também Turnu et al. que apresentam um modelo para simular o processo de desenvolvimento de projetos de software livre [Turnu et al. 2004]. Eles utilizam o modelo para comparar a produção de SL utilizando Desenvolvimento Guiado por Testes (TDD) [Beck 2002] *versus* a produção do SL sem utilizar essa prática ágil. Eles concluem que a prática de TDD em projetos SL aumenta o tempo gasto para se escrever uma linha de código de produção, mas que porém reduz os defeitos no software.

À exceção de Corbucci e Goldman, os demais estudos investigados indicam que a ideia de aplicar práticas ágeis em projetos SL aparece de maneira natural, desejável e sem empecilhos. Já Corbucci e Goldman começam questionando esse relacionamento, mas por meio de questionários concluem que as práticas adotadas nas comunidades não são tão diferentes. Vale ressaltar que esse questionário abordou apenas alguns aspectos do desenvolvimento, se focando em ferramentas e canais de comunicação.

Não encontramos artigo fazendo análise mais extensiva sobre a compatibilidade de diferentes práticas ágeis com o estilo de desenvolvimento e gerenciamento normalmente adotados em projetos SL. Dessa forma, esperamos fornecer uma contribuição para que a comunidade tenha um maior entendimento sobre as especificidades em se adotar práticas ágeis em um projeto SL.

5. Práticas da XP e suas aplicabilidades em um projeto de software livre

Nesta seção descreveremos a relação entre as práticas da XP e as adotadas no PROJETO. Quando a prática for aplicada, se possível, tentaremos sugerir uma explicação de como essa prática ajuda e entregar valor no contexto específico de um projeto de SL. Quando a prática não for aplicada, tentaremos encontrar as razões para tal e se há práticas alternativas no projeto que procuram atingir o mesmo objetivo que a prática da XP.

Para que a análise das práticas da XP no PROJETO possa ser generalizada para outros projetos SL, iremos contextualizar, quando aplicável, essa comparação com base

no que Raymond define como as características de projetos SL desenvolvidos no modelo de Bazar [Raymond 1999]. Assim, ao identificar características do PROJETO com aquelas apresentadas por Raymond, caracterizamos o PROJETO como uma amostra válida de um típico projeto SL.

5.1. Sentar-se junto

O compartilhamento de um espaço físico comum entre os membros da equipe é a primeira grande diferença entre equipes ágeis, formadas por profissionais que trabalham juntos em uma organização, e equipes de projetos SL, tradicionalmente distribuídas e formadas até por pessoas nem se conhecem pessoalmente. Nesse modo distribuído de desenvolvimento, é inviável seguir a prática de compartilhamento de um espaço físico de trabalho.

Porém, é preciso considerar que o maior objetivo do compartilhamento do ambiente de trabalho é a promoção da comunicação. No PROJETO, a ausência da co-localidade é compensada pela comunicação via lista de e-mail e *issue tracker*. Há também ocasionalmente a promoção de *hackathons*, onde o trabalho co-localizado é exercido intensamente por um curto período de tempo. A utilização de listas de e-mail e *issue tracks* é bem comum também em outros projetos SL [Corbucci and Goldman 2010]. Também encontramos evidências sobre a promoção de *hackathons* periódicos em outros projetos [During 2006].

No contexto de contribuições feitas no *modo disciplina*, a prática de sentar-se junto vem se mostrando bastante promissora. Quanto mais próximos os alunos estão, mais fácil é a disseminação do conhecimento e mais rápido se dá o alinhamento do time em relação a determinado assunto. Obviamente o time deve ainda interagir com a comunidade do PROJETO, que está geograficamente distante. Isso dá ao aluno tanto a experiência de trabalhar com pessoas fisicamente próximas, e ao mesmo tempo interagir com outros desenvolvedores no modo distribuído, se valendo de lista de e-mail, *issue tracker* e vídeo-conferências.

5.2. Equipe integral

Acreditamos que discutir a aplicação dessa prática a projetos SL não se aplique, uma vez que todos os contribuidores de um projeto SL formam a "equipe". Em organizações formais, a orientação para uma equipe integral faz sentido pois muitas vezes há uma equipe comprometida com o produto, mas algumas atividades são terceirizadas para outras equipes da organização, tais como atividades de testes ou implantação. Em projetos SL a princípio não há o conceito de "outras pessoas da organização fora da equipe".

No entanto, uma interpretação mais específica pode ser feita para o *modo disciplina*: o sucesso da equipe no contexto da disciplina não pode depender de membros do projeto que não estejam no grupo da disciplina. O mesmo vale para *hackathons*: atingir um objetivo em um *hackathon* não pode depender de contribuidores não presentes no *hackathon*. Assim, no contexto de uma disciplina ou de um *hackathon*, a equipe integral é uma prática importante a ser seguida em projetos SL.

5.3. Ambiente informativo

Em um projeto SL não faz sentido falar de um ambiente *físico* informativo. Mas no PROJETO, temos algumas informações que poderiam formar um espaço *virtual* informativo,

tais como: estado da *build* executada pela integração contínua, quantidade de visitas à aplicação no período, número de curtidas no Facebook no período, quantidade de linhas de código entregues no período, quantidade de *issues* fechadas no período, métricas de qualidade de código (atualmente em andamento). Algumas dessas informações já são coletadas pelo próprio sistema de repositório, no caso o Github. Na Figura @@@, é mostrada, por exemplo, a quantidade de entregas de código em relação ao tempo.

No contexto do PROJETO, o problema é que todas essas informações estão dispersas em páginas diferentes. Mas se fossem integradas em um único espaço virtual, formariam um ambiente informativo equivalente à prática da XP, o que poderia trazer novos benefícios ao projeto. Assim acreditamos que essa prática da XP é aplicável e desejável a projetos SL.

5.4. Trabalho energizado

No modo de desenvolvimento distribuído, cada colaborador age voluntariamente e, portanto, é autônomo para decidir em quê empenhará seu esforço ao longo do tempo. Um colaborador pode individualmente seguir a prática de trabalho energizado e trabalhar apenas quando se sentir produtivo, mas acreditamos que não faz sentido falar dessa prática como algo adotado corriqueiramente pelo projeto.

No caso dos *hackathons*, a situação é extremamente oposta. Como são raros esses momentos e com apenas poucos dias de duração, é comum que os participantes trabalhem durante todo o dia, às vezes até a madrugada. É a tentativa de aproveitar ao máximo possível um momento de esforço intensivo e de possibilidade de interação presencial com os outros membros do projeto.

No *modo disciplina* a lógica é similar ao *hackathon*, mesmo que menos intensa. Mesmo que apliquem algum esforço extra-classe, em geral é apenas na sala de aula que todos os membros estão co-localizados, fazendo com que essas quatro horas semanais devam ser aproveitadas o máximo possível, mesmo que eventualmente em alguns dias nem todos os membros estejam tão "energizados".

5.5. Programação em par

Devido a distribuição geográfica e às limitações tecnológicas ainda existentes, a programação pareada é rara no *modo distribuído*. No entanto, é comum que um desenvolvedor mais experiente no projeto revise o código entregue por colaboradores mais ocasionais. Essa revisão é feita de forma mais intensa e criteriosa no *modo disciplina*. Já no modo de *hackathon*, a programação em par é normalmente adotada, possibilitando a troca de conhecimentos e ideias entre colaboradores que raramente se encontram presencialmente.

No *modo disciplina* o grupo define sua própria estratégia de revisão, mas o desenvolvimento é sempre feito em pares no horário de sala de aula. Normalmente, cada time possui um *coach*, que fica responsável por direcionar a equipe, priorizar o que deve ser feito na iteração, e levar o time a refletir sobre questões técnicas que muitas vezes passam desapercebidas. Essa organização interna do time, muitas vezes é influenciada tanto pela comunidade do SL quanto pela formação técnica dos alunos.

5.6. Histórias

Nos modos *hackathon* e *disciplina*, o time elege funcionalidades a serem implementadas em um dado período de tempo. Desenvolvedores mais experientes no projeto ajudam o time a equacionar o compromisso entre valor e dificuldade de implementação da história. Assim nos aproximamos da prática da seleção de histórias da XP. O problema é que não há uma sistemática para que a seleção de histórias seja influenciada por usuários da aplicação.

Raymond é enfático ao longo de seu artigo sobre a importância do envolvimento dos usuários. "Ouça seus clientes", "A melhor coisa depois de ter boas ideias é reconhecer boas ideias dos seus usuários", "Se você tratar seus *beta testers* como seus recursos mais valiosos, eles irão responder tornando-se seus mais valioso recurso" [Raymond 1999]. Essas e outras passagens do texto deixam clara a importância central do envolvimento do usuário em um projeto SL.

Isso tudo talvez seja uma lição sobre uma lacuna a ser trabalhada no PROJETO. Atualmente procuramos ouvir o *feedback* de usuários nos eventos comunitários dos quais participamos. Uma importante funcionalidade do PROJETO, que é @@@, surgiu de sugestões de diversos usuários que reportaram que tal funcionalidade era desejada. No entanto, esse contato com os usuários é muito esporádico. Considerando suas 451 curtidas no Facebook e suas mais de 100 visitas mensais, acreditamos que alguns usuários poderiam ser mobilizados para fornecer um melhor *feedback* aos desenvolvedores do PROJETO.

5.7. Ciclo semanal

No *modo distribuído* de desenvolvimento com contribuições voluntárias, tal prática não é imposta, já que nesse modo as contribuições normalmente ocorrem em intervalos maiores do que uma semana. Já no modo de *hackathon*, o evento todo constitui um único ciclo, curto. Por fim, mesmo no *modo disciplina*, uma semana é um período curto demais, já que o time trabalha por apenas 8 horas.

Porém, cabe lembrar que a primeira edição da XP [Beck and Fowler 2000] apresentava a prática de *fases pequenas*. Essa prática se relaciona com o princípio de "libere cedo e frequentemente; e ouça seus clientes", utilizada no desenvolvimento do *kernel* do Linux [Raymond 1999]. Ou seja, mais do que entregar semanalmente, o intuito é entregar o mais cedo possível para acelerar o ciclo de *feedback*.

Embora o PROJETO não apresente ciclos semanais, suas funcionalidades costumam ser entregues o mais cedo possível. O maior exemplo foi o nascimento do próprio projeto, que nasceu não como um sistema em si, mas meramente como um *post* de *blog* apresentando uma análise específica. Dessa forma, acreditamos que projetos SL que seguem o princípio de "libere cedo e frequentemente" se adequam ao intuito da prática da XP denominada "ciclo semanal".

5.8. Ciclo trimestral

Reflexões de médio e longo prazo sobre o projeto são também sugeridas por Raymond: "Frequentemente, as soluções mais impressionantes e inovadoras surgem ao se perceber que o seu conceito do problema estava errado" [Raymond 1999]. Ambos os métodos de

Raymond e de Beck reconhecem que mais difícil do que construir é saber o que deve ser construído.

No PROJETO não há ciclos ou momentos formais para essas reflexões. No entanto, o início da colaboração dos alunos das disciplinas a cada semestre gera um debate sobre as prioridades do projeto, principalmente sobre demandas que podem ser potencialmente atacadas pelos alunos. Com isso, o semestre acaba se tornando o intervalo de tempo entre esses momentos de maior reflexão sobre o projeto.

5.9. Folga

Acreditamos que esse princípio não se aplica ao *modo distribuído* de desenvolvimento com contribuições voluntárias, pois normalmente não há comprometimento de prazos por parte dos colaboradores. Por outro lado, acreditamos que a "folga" possa se aplicar ao planejamento de *hackathons*, pois durante o evento é interesse o time ter um objetivo cuja estimativa de execução não chegue a ocupar todo o tempo do evento, já que bloqueios imprevistos no desenvolvimento acontecem.

Quanto às contribuições no *modo disciplina*, a folga não é algo pensado no planejamento das primeiras iterações do time. Grande parte dos alunos não possuem um grande conhecimento sobre os métodos ágeis ou mesmo sobre as tecnologias envolvidas no projeto. Assim, os alunos inicialmente não são capazes de prever o quanto são capazes de produzir ao longo de uma iteração. Consequentemente não são capazes de determinar uma "folga" adequada em seus planejamentos iniciais. No contexto da disciplina, refinar melhor essa capacidade de planejar com folga pode levar várias iterações.

5.10. *Build* de dez minutos

O PROJETO possui uma bateria de testes automatizados. O maior foco é nos testes de unidade, que são os que fazem testes das classes e métodos do código (isolando totalmente ou não as unidades). O projeto possui ainda alguns poucos testes de integração com serviços externos (@@@), mas nenhum teste de usuário, no qual a interface seria manipulada de forma automatizada sob a perspectiva do usuário. A presença de poucos testes de integração e nenhum teste de aceitação faz com que a *build* seja executada em menos de 2 minutos.

Acreditamos que independentemente do modo de desenvolvimento, esta prática seja importante e deva ser seguida por qualquer projeto de software no qual o uso de testes automatizados se aplique.

5.11. Integração contínua

Raymond apresenta a Lei de Linus como "Dados olhos suficientes, todos os erros são triviais". Embora não seja explícito, a Lei de Linus se relaciona com a filosofia ágil como um esforço para execução de testes de maneira contínua e intensiva. Segundo Raymond, projetos SL alcançam isso com uma "base grande o suficiente de beta-testers e co-desenvolvedores". Já times ágeis tradicionais, normalmente não dispõe de uma base grande de usuários ou desenvolvedores, e por isso implementam o esforço contínuo e intensivo de testes por meio da automação. Apesar dessa dicotomia inicial, uma vez que a automação de testes se tornou bem aceita no mundo do desenvolvimento de software,

a execução contínua de testes automatizados passa a ser mais uma ferramenta para o aumento da qualidade de softwares livres.

No PROJETO, a cada entrega realizada no repositório, a bateria de testes automatizados é executada. Caso a *build* falhe, um e-mail é enviado aos membros do projeto, dando visibilidade imediata do ocorrido.

Acreditamos que a existência de uma bateria de testes automatizados traga um valor a mais para projetos SL dependentes de voluntários, quando comparado a projetos de software em geral. Isso porque colaboradores dedicam seu pouco tempo livre ao desenvolvimento do projeto. A existência de uma bateria de testes dá confiança ao colaborador de que ele não está inserindo defeitos no projeto, dispensando-o do esforço de uma compreensão geral do sistema além do escopo de sua contribuição. Tal esforço seria um grande inibidor de colaborações.

Na integração contínua do PROJETO estamos tomando um passo além ao investir num fluxo de entrega contínua [Humble and Farley 2010], no qual a integração contínua dispara a implantação da nova versão da aplicação caso os testes tenham sucesso. Novamente, no contexto de projetos de SL dependentes de voluntários isso tem um valor a mais, uma vez que se diminui o esforço de gerenciamento manual da infraestrutura. Sem a implantação automatizada, toda contribuição de um colaborador só entrega valor ao usuário depois que outro colaborador com acesso à infra atualiza o sistema em produção. Com a implantação contínua, toda colaboração entrega valor imediatamente ao usuário. Essa rápida entrega também acelera o ciclo de *feedback*, o que é importante, pois quanto mais cedo um voluntário tenha o *feedback* de suas alterações, maiores são as chances de que ele esteja disposto a refinar sua solução.

Acreditamos que a implementação do fluxo de entrega contínua seja uma estratégia crítica para o projeto, pois atualmente um esforço em um *hackathon* não pode ser publicizado sem a participação no evento de um dos dois membros do projeto com domínio da infra. No *modo disciplina* isso também é estratégico, já que os alunos não possuem acesso à infra de produção.

5.12. Desenvolvimento Orientado a Testes

No *modo distribuído*, não é possível impor uma disciplina de "testes antes", como advoga o TDD, já que o código é entregue com os testes, não sendo possível a verificação do que veio antes. Mesmo a entrega de código com testes automatizados é uma exigência que em alguns contextos poderia inibir a colaboração. Portanto, no PROJETO a entrega de testes automatizados não é exigida. Por outro lado, às vezes desenvolvedores refatoram e entregam testes sobre código já entregue por outros colaboradores.

No modo *hackathon*, a pressa em se gerar resultados tendo em vistas uma possível premiação é um grande fator para a geração de dívida técnica. Ou seja, nessa situação é comum que muito código seja produzido sem testes automatizados. Mas assim como no modo distribuído, em alguns contextos parte da dívida técnica é paga depois do *hackathon*, com a produção de testes automatizados.

O *modo disciplina* é o mais favorável à aplicação de uma maior disciplina na entrega de código com testes automatizados. Nesse contexto, é possível colocar como objetivo da disciplina a prática de TDD. Com isso, é comum que o código entregue no

modo disciplina possua testes de unidade.

Por fim, mesmo com as dificuldades apresentadas, vale ressaltar que na última medição, o PROJETO apresentou um índice de cobertura de testes de 64%[1], o que consideramos um resultado satisfatório.

5.13. Design incremental

"Os programadores bons sabem o que escrever. Os grandes sabem o que re-escrever", "Planeje jogar algo fora; você irá, de qualquer maneira" [Raymond 1999]. Essas características do modelo de desenvolvimento apresentado por Raymond convergem com a filosofia ágil de evitar um grande esforço antecipado de *design*. Na XP, trechos do sistema são "jogados fora" continuamente com a prática de *design incremental*, pois o código está sendo constantemente evoluído por meio de refatorações.

Em todos os modos de desenvolvimento do PROJETO, os desenvolvedores mais experientes do projeto procuram manter uma base de código limpo [Martin 2008]. Isso é ocasionalmente feito com refatorações sobre código entregue por outros colaboradores. Ou seja, um *design* elaborado não é condição para que uma colaboração seja aceita, pois sabemos que o *design* pode evoluir.

No contexto do PROJETO, identificamos desafios para a manutenção de um *design* limpo devido a código existente que não está gerando valor para o projeto. A existência desse tipo de código acontece por dois motivos: 1) a compreensão sobre as necessidades não estava clara na primeira tentativa de codificação de uma certa funcionalidade; ou 2) os desenvolvedores tentaram se antecipar construindo uma base para futuras funcionalidades que acabaram não sendo implementadas. Em ambos os casos, o código gerado deve ser mantido livre de defeitos, o que ocasionalmente exige esforço de outros desenvolvedores preocupados com outras funcionalidades. Nessas situações pode ser preciso esperar o momento adequado para "jogar algo fora", pois os colaboradores não devem ser desmotivados a continuarem a colaborar com o projeto. Acreditamos que esse tipo de problema possa ser comum em outros projetos SL.

6. Ameaças à validade

O primeiro e maior viés que temos é discutir os relacionamentos entre SL e práticas ágeis com base na análise de apenas um projeto SL. Contudo, esse viés é normalmente encontrado em estudos de caso. Em compensação, diferentemente de estudos quantitativos envolvendo uma grande amostra de projetos, podemos mais facilmente apresentar explicações com certa profundidade sobre as relações encontradas. Além disso, nossa análise é embasada na literatura, tendo como principal referência sobre SL o artigo "A Catedral e o Bazar" [Raymond 1999], e como principal referência sobre práticas ágeis o livro "Programação Extrema" [Beck 2004].

Outro viés é a presença do modo de desenvolvimento do PROJETO em disciplinas universitárias. Nesse contexto, os alunos já são orientados a seguir determinadas práticas ágeis, e cobrados por isso. Isso pode causar uma maior adesão a práticas ágeis no PROJETO do que talvez se encontre em projetos sem essa característica. Para tratar esse viés, procuramos realizar as análises de forma separada para cada modo de desenvolvimento

[1]Ou seja, ao se executar os testes, 64% das linhas de código são exercitadas, sem contar código de teste.

encontrado no PROJETO, destacando assim as peculiaridades do modo disciplina. Além disso, as contribuições dos alunos no *modo disciplina* não se distanciam tanto do *modo distribuído* ao se considerar que os alunos estão livres para escolher quais funcionalidades implementar.

Por fim, as práticas da XP foram adotadas como referência de práticas ágeis. Contudo, se nossa referência tivesse sido outra, como o *scrum* [Schwaber 2004] ou o próprio manifesto ágil [Beck et al. 2001] por exemplo, o resultado poderia ter sido bem diferente. No entanto, achamos razoável a escolha da XP por apresentar um grande conjunto de práticas diretamente relacionadas ao desenvolvimento de software, diferentemente do *scrum* que é um arcabouço para o gerenciamento de projetos em geral. Também preferimos para o estudo as práticas da XP aos valores da XP ou princípios do manifesto ágil, pois consideramos que a aderência a valores ou princípios não é tão objetivamente verificável quanto a adesão a práticas.

7. Conclusão

Os métodos ágeis foram pensados para times profissionais que trabalham em tempo integral e de forma co-localizada. Esse cenário diverge drasticamente de um típico projeto de software livre, com uma quantidade muito maior de colaboradores interagindo voluntariamente de forma distribuída. O prefácio de Erich Gamma no livro da XP já ilustra um projeto de SL que utiliza algumas, mas nem todas, práticas da XP. Dessa forma, dúvidas nascem sobre a possibilidade da adoção de práticas ágeis no contexto de projetos SL.

No entanto, ao longo de nosso estudo de caso vimos que boa parte das práticas da XP são diretamente aplicáveis em projetos SL. Outra parte não é diretamente aplicável, mas é passível de adaptações que procurem atingir os mesmos objetivos da prática original. Notamos também que a aplicabilidade de algumas práticas depende não somente da natureza do projeto, mas do *modo de desenvolvimento* aplicado no momento. Em nosso estudo de caso identificamos três modos: distribuído com contribuições voluntárias; *hackathons*; e disciplinas universitárias.

Outra análise relevante foi a grande intersecção encontrada nos propósitos das práticas da XP e dos princípios do modelo Bazar de desenvolvimento apresentado por Raymond na obra considerada o manifesto do SL [Raymond 1999]. Isso também evidencia como, apesar de algumas divergências superficiais, esses dois mundos possuem grande afinidade, principalmente quando contrapostos a modelos mais tradicionais de desenvolvimento de software.

Em particular, acreditamos que a fusão dos mundos do software livre e da XP no contexto de disciplinas universitárias proporciona um grande crescimento aos alunos. Do ponto de vista do estudante, um dos grandes ganhos envolvidos com as práticas ágeis na contribuição com um projeto SL é alcançar a auto-organização. Em certo momento, o papel de *coach*, que normalmente é o aluno com maior conhecimento técnico na linguagem ou no *framework* que o projeto utiliza, perde um pouco o seu foco como entidade central do grupo, justamente porque a equipe alcança um certo nível de maturidade que proporciona a auto-organização. A explicação dessa evolução, apesar de empírica, é clara na maioria dos times. As equipes entendem como a comunidade funciona internamente, entendem como eles (alunos) funcionam como time, e equilibram a curva de aprendizado. Esses fatores levam à auto-organização do time.

Por fim, é preciso mencionar a característica auto-adaptativa dos métodos ágeis. O manifesto ágil declara que "Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo" [Beck et al. 2001]. Isso significa que os métodos ágeis não devem ser seguidos cegamente como uma receita, mas sim adequadamente adaptados para cada contexto em que são utilizados, como no caso do contexto particular de projetos de software livre.

Referências

Ahalt, S., Band, L., Christopherson, L., Idaszak, R., Lenhardt, C., Minsker, B., Palmer, M., Shelley, M., Tiemann, M., and Zimmerman, A. (2014). Water science software institute: Agile and open source scientific software development. *Computing in Science & Engineering*, 16(3):18–26.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley.

Beck, K. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). Manifesto for agile software development.

Beck, K. and Fowler, M. (2000). *Planning Extreme Programming (XP)*. Addison-Wesley.

Corbucci, H. and Goldman, A. (2010). Open source and agile methods: Two worlds closer than it seems. In *Agile Processes in Software Engineering and Extreme Programming*, pages 383–384. Springer.

During, B. (2006). Trouble in paradise: the open source project pypy, eu-funding and agile practices. In *Agile Conference, 2006*, pages 11–pp. IEEE.

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

Raymond, E. S. (1999). *The Cathedral & the Bazaar*. O'Reilly.

Santos, V. A., Goldman, A., and Santos, C. D. (2012). Uncovering steady advances for an extreme programming course. *CLEI Electronic Journal*, 15(1):2–2.

Schwaber, K. (2004). *Agile project management with Scrum*. Microsoft Press.

Tsirakidis, P., Kobler, F., and Krcmar, H. (2009). Identification of success and failure factors of two agile software development teams in an open source organization. In *Fourth IEEE International Conference on Global Software Engineering (ICGSE 2009)*, pages 295–296. IEEE.

Turnu, I., Melis, M., Cau, A., Marchesi, M., and Setzu, A. (2004). Introducing TDD on a free libre open source software project: a simulation experiment. In *Proceedings of the 2004 workshop on Quantitative techniques for software agile process*, pages 59–65. ACM.