Abordagens para o ensino de práticas de programação extrema

Mariana Vivian Bravo

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Mestrado em Ciência da Computação
Orientador: Prof. Dr. Alfredo Goldman Vel Lejbman

Durante o desenvolvimento deste trabalho a autora recebeu auxílio financeiro da CNPq

São Paulo, Fevereiro de 2010

Resumo

Métodos ágeis de desenvolvimento de *software* constituem uma alternativa aos métodos tradicionais na qual se valoriza mais os aspectos humanos relacionados ao desenvolvimento de *software*. O ensino de práticas ágeis é fortemente baseado no uso e aplicação dessas práticas, ao invés de em exposições conceituais sobre elas. Este trabalho propõe duas maneiras de auxiliar no ensino dessas práticas. A primeira é o uso de *Dojos* de programação, um encontro de desenvolvedores que têm como objetivo resolver exercícios de programação. A segunda é o uso de ferramentas que forneçam *feedback* automatizado da maneira do usuário trabalhar com relação às práticas ágeis. Além de apresentar em detalhes estas duas soluções, apresentamos também uma validação destas técnicas como abordagens para o ensino de práticas ágeis.
Palavras-chave: ensino de programação extrema, *dojo* de programação.

Abstract

Agile methods for software development represent an alternative to traditional methods in which human aspects of software development are given special attention. The teaching of agile practices is strongly based in the use and application of such practices, as opposed to theoretical exposures on the subject. This work proposes two approaches to help in teaching agile practices. The first one is the use of coding *Dojos*, a meeting where developers solve programming challenges together. The second approach is the use of tools that give automated feedback on the user's actions in regard to agile practices. Besides describing in details these two approaches, we also report on studies to validate them as tools for teaching agile practices.
Keywords: teaching extreme programming, coding *dojo*.

Sumário

Capítulo 1

Introdução

Os métodos ágeis correspondem a uma forma de se desenvolver *software* onde ao invés de se criarem processos formais e diversos níveis de documentação existe uma preocupação com outros valores como: pessoas e a comunicação entre elas, *software* funcionando, colaboração com o cliente e adaptação a mudanças. Logo, o ensino de métodos ágeis deve considerar diversos fatores que nem sempre podem ser vistos de forma completamente objetiva.

O ensino de métodos ágeis não é exatamente como o ensino de linguagens de programação, por exemplo. Para linguagens de programação, existem sintaxe e semântica bem definidas, assim como regras de funcionamento objetivas. As relações de causa e efeito são claras, mesmo que talvez o iniciante não as entenda muito bem.

O mesmo não é verdade para a atividade de desenvolver *software*, que é do que tratam os métodos ágeis. Boas práticas que o desenvolvedor pode usar em seu cotidiano são sugeridas, mas não existe uma maneira correta ou absoluta de realizá-las. A utilização de uma prática está sujeita a muitos fatores e seus efeitos podem variar com isso. Neste contexto, a experiência que os desenvolvedores possuem usando uma prática vale muito mais do que seu conhecimento teórico sobre ela. Por isso, muitas abordagens de ensino de métodos ágeis valorizam atividades práticas. Considerando esses fatores, o objetivo deste trabalho é propor e avaliar maneiras de facilitar o ensino de práticas de métodos ágeis voltadas para o desenvolvimento de *software*. É interessante notar que o ensino de métodos ágeis é bem mais amplo do que apenas as práticas abordadas neste texto – por exemplo, existem práticas que não estão diretamente ligadas a programação.

A primeira maneira proposta é através do *Dojo* de programação. Trata-se de um ambiente seguro no qual desenvolvedores de diferentes níveis se reúnem para resolver desafios de programação utilizando algumas práticas ágeis para isso. O objetivo dessas reuniões é o estudo e discussão de diferentes tópicos, desde algoritmos e *design* do código até as práticas usadas para desenvolvimento.

O potencial do *Dojo* de programação como ferramenta de aprendizado reside em dois fatores principais. O primeiro é a repetição das práticas de desenvolvimento em quase todas as sessões, o que propicia ao participante diversas oportunidades de adquirir experiência com essas práticas. O segundo é a presença de outros desenvolvedores enquanto o participante pratica, criando um ambiente onde ele recebe *feedback* a respeito de seu desempenho com aquela prática. Para validar se essa técnica pode ajudar no aprendizado de práticas ágeis, conduzimos uma pesquisa entre participantes de *Dojo* de programação e analisamos os resultados.

A segunda maneira proposta é através de ferramentas que observem a atuação do programador enquanto ele trabalha e forneçam *feedback* de como estão suas ações com relação a algum objetivo

almejado. Esse *feedback* poderia ser na forma de dicas ou lembretes a respeito de atitudes que são consideradas boas para o desenvolvimento de *software*.

Consideramos que essa ferramenta tem potencial para o ensino de práticas ágeis pois ela diminui, até certo ponto, a necessidade de desenvolvedores mais experientes na prática para fornecer esse *feedback*. Para validar essa suposição, desenvolvemos um protótipo dessa ferramenta com foco em algumas práticas e avaliamos os efeitos de usá-la através de um experimento com estudantes de métodos ágeis no IME-USP.

As principais contribuições deste trabalho são as seguintes:

- Um estudo bibliográfico sobre o panorama relativo ao ensino de métodos ágeis;
- Duas propostas para a ajudar o ensino de algumas práticas da programação extrema;
- Uma validação dessas propostas.

Este texto está organizado da seguinte forma: no Capítulo 2 descrevemos os métodos ágeis e, em particular, a programação extrema e apresentamos um panorama de como o ensino de práticas e métodos ágeis é feito em universidades. No Capítulo 3 descrevemos a proposta do uso do *Dojo* de programação para ensino de algumas práticas ágeis específicas, apresentando a origem e funcionamento desta técnica. No Capítulo 4 explicamos a proposta do uso de ferramentas para *feedback* automatizado para o ensino de práticas ágeis e descrevemos as funcionalidades e a arquitetura do protótipo implementado. Nos Capítulos 5 e 6 apresentamos a metodologia e os resultados da validação do *Dojo* de programação e das ferramentas para *feedback* automatizado, respectivamente. Por fim, no Capítulo 7 apresentamos as conclusões deste trabalho.

Capítulo 2

Métodos ágeis

Em métodos tradicionais de desenvolvimento de *software*, os projetos são divididos em algumas fases distintas e sequenciais, entre elas: análise de requisitos, desenho da arquitetura, implementação e testes. Em cada fase, são produzidos documentos e, no caso da implementação, código, para serem usados em fases posteriores do projeto. Tais métodos têm como premissa o planejamento prévio de cada fase e se apoiam em documentação detalhada e ferramentas específicas para controlar os processos.

Os métodos ágeis surgiram como alternativa a esses métodos tradicionais e têm como fundamento os termos do Manifesto Ágil [2]:

- **Indivíduos e interações** valem mais que processos e ferramentas;
- ***Software* que funciona** vale mais que documentação abrangente;
- **Colaboração com o cliente** vale mais que negociação de contratos;
- **Reagir a mudanças** vale mais que seguir um plano.

Percebe-se por esses termos quais são as principais críticas das pessoas ligadas ao Manifesto Ágil com relação aos métodos tradicionais de desenvolvimento, uma vez que o balanço de valores nesses métodos é considerado justamente o contrário do colocado no manifesto. Em geral, nos métodos ágeis, as fases citadas acima ocorrem ao mesmo tempo e iterativamente, com o mínimo de documentação que seja suficiente e com mais responsabilidade para os indivíduos envolvidos no projeto do que para as ferramentas que eles utilizam. A principal medida de progresso do projeto é *software* funcionando e o cliente deve se envolver com esse progresso do começo ao fim.

Existem vários métodos ágeis que surgiram a partir dessas ideias e cada um propõe diferentes abordagens para realização de projetos de *software*. Um dos mais conhecidos e ensinados nas universidades é a programação extrema. Este trabalho se baseia principalmente nas práticas da programação extrema, por isso ela será melhor explicada a seguir. Após a explicação de programação extrema, este capítulo entrará em mais detalhes do ensino da mesma em universidades.

2.1 Programação extrema (XP)

A programação extrema (abreviadamente, XP) foi proposta por Kent Beck em 2000, no livro "Extreme Programming explained: embrace change" [4]. Em 2004, Beck lançou uma segunda edição

do livro [6], em que praticamente apresentou um novo método, incorporando mudanças adotadas após quatro anos de amadurecimento. Nesta seção, descrevemos XP principalmente com base na segunda edição.

O método baseia-se em cinco valores centrais: comunicação, simplicidade, *feedback*, coragem e respeito. Esses conceitos ajudam a guiar as decisões e ações de uma equipe XP quando passando por mudanças ou novas situações. Além deles e com base neles, XP é formada por um conjunto de práticas a serem utilizadas no cotidiano da equipe.

No que concerne esse trabalho, as práticas da programação extrema podem ser divididas em dois grupos: as **práticas de projeto**, que estão ligadas ao planejamento, e as **práticas de código**, mais relacionadas à programação. Enquanto o primeiro grupo é essencial para o entendimento do método e de como ele se relaciona com os princípios ágeis, é o segundo grupo que este trabalho abordará com mais detalhes.

As práticas de projeto podem ser vistas como atividades para gerenciar os requisitos e o progresso do projeto. Elas envolvem o modo de trabalhar de uma equipe XP com relação a tudo menos o código propriamente dito. A seguir algumas dessas práticas são descritas, com ênfase naquelas que são necessárias para entender o funcionamento do método:

- **Histórias**

  Uma história é uma unidade funcional para o cliente e uma unidade de trabalho para o desenvolvedor. Ela é definida e priorizada pelo cliente, estimada pelos desenvolvedores e usada para planejamento do projeto. É através dela que os requisitos de um projeto são capturados, não de forma completa e abrangente, mas de forma breve e independente. Isto é, cada história é um pedaço de funcionalidade que o cliente deseja no *software*, de preferência independente de outras histórias, e descrita com poucas palavras. O nível de detalhamento de sua descrição aumenta conforme se aproxima o momento de implementá-la, através de explicações do cliente e dúvidas dos desenvolvedores. Em outras palavras, a história vai se tornando mais clara quanto mais próxima fica a implementação.

- **Ciclo semanal e de estação**

  Também conhecido como **iterações pequenas**, o **ciclo semanal** dá ritmo ao desenvolvimento do projeto. A cada iteração, as histórias existentes são reavaliadas tanto em prioridade quanto em estimativa e com base nisso a equipe planeja a próxima iteração. Esse ciclo deve ser curto para permitir que mudanças sejam rapidamente levadas em consideração. Por exemplo, ao se perceber, no final de uma iteração, que a estimativa para uma determinada história cresceu muito, o cliente pode querer deixá-la de lado em prol de histórias mais curtas, de forma a ter mais funcionalidades desejadas.

  O **ciclo de estação** é semelhante ao semanal, exceto que o planejamento é feito em um nível mais alto. As histórias tratadas geralmente têm menor granularidade, são mais como temas do que funcionalidades. Conforme essas histórias entram nas iterações, elas são quebradas em tarefas ou histórias menores. Geralmente é no final do ciclo de estação que ocorrem as entregas do sistema, por isso ele também pode ser chamado de ***release***. Da mesma forma que as iterações, os *releases* também devem ser curtos, de forma a obter e considerar o *feedback* do cliente e dos usuários o quanto antes.

- **Envolvimento real com o cliente**

  O envolvimento real com o cliente significa que o cliente faz parte da equipe tanto quanto os programadores. Ele deve ser o mais disponível possível para tirar dúvidas, verificar funcionalidades e reconsiderar prioridades. A presença constante do cliente cria um ambiente de comunicação e *feedback* que melhora o valor que o programa entregue agrega ao cliente. Portanto é também de seu interesse participar de forma tão intensa do projeto.

  Muitas vezes não é possível ter o envolvimento real com o cliente e por isso são usados representantes que atuam como ligação entre ele e a equipe. Esse representante, conhecido como cliente *proxy* [21], pode ser um programador ou gerente que entra em constante contato com o cliente ou pode ser alguém apontado pelo próprio cliente, que entenda bem do domínio da aplicação, para cumprir o papel.

- **Área de trabalho informativa**

  A área de trabalho de uma equipe XP deve conseguir transmitir informações relevantes sobre o projeto de forma rápida e acessível para a equipe e para o cliente. Dados como quanto tempo falta para terminar a iteração, qual é o estado atual de cada história ou se existem dúvidas pendentes a respeito dos requisitos são exemplos de informações que podem ser úteis numa área de trabalho informativa. Para conseguir transmitir essas informações, em geral faz-se uso de gráficos grandes e visíveis desenhados em lousas ou cartazes, quadros com *post-its* ou até meios mais criativos como lâmpadas coloridas que indicam o estado atual dos testes.

  É importante cuidar para que a área de trabalho não fique sobrecarregada com informações, sob o risco de não conseguir transmitir nenhuma informação relevante. Se alguma informação que está na parede se tornar inútil para a equipe, ela deve ser removida.

  Algumas equipes escolhem um responsável para cuidar da atualização, adição e remoção de informações na área de trabalho. Esse papel é conhecido como ***tracker*** e pode variar ao longo do projeto.

- **Implantação incremental**

  O sistema é implantado para uso conforme iterações ou *releases* terminam, de maneira incremental e antes que todas as funcionalidades esperadas ou imaginadas estejam prontas. Isso permite que os usuários aproveitem o programa produzido ao máximo, gerando possivelmente novos pedidos e *feedback* sobre as tarefas prontas e em andamento. Nem sempre é possível realizar a implantação incremental, e a decisão de lançar ou não o *software* cabe ao cliente.

- Outras práticas: **Sentar junto**, **Time completo**, **Folga**, **Trabalho energizado**, **Implantação diária**, **Contrato de escopo variável**, **Redução do time**, **Continuidade do time**, **Pague pelo uso** e **Análise de causa inicial**

As práticas de código, por outro lado, estão relacionadas à programação e são usadas com muita frequência no cotidiano dos desenvolvedores. As principais delas são descritas a seguir:

- **Programação em pares**

  Em XP, recomenda-se que todo código entregue para o cliente seja produzido em pares. Dois programadores desenvolvem uma tarefa juntos, no mesmo computador. Um deles é o

"piloto", que digita o código; o outro é o **"co-piloto"**, que fica ao lado procurando por erros e oportunidades de melhoria. Eles devem trocar de papel com frequência. Baseada no princípio de redundância para revisão de código, a programação em pares ajuda a diminuir a quantidade de *bugs* no sistema e, ao contrário, do esperado ela não diminui a produtividade da dupla pela metade [14, 40]. Outra forte recomendação é que os programadores troquem de dupla com frequência, para aumentar a circulação de conhecimento e a comunicação dentro da equipe.

- **Integração contínua e código compartilhado**

  Essa prática é usada para melhorar a comunicação e o *feedback* dentro da equipe de desenvolvimento. O código produzido por cada dupla deve ser integrado com a maior frequência possível, de forma que a equipe possa ver o sistema como um todo e sempre atualizado. Isso previne problemas muito comuns que ocorrem quando cada programador faz uma parte do sistema e elas só são testadas em conjunto no final, com testes de integração.

  A integração contínua e programação em pares promovem, em conjunto, a noção de **código compartilhado**, o que significa que nenhum indivíduo na equipe é dono de um pedaço do código. O código pertence à equipe inteira e é responsabilidade de todos manter sua qualidade alta.

- ***Design* incremental**

  O *design* da arquitetura do sistema, assim como o escopo do mesmo, é definido aos poucos em cada iteração. Não existe uma fase de projetar toda a arquitetura. Ao invés disso, essa prática dita que o código seja implementado da maneira mais simples possível que seja aceitável em termos de modelagem e que funcione. Ou seja, novas funcionalidades não devem ser feitas com simples remendos ao sistema existente, mas sim como adições à arquitetura. Técnicas de **refatoração** [20] – alterar o código de forma a melhorar sua legibilidade sem alterar seu comportamento – e **padrões de projeto** [22] – soluções conhecidas para problemas de arquitetura recorrentes – ajudam a manter o sistema claro e bem estruturado.

- **Testes automatizados**

  Uma equipe XP deve escrever testes automatizados [24, pp. 189–197], [25] para uma tarefa antes de desenvolver o código correspondente. Isso ajuda a dupla a focar apenas na funcionalidade que foi pedida e a manter o sistema testável, o que em geral leva a um *design* com baixo acoplamento e alta coesão.

  Existem várias maneiras de classificar os testes automatizados, mas podemos destacar dois tipos principais para XP: testes de unidade e testes de aceitação. Os **testes de unidade** verificam o funcionamento das classes ou rotinas do sistema. Eles ajudam os desenvolvedores a saber como cada unidade de construção do sistema funciona. Já os **testes de aceitação** são usados para validar que uma história funciona como especificado. Eles podem ser feitos manualmente, mas o ideal é que sejam automatizados com alguma ferramenta.

- **Desenvolvimento dirigido por testes**

  Essa é uma técnica bem específica usada para desenvolvimento de testes de unidade e código em conjunto [5]. Ela consiste de uma sucessão de pequenos ciclos em que escreve-se primeiro

um teste que falha, em seguida faz-se passar este teste e por fim refatora-se o código.

O teste que falha deve ser uma pequena adição ao conjunto de testes que já existem, em geral apenas um caso a mais. Isso é feito para que o ciclo não seja muito grande, mantendo assim um bom ritmo [26] para o desenvolvimento. Se o teste for um passo grande demais, as etapas seguintes ficam mais longas; e quanto mais longas, maior o risco de que as modificações introduzam erros no programa. A ideia é rodar os testes a cada poucas modificações no código, pois desse modo se algum teste que estava passando falhar, é certo que a falha foi inserida na última modificação. Assim, uma vez pronto o teste que falha, todos os testes são rodados.

Em seguida, altera-se o código para fazer o novo teste passar. A modificação deve ser o mais simples e óbvia possível, sem perder muito tempo com detalhes de clareza e organização do código. Outro ponto importante é escrever apenas o suficiente para que o teste passe, sem implementar nenhum detalhe a mais. A ideia é não escrever código a não ser que algum teste precise dele para passar. Se feito cuidadosamente, isso ajuda a garantir que não existe uma instrução cuja remoção ou modificação não faça com que algum teste falhe. Dessa forma, os testes acabam com uma excelente cobertura do código.

Para fechar o ciclo, refatora-se o código atual para procurar torná-lo mais claro, melhor organizado, para eliminar duplicação, entre outros. O mesmo cuidado para não escrever código desnecessário deve ser tomado nessa etapa. Quando terminar, volta-se ao começo do ciclo fazendo um pequeno teste que falha.

- **Código e testes**

  Em um time XP, os testes automatizados devem ser considerados tão valiosos quanto o código. Eles devem ser mantidos sempre passando . Se uma funcionalidade mudar, tanto o código quanto seus testes devem ser alterados para refletir a mudança. O time dedica quanto tempo for necessário aos testes, inclusive investindo em refatorações para que o código dos testes seja mais legível. Dessa forma, os testes servem como uma documentação atual do código. São uma forma de documentação pois ilustram como as classes e métodos do sistema podem ser utilizados e o que se espera de suas funcionalidades. São atuais pois, uma vez que sempre passam, pode-se confiar que documentam o estado atual do sistema.

- Outras práticas: ***Build* de 10 minutos**, **Repositório único de código**

Diferentemente da divisão apresentada acima, na segunda edição de "Extreme Programming explained", Beck separa as práticas em duas categorias: as primárias e as corolárias. A primeira categoria agrupa práticas mais simples de serem adotadas e mais independentes entre si. Elas são benéficas individualmente, mesmo que tragam mais vantagens quando usadas em conjunto. A segunda categoria junta práticas consideradas mais avançadas e difíceis de adotar em um ambiente não ágil. Elas requerem maiores mudanças da cultura e forma de trabalho da equipe de desenvolvimento como um todo. Na Tabela 2.1, mostramos a relação entre a divisão das práticas em primárias e corolárias apresentada por Beck e a divisão em de projeto e de código proposta acima.

Além das práticas explicadas acimas, existem diversas outras que fazem parte do repertório de uma equipe XP. Em seu livro "The Art of Agile Development" [34], Shore e Warden descrevem diversas outras práticas que ficaram apenas implícitas ou nem foram mencionadas nas primeiras

	Práticas primárias	**Práticas corolárias**
Práticas de projeto	Histórias Ciclo semanal Ciclo de estação Sentar junto Time completo Folga Trabalho energizado Área de trabalho informativa	Implantação diária Implantação incremental Contrato de escopo variável Redução do time Continuidade do time Envolvimento real com o cliente Pague pelo uso Análise de causa inicial
Práticas de código	Programação em pares *Build* de 10 minutos *Design* incremental Desenvolvimento dirigido por testes	Código e testes Código compartilhado Repositório único de código

Tabela 2.1: *Práticas primárias e corolárias classificadas em de projeto e de código*

descrições de XP. Em particular, gostaríamos de destacar duas práticas importantes para a aplicação de XP: **retrospectivas** e ***coaching***.

A **retrospectiva** [17] é uma maneira de propiciar reflexão, melhoria e aprendizado a um time. Ela é uma reunião que pode ser realizada em diversos momentos do projeto – em geral ao final de cada iteração, mas também ao final de *releases*, em momentos críticos ou quando um projeto terminar. O objetivo da retrospectiva é permitir que a equipe reflita sobre seu processo atual, procurando lições a se aprender e oportunidades de melhoria. Um dos formatos mais comuns para uma retrospectiva de iteração consiste das seguintes etapas: primeiro, a equipe faz "brainstorming" de pontos que merecem destaque na última iteração, classificando-os em categorias (por exemplo, "O que queremos manter?" e "O que queremos melhorar?"). Em seguida, a equipe organiza e prioriza os pontos levantados de forma a identificar um ou dois problemas mais importantes que aconteceram na iteração. Por fim, a equipe discute possíveis soluções para esses problemas e escolhe uma solução por problema para ser tentada na próxima iteração.

Um aspecto muito importante da retrospectiva é que ela seja um mecanismo para a equipe **efetuar mudanças** na sua maneira de trabalhar. Uma retrospectiva perde bastante do seu sentido se a equipe identificar possíveis soluções para problemas que está enfrentando em seu dia-a-dia mas não executá-las – os mesmos problemas continuarão acontecendo. Outro aspecto que vale à pena mencionar é que as retrospectivas muitas vezes podem alterar as próprias práticas que a equipe utiliza em seu processo.

A segunda prática que queremos destacar é o **coaching**, descrita em detalhes por Davies e Sedley [15] e por Adkins [3]. Algumas equipes fazem uso de um papel especial: o ***coach*** XP. Ele é o líder da equipe, responsável por ajudar no uso das práticas, nas reuniões de planejamento e nas dificuldades que possam surgir no cotidiano, tanto técnicas quanto não técnicas. O trabalho do *coach* não é de dizer o que cada um deve fazer mas sim de ajudar a equipe a se auto-organizar,

liderando por exemplo com suas próprias atitudes. O *coach* não é um papel exclusivo, ou seja, ele também atua como programador.

2.2 Ensino de programação extrema

Existem diversos trabalhos publicados a respeito de como organizar uma disciplina de ensino de programação extrema ou de algumas de suas práticas, bem como sobre os resultados obtidos com isso, do ponto de vista dos instrutores e dos alunos. Os contextos e soluções para inclusão de XP e suas práticas no ensino em computação variam em diversos aspectos. Alguns dos pontos que ajudam a caracterizar cada experiência são descritos a seguir:

- O **grau de experiência dos alunos** varia desde alunos no primeiro ano de faculdade até alunos de pós-graduação com vários anos de experiência no mercado.
- O **tipo de curso** ministrado e os **objetivos de aprendizado** associados com o curso também diferem. Alguns dos cursos são voltados exclusivamente para o ensino de XP enquanto outros são cursos voltados a engenharia de software como um todo nos quais alguma parte de XP é aplicada. Outros ainda são cursos não relacionados diretamente a engenharia de software mas nos quais algumas práticas de XP são utilizadas, como por exemplo cursos introdutórios a programação ou cursos sobre estruturas de dados.
- A **duração** dos cursos e a **distribuição das aulas** na semana é outro fator de variação importante. Os cursos podem durar desde algumas semanas a um semestre inteiro. Alguns deles possuem tempo oficial de aula em laboratório para que os alunos trabalhem em projetos, enquanto em outros espera-se que os alunos completem projetos em tempo extra-aula.
- As **práticas utilizadas** nos cursos também variam bastante, apesar de a maioria dos trabalhos mencionar programação em pares, testes e planejamento.
- As **características do projeto** podem variar tanto no aspecto de proximidade da realidade quanto no aspecto de quem é o cliente. Alguns projetos são totalmente fictícios, como desenvolver um programa que faz operações com matrizes. Em outros casos, o projeto tem uma aplicação real (por exemplo, um *site* para compra e venda de produtos) mas não existe um cliente de verdade interessado em utilizá-lo – em geral, o instrutor atua como cliente nesse caso. Finalmente, em alguns cursos o projeto é de interesse de algum cliente externo real, que pode ou não ter a intenção de usá-lo quando desenvolvido.
- As **características da equipe** também podem variar em aspectos como **tamanho**, **heterogeneidade** dos grupos, **presença ou não de um *coach*** (algumas vezes pode ser um instrutor), **distribuição geográfica**, etc.

Com o objetivo de contextualização, apresentamos a seguir alguns desses trabalhos, os cenários de ensino, as práticas utilizadas e os resultados que eles relatam.

R. Müller [31] descreve um curso que consistiu de um semestre prático com dedicação integral de quatro alunos da pós-graduação. O projeto desenvolvido era uma máquina virtual `Java` para uso em pesquisa pelos instrutores, que atuavam tanto como *coaches* quanto como clientes. O autor

relata sucesso com uso de iterações curtas, *design* incremental, testes automatizados, repositório de código e programação em pares. São mencionados alguns problemas com o planejamento das histórias, que o autor associa principalmente à complexidade técnica do sistema implementado.

Bunse et al. [12] apresentam uma disciplina que teve duração de um mês cujo objetivo era descobrir se estudantes conseguem desenvolver um projeto de XP com sucesso e como treiná-los para isso. Durante a disciplina, alunos de mestrado entraram em contato com as práticas de XP e desenvolveram um pequeno projeto fictício em três iterações. Os alunos formavam a equipe de desenvolvimento, que consistia de oito programadores, dois clientes e um *tracker*. Os instrutores atuavam como supervisores da equipe, intervindo apenas em casos graves como problemas de infra-estrutura e equívocos com relação a alguma prática de XP. Não foi usado o papel de *coach*. A equipe aplicou com sucesso as principais práticas de projeto, porém teve dificuldades com algumas das práticas de código. Foram implementados poucos testes de unidade e os testes de aceitação eram manuais. Além disso, os autores relatam problemas no uso de *design* incremental, que eles relacionam a pouca refatoração e um foco em implementar novas funcionalidades. Eles também observam a baixa usabilidade e manutenibilidade do *software* final, que eles atribuem à falta de disciplina e experiência dos desenvolvedores.

Del Bianco e Sassaroli [8] relatam uma tentativa de conciliar aulas sobre conceitos de engenharia de *software* tradicional com projetos usando práticas de XP. Para disso, adaptaram XP para que os conceitos aprendidos pudessem ser aplicados e cobrados. Os autores apresentam problemas e soluções encontrados após diversas edições do mesmo curso. Durante a disciplina, os alunos desenvolvem um só projeto, e os grupos são formados de maneira a obter heterogeneidade de conhecimento. Em cada grupo, o aluno com menos experiência recebe o papel de treinador junto com o aluno com mais experiência, que se responsabiliza pelos aspectos técnicos. Além disso, as reuniões dos grupos são realizadas em um tempo pré-determinado e o resto da sala pode observar a reunião, participar das discussões e fazer sugestões. Os autores também relatam uma forte resistência dos alunos com relação à escrita de testes automatizados, tanto antes como depois do código. Apesar da insistência dos instrutores no uso de TDD, os alunos declararam que não fizeram testes pois tinham pouco tempo disponível e testar consome muito tempo.

Em um curso de *Design* e Desenvolvimento de *Software* que tem como principal objetivo propiciar aos alunos experiência em desenvolver *software* em grupo, Wainer [38] adotou uma abordagem iterativa ao ensino. Ao invés de apresentar os tópicos um por um, apresentou alguns tópicos em conjunto e, em iterações seguintes, aprofundou o conhecimento dos tópicos já vistos ou introduziu novos conceitos. Ao longo do semestre, foram abordados tópicos que ajudariam a turma a resolver dificuldades que estavam enfrentando. Como projeto, foram usadas também duas iterações. Na primeira, tida como "projeto prática", os alunos foram divididos em grupos de 3 ou 4 e cada grupo desenvolveu um projeto fictício com o objetivo de se familiarizar com as técnicas e ambiente de programação. Na segunda, a turma se juntou para desenvolver um projeto com um cliente real, diferente do instrutor, com o objetivo de melhorar o envolvimento dos alunos e sua sensação de que a experiência vale no "mundo real". Em ambos os casos, as equipes não tinham *coach* mas eram supervisadas tanto pelo instrutor quanto pelo assistente da matéria. Wainer não relata problemas com as práticas de código, mas observa que a falta de tempo na disciplina resulta em menos iterações, menos refatoração e menos programação em pares. Ele considera que uma possível forma de resolver isso seria apresentar conceitos mais básicos como testes e refatoração mais cedo no currículo, desta

forma conseguindo alunos mais avançados ao iniciar um curso como esse.

Em uma abordagem similar a esta, Fenwick Jr. [27] descreve uma disciplina de engenharia de *software* sem tempo em laboratório que usou três projetos para ensinar XP de forma iterativa. O primeiro projeto era um projeto fictício bem simples e foi feito em dupla, com programação em pares, testes antes e uma ferramenta para controle de versões. O segundo projeto foi realizado em grupos de quatro alunos. Quase todas as práticas restantes de XP foram utilizadas, inclusive ciclos curtos, planejamento, estimativas e um pouco de refatoração. Finalmente o terceiro projeto foi feito em grupos de oito, usando as mesmas práticas de antes em um projeto maior. Nesse último caso, não houve tempo suficiente para realizar iterações curtas. Como os grupos mudam a cada projeto, a terceira fase acabou sendo uma única iteração. O autor considera que o uso de 3 projetos diferentes na disciplina ajudou os alunos a manterem a simplicidade no código e ajudou a evitar problemas que poderiam surgir ao juntar dois grupos. Porém, ele considera que isso também não permitiu aos alunos experimentarem por si mesmos os benefícios a longo prazo de escrever testes automatizados.

Mugridge et al. [30] escrevem sobre sua experiência em três instâncias de um curso de XP para alunos menos experientes, falando sobre os principais desafios encontrados nessa situação. Ao longo dessas experiências, a organização do curso foi evoluindo de forma a incorporar soluções para problemas de instâncias anteriores. Um dos desafios observados é que, se o projeto começa antes que os alunos aprendam conceitos de controle de versões, testes e refatoração, então essas práticas não funcionam bem. Assim, no terceiro oferecimento da disciplina o projeto só é iniciado depois que os alunos tiveram aulas e exercícios práticos sobre esses temas. Outro desafio interessante foi a dificuldade de fazer com que os alunos seguissem as práticas de fato. No primeiro oferecimento do curso, os alunos se preocuparam mais com detalhes do projeto e novas funcionalidades do que com qualidade de código e as práticas de XP. Os autores procuraram corrigir isso posteriormente alterando os critérios de avaliação do curso para dar mais ênfase às características desejadas.

Já McKinney et al. [29] descrevem o uso de algumas práticas de XP num curso introdutório de programação. Tal disciplina consiste de três aulas de 50 minutos e um laboratório de 75 minutos por semana. Com o objetivo de propiciar aos alunos experiência com trabalho em grupo desde cedo, foi decidido adotar uma adaptação de XP para desenvolver um projeto no laboratório. A instrutora atuava como cliente e *coach* de todas as equipes, que implementaram o mesmo projeto fictício. A maioria das práticas foi adotada em algum grau. Histórias, ciclos curtos e programação em pares foram usadas com sucesso, porém os testes tanto de aceitação quanto de unidade eram feitos manualmente. *Design* incremental também foi um problema, simplesmente pela inexperiência dos alunos com programação. A *coach* conseguia lembrá-los de manter o sistema simples e não inventar funcionalidades, mas a turma não tinha conhecimento o suficiente para apreciar uma prática como refatoração.

Becker-Pechau et al. [7] relatam um curso de verão sobre orientação a objetos com duração de 3 semanas, das quais uma é voltada a uma imersão de tempo integral em laboratório. O curso é oferecido para alunos que acabaram de completar o primeiro ano de faculdade e é organizado da seguinte forma: na primeira semana são apresentados os conceitos e ferramentas que serão usados no projeto; na segunda semana os alunos desenvolvem um projeto fictício usando XP; e na terceira semana os alunos apresentam os resultados e refletem sobre o aprendizado. Durante a semana de projeto, cada equipe é acompanhada em tempo integral por um ou dois instrutores, que atuam como *coaches* e como clientes da equipe. Os autores documentam suas observações sobre o uso e as difi-

culdades de cada prática adotada. Em particular, eles destacam a importância do acompanhamento do instrutor em garantir que as práticas são seguidas corretamente.

Hedin et al. [23] chegam a uma conclusão semelhante sobre a importância do papel do *coach*. Eles descrevem sua experiência usando XP em dois cursos paralelos: um curso de Programação em Equipe, obrigatório para aproximadamente 100 alunos do segundo ano de faculdade; e um curso de *Coaching*, optativo para em torno de 25 alunos dos terceiro e quarto anos de faculdade que tenham cursado o primeiro. No primeiro curso, o objetivo é ensinar conceitos básicos de engenharia de *software* usando XP. No segundo curso o objetivo é que os alunos aprofundem seu conhecimento de XP, desenvolvam habilidades em treinar e liderar uma equipe e pratiquem ideias de arquitetura de *software* em pequena escala. As equipes são formadas por 8 a 10 alunos do curso básico, 2 alunos do curso avançado atuando como *coaches* e um instrutor atuando como cliente. As equipes trabalham num mesmo ambiente, com uma dedicação fixa semanal e um projeto fictício mas verossímil. Os autores relatam os resultados de uso de diversas práticas de XP, bem como descrevem práticas que consideram valiosas para o sucesso de ambos os cursos. Eles destacam a importância do papel dos *coaches* em garantir que as práticas são seguidas e em promover discussões produtivas que proporcionam aprendizado em suas equipes. Eles também reforçam o fato de que ter dois *coaches* por equipe é uma boa maneira de obter redundância neste papel, já que ele é tão crucial para a equipe.

Dubinsky e Hazzan [18] relatam sua experiência em 5 oferecimentos de uma disciplina baseada em projetos que usa XP, para alunos no último ano de faculdade. Eles defendem uma abordagem diferente do uso de *coaches* mais experientes para garantir que as práticas são seguidas. Essa abordagem consiste em atribuir papéis especiais para cada aluno, além do papel de desenvolvedor. Por exemplo, um aluno pode ser "responsável pelos testes de unidade" enquanto o outro faz papel de "usuário final" e outro é "responsável por apresentações". Cada aluno é responsável por guiar e apoiar o resto da equipe nas tarefas relacionadas ao seu papel, bem como medir seu progresso. De acordo com os autores, essa abordagem aumenta o envolvimento pessoal de cada aluno com o projeto. Eles também descrevem um conjunto de papéis que consideram ideal e como escolher os papéis de forma a garantir que as práticas desejadas são seguidas.

Spacco e Pugh [35] descrevem sua experiência em ensinar alunos a escreverem testes e tentar motivá-los a escrever os testes antes da implementação em diversos cursos do currículo de computação não necessariamente ligados a engenharia de software. Eles relatam o uso de uma ferramenta chamada Marmoset para submissão e avaliação de exercícios programa desenvolvidos pelos alunos, assim como para a coleta e análise de dados sobre os projetos e testes submetidos. O sistema contém uma série de recursos pensados para ajudar o aluno a obter *feedback* sobre a tarefa que está tentando resolver, ajudar o instrutor a saber quais os pontos de dificuldade que a turma está tendo com o exercício e ajudar o monitor a ter mais informações sobre o projeto do aluno quando este vier pedir ajuda. Os autores descrevem esses recursos e relatam maneiras que buscaram para estimular mais os alunos a escrever seus próprios testes antes ou enquanto resolvem o exercício e revisam os resultados obtidos. Eles também relatam as vantagens e limitações de usar informações sobre a cobertura de código na avaliação da qualidade dos testes desenvolvidos pelos alunos.

Wellington [39] relata a introdução de XP e suas práticas em diferentes cursos da graduação em ciências da computação. Em primeiro lugar ele aborda a inserção de XP em cursos mais avançados sobre engenharia de *software*. Depois o autor relata a experiência de inserir algumas práticas de XP

em cursos mais básicos. Em particular, ele relata sucesso de aplicar programação em pares quando são seguidas as recomendações de trocar os papéis e trocar de duplas frequentemente. Ele também descreve tentativas de incorporar TDD de maneira incremental em um curso básico de programação. O primeiro passo é fornecer uma bateria de testes prontos para que os alunos "resolvam", isto é, façam os testes passar. Dessa forma os alunos podem focar em resolver os exercícios de programação apresentados. Mais para frente, é fornecida apenas uma descrição dos exercícios e pede-se que os alunos desenvolvam os próprios testes com a ajuda de alguns *plug-ins* para a ferramenta Eclipse[1] que simplificam a tarefa (por exemplo, adicionando dependências ao projeto automaticamente).

Acima pudemos constatar que existem vários trabalhos relacionados ao ensino de XP em diversos contextos. Entretanto, apesar de alguns apresentarem sugestões de como melhorar o aprendizado de XP, as mesmas ficam de alguma forma restritas aos ambientes propostos, ou ao menos a um ambiente de um curso. A seguir, apresentaremos algumas técnicas com potencial de reforçar o ensino de práticas ágeis em contextos mais amplos.

[1] http://eclipse.org. Último acesso em Fevereiro 2011.

Capítulo 3

Dojo de programação

Um *Dojo* de programação, ou simplesmente *Dojo*, é um encontro periódico – ou pelo menos frequente – no qual um grupo de programadores se reúne para aprender, praticar e compartilhar experiências em programação. A técnica se baseia em alguns princípios relacionados a métodos ágeis, listados abaixo.

- **Disposição para aprender** – Numa sessão de *Dojo*, o participante deve estar disposto a aprender junto com o grupo. Para isso, algumas atitudes são incentivadas [1], como repetir diversas vezes o mesmo exercício, usar abordagens diferentes das já conhecidas, cometer erros e aprender com eles, seguir um ritmo mais lento que o normal, explicar um conceito diversas vezes até que todos entendam, entre outras.

- **Participação ativa** – O *Dojo* de programação proporciona o aprendizado através da prática. O programador não deve ir a uma sessão apenas para observar como os outros trabalham. Ele deve também se envolver, tanto para programar quanto para discutir e questionar os assuntos levantados ao longo da sessão.

- **Ambiente seguro, colaborativo e inclusivo** – Uma sessão de *Dojo* deve ter um ambiente onde todos os presentes sintam-se incentivados a participar, com dúvidas, sugestões, soluções, etc. Receio de errar e competitividade são fatores que, em geral, inibem participantes e prejudicam suas oportunidades de aprendizado. Por isso, a atmosfera do Dojo deve mostrar que o grupo de pessoas colabora para resolver os problemas e recebe recém-chegados de qualquer nível.

- **Discussão com base concreta** – Partindo da premissa de que as pessoas aprendem melhor através de exemplos concretos do que de abstrações, no *Dojo* os participantes são encorajados a basearem todas as discussões em torno do código. Acredita-se que **o código é o design** [32] e que **código sem testes não existe** [4, pp. 43–45].

Em seu formato padrão, uma sessão de *Dojo* de programação requer um computador e um projetor, montados de forma que os participantes se sentem de frente para a projeção e quem estiver usando o computador se sente de frente para o resto dos participantes. Se uma lousa para rascunhos estiver disponível também, ela enriquece bastante as discussões sobre o problema, o design e a implementação. A Figura 3.1 exemplifica a montagem padrão de uma sessão de *Dojo*.

Figura 3.1: *Exemplo de montagem de uma sessão de Dojo de programação no IME-USP.*

O **objetivo** de uma sessão, em geral, é trabalhar em um exercício de programação simples e específico, como o *Kata*[1] do Boliche[2] ou o *Kata* do Karate Chop[3]. Os participantes são encorajados a programar a solução usando TDD e podem escolher livremente a linguagem de programação a ser usada para tal. Os dois formatos de sessão mais conhecidos são o *Kata* preparado e o *Randori*, descritos a seguir.

No ***Kata* preparado**, uma pessoa ou dupla já resolveu o problema proposto e prepara uma apresentação para mostrar sua solução ao resto dos participantes. Porém, ao invés de simplesmente navegar pelo código e testes prontos explicando como eles funcionam, o apresentador implementa a solução a partir do zero, mostrando sua evolução ao resto do grupo. A plateia deve ser capaz de acompanhar cada passo da solução, podendo interromper para fazer perguntas a qualquer momento. O objetivo da sessão é que, ao final, todos consigam reproduzir as etapas apresentadas para resolver o mesmo problema.

No ***Randori***, o grupo resolve o desafio proposto de forma colaborativa, usando TDD e programação em pares. Para isso, pares da plateia se revezam no uso do computador em turnos de tempo fixo (normalmente entre 5 e 7 minutos). Ao final de cada turno, o piloto volta para a plateia, o co-piloto vira piloto e alguém da plateia passa para co-piloto. Uma regra adicional do *Randori* é que, enquanto houver algum teste falhando ("no vermelho"), a plateia não interfere com o par que está programando. Apenas quando eles fizerem o teste passar (ficar "verde"), as pessoas assistindo podem sugerir refatorações, novos testes, etc. Essa regra existe para que o par tenha uma chance de se concentrar e treinar sem interferência externa.

Na próxima seção, falaremos sobre o surgimento e propagação do *Dojo* de programação pelo mundo. Em seguida, apresentaremos a história e processo do *Dojo* realizado no IME-USP. Por fim, será discutida a relação do *Dojo* com aprendizado de algumas práticas de código de XP.

[1] Ou exercício. O termo será explicado na Seção 3.1
[2] http://codingdojo.org/cgi-bin/wiki.pl?KataBowling. Último acesso em Abril 2010.
[3] http://codekata.pragprog.com/2007/01/kata_two_karate.html. Último acesso em Abril 2010.

3.1 *Dojo* de programação no mundo

Inspirado na ideia de exercícios das artes marciais, que são chamados de *Kata*, Dave Thomas[4] propôs a ideia de um *Kata* de programação [36, 37]. Sua motivação é que desenvolvedores treinam programação em seu trabalho e por isso cometem erros em seu trabalho. Um *Kata* de programação é um exercício que o programador pode resolver escrevendo código para ser jogado fora, sem compromissos. Treinando dessa maneira com frequência, o desenvolvedor cometeria menos erros em seu trabalho.

Thomas propõe vários exercícios que podem ser usados como *Kata* e enumera algumas condições que fazem uma boa sessão de prática:

> What makes a good practice session? You need time without interruptions, and a simple thing you want to try. You need to try it as many times as it takes, and be comfortable making mistakes. You need to look for feedback each time so you can work to improve. There needs to be no pressure: this is why it is hard to practice in a project environment. It helps to keep it fun: make small steps forward when you can. Finally, you'll recognize a good practice session because you'll come out of it knowing more than when you went in.

A partir do conceito de *Kata* de programação, Laurent Bossavit[5] e outros propuseram a ideia do *Dojo* de programação [9, 10]. Desde janeiro de 2005, o grupo de *Dojo* de programação em Paris se reúne semanalmente para praticar. Além disso, eles divulgaram o formato de reunião mundialmente, de forma que hoje existem *Dojos* de programação em diversas cidades do mundo (ver Figura 3.2).

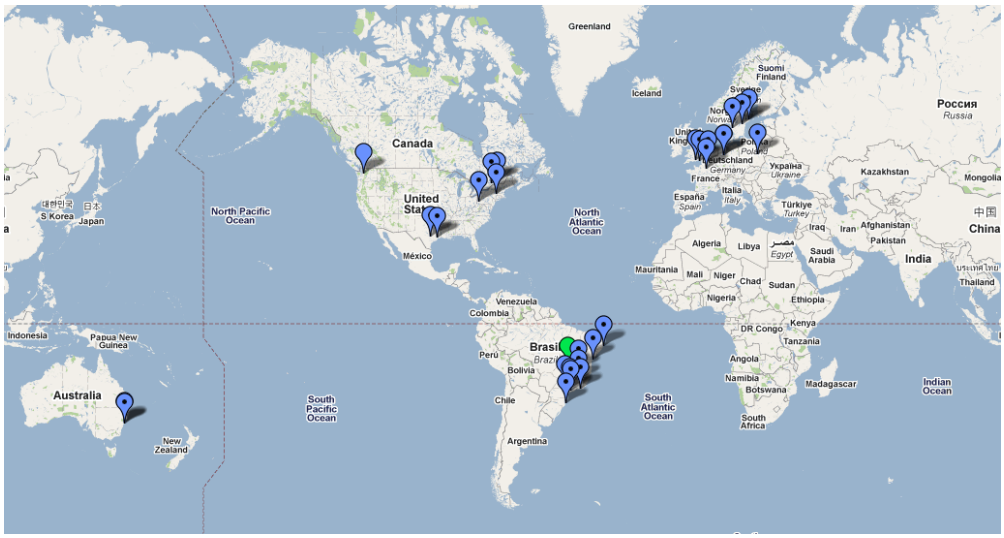


Figura 3.2: *Localização dos Dojos de programação listados em http://codingdojo.org/cgi-bin/wiki.pl?CodingDojos. (Último acesso em Novembro 2010).*

Hoje em dia, o *Dojo* de programação é uma técnica de aprendizado e treino reconhecida pela comunidade de métodos ágeis.

[4] Co-autor do livro *The Pragmatic Programmer: From Journeyman to Master*.

[5] Recipiente do Gordon Pask Award 2006 (http://www.paskaward.org/) e membro do Board of Directors da Agile Alliance (http://www.agilealliance.org/).

3.2 *Dojo* de programação em São Paulo

As reuniões de *Dojo* de programação em São Paulo[6] começaram em 12 de julho de 2007 e aconteceram semanalmente por mais de dois anos no Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP). Foram quase 100 reuniões, com o número de participantes variando entre 3 e 25. Desde meados de 2009, reuniões de *Dojo* de programação também tem sido organizadas fora do IME-USP, por empresas ou grupos de estudo interessados em praticar programação.

Danilo Sato et al. [33] relataram o formato e as lições aprendidas com o *Dojo* de programação no IME-USP. O formato geral de uma sessão do *Dojo* em São Paulo é o seguinte:

- **Escolha do problema** (5 a 10 minutos)

  Para começar a sessão, entre 3 e 5 problemas são apresentados brevemente, acompanhados ou não de uma linguagem na qual eles devem ser resolvidos. Os problemas são escolhidos de diversas fontes na Internet (como RubyQuiz[7], UVa[8] e SPOJ[9]) e após a apresentação os participantes votam em qual será resolvido nesta sessão.

- **Discussão do problema** (10 a 20 minutos)

  Uma vez determinado o problema, o grupo discute com mais detalhes sua definição e possíveis abordagens para resolvê-lo. Normalmente, faz-se uma lista de "coisas a fazer" para resolver o problema, como proposto por Kent Beck [5].

- **Programação** (1 a 2 horas)

  Após escolher uma abordagem para a solução, inicia-se a implementação, seja no formato *Randori* ou *Kata* preparado. O grupo segue **programação em pares** e **desenvolvimento dirigido por testes** como regra geral. Nem sempre a solução é implementada até o fim, mas o mais importante é que todos os presentes entendam e acompanhem o que acontece e que suas dúvidas sejam resolvidas.

  Em algumas reuniões, o grupo também usa controle de versão para fazer ***commits* frequentes** da evolução do código ao longo da sessão. A cada ciclo do TDD ou a cada turno, um novo *commit* é feito. Depois, essa evolução é publicada *online*[10].

- **Retrospectiva** (10 a 20 minutos)

  Para terminar a sessão, o grupo realiza uma retrospectiva para refletir sobre maneiras de melhorar o *Dojo* em si e sobre técnicas aprendidas ou conhecimentos adquiridos ao longo da sessão.

Atualmente, não são mais realizados *Dojos* no IME-USP, porém o grupo de São Paulo continua ativo e encontros são organizados com uma certa frequência em outros lugares, apesar de alguns períodos de inatividade.

[6] http://groups.google.com/groups/dojo_sp. Último acesso em Novembro 2010.
[7] http://www.rubyquiz.com. Último acesso em Agosto 2010.
[8] http://uva.onlinejudge.org/. Último acesso em Agosto 2010.
[9] http://www.spoj.pl, http://br.spoj.pl. Último acesso em Agosto 2010.
[10] https://github.com/dojosp/participant-s-projects. Último acesso em Fevereiro 2011.

3.3 *Dojo* de programação e aprendizado de práticas ágeis

Em áreas como xadrez e música, é reconhecido que um profissional precisa de muita prática para atingir um nível alto de perícia. Mais que isso, defende-se que apenas experiência não é fator suficiente para justificar as diferenças de habilidade desses profissionais. Existem estudos que procuram caracterizar o tipo de prática necessária para que haja evolução no desempenho. Essa evolução no desempenho deve ser mensurável objetivamente, ou seja, é preciso definir tarefas representativas do domínio estudado de forma que o indivíduo mais capacitado possa exibir um melhor desempenho de maneira consistente e reprodutível. Dessa forma, cria-se um ambiente de estudo controlado onde é possível investigar vários aspectos da aquisição de perícia no domínio.

Em seu trabalho, Ericsson [19] defende que a melhoria consistente e gradual no desempenho de tais tarefas ocorre sob as seguintes condições, que ele define como ***deliberate practice***:

> First, the participants were instructed to improve some aspect of performance for a well-defined task. Second, they were able to get detailed immediate feedback on their performance. Finally, they had ample opportunities to improve their performance gradually by performing the same or similar tasks repeatedly. [...] Engaging in practice activities with the primary goal of improving some aspect of performance is an integral part of deliberate practice.

O autor afirma que, a partir da revisão de estudos a respeito de aprendizado e aquisição de habilidades, foi possível encontrar evidências de que as condições citadas acima tem uma forte relação com melhoria no desempenho. A partir disso, ele estuda como seria possível propiciar *deliberate practice* (ou treino deliberado) em outras áreas de conhecimento, mais especificamente na medicina.

Apesar das semalhanças, achamos interessante ressaltar que no *Dojo* não existe um objetivo específico a ser atingido com cada tarefa. Por outro lado, existe um mecanismo complexo de relações interpessoais que traz benefícios indiretos ao aprendizado. Logo os efeitos possíveis do aprendizado no *Dojo* precisa ser estudado de uma maneira mais aprofundada.

Capítulo 4

Ferramentas para *feedback* automático

Em uma equipe tendo contato pela primeira vez com práticas ágeis como programação em pares e integração contínua, é possível que os membros da equipe esqueçam de realizar as práticas ou façam-as de maneira inapropriada. Por exemplo, se o membro da equipe não tem o costume de escrever nem executar testes, ele pode facilmente esquecer dessa necessidade por longos períodos de tempo. Mesmo que a equipe tenha um *coach* para ajudar na adoção das práticas, é difícil que ele consiga acompanhar e ajudar todas as duplas ou membros enquanto eles trabalham. Neste contexto, pode ser interessante dar aos membros da equipe algum *feedback* automático que lhes permita aprender sobre as práticas enquanto fazem elas.

No exemplo citado acima, para a prática de **código e testes**, uma ferramenta de *feedback* automático poderia observar a frequência de execução dos testes e, caso eles deixassem de ser executados por um longo período, lembrar o usuário de executar os testes. Supomos que um lembrete desse tipo teria dois potenciais efeitos positivos. O primeiro seria o simples e direto fato de o usuário rodar os testes com mais frequência, percebendo possíveis erros e confiando nos testes para encontrá-los. O segundo é mais indireto, mas poderia seguir-se que por rodar os testes com mais frequência o usuário fizesse mais testes ou fizesse testes mais relevantes, dessa forma melhorando seu aprendizado sobre testes automatizados.

Com o objetivo de procurar determinar se tal *feedback* automático teria efeitos positivos sobre o aprendizado, desenvolvemos uma ferramenta com lembretes para 3 práticas ágeis. Na Seção 4.1, descreveremos as funcionalidades e características presentes na ferramenta desenvolvida. Na Seção 4.2, descreveremos a plataforma Eclipse, que foi escolhida como base para o desenvolvimento da ferramenta. Por fim, na Seção 4.3, descreveremos a arquitetura e implementação do projeto. No Capítulo 6, apresentaremos a metodologia usada para avaliar os efeitos da ferramenta e os resultados obtidos.

4.1 Funcionalidades

De forma geral, a ferramenta de *feedback* de uma dada prática observa alguma ação relacionada a essa prática no ambiente de trabalho do usuário. Se o usuário ficar um determinado tempo sem realizar essa ação, a ferramenta gera um lembrete para que a ação seja realizada. Isso implica que a ferramenta deve ser capaz de detectar de alguma forma o acontecimento da ação, deve acompanhar o intervalo de tempo desde a última ocorrência da ação e deve gerar uma notificação ao usuário caso esse intervalo ultrapasse o tempo determinado.

É possível pensar em oferecer *feedback* automático para diversas práticas de código. Em particular, escolhemos as práticas de **programação em pares**, **código e testes** e **integração contínua**. A ferramenta de *feedback* implementada para cada prática é descrita a seguir:

1. **Programação em pares**: Lembrar a dupla de trocar o piloto periodicamente. Tentar perceber que houve uma troca de piloto automaticamente fugiria do escopo deste trabalho. Por isso, colocamos um botão na interface do usuário que permite à dupla informar ao programa quando eles realizaram uma troca.

2. **Código e testes**: Lembrar a dupla de executar os testes se eles ficarem mais de um determinado intervalo de tempo sem fazer isso. A execução de testes deveria ser detectada automaticamente.

3. **Integração contínua**: Lembrar a dupla de fazer *commits* se eles ficarem mais de um determinado intervalo de tempo sem fazer isso. Os *commits* deveriam ser detectados de forma automática através da observação dos arquivos sob controle de versão.

Além dos lembretes acima, seria interessante observar a atuação dos alunos em seus respectivos ambientes de trabalho para poder avaliar os resultados de forma objetiva. Por isso, desenvolvemos também uma ferramenta de acompanhamento que apenas observa as ações do usuário com relação às práticas escolhidas e grava-as em arquivos para análise posterior. Essa ferramenta observa:

- Os eventos de execução de testes;
- Os eventos de sincronização de projetos;
- Os eventos de troca de piloto através do botão disponível na interface;
- A aparição dos lembretes para o usuário.

Por fim, decidimos fornecer uma maneira de pausar os lembretes caso o usuário assim desejasse. Ao pausar os lembretes, as ferramentas de *feedback* ficam desativadas, ou seja, sem contar o tempo nem realizar notificações. Porém, a ferramenta de acompanhamento continua funcionando e grava também os eventos de início e pausa da ferramenta de lembretes.

Na Figura 4.1, é possível ver o lembrete para trocar de piloto em funcionamento. Também visível na barra de ferramentas estão os botões para pausar as notificações e para avisar a ferramenta que o piloto foi trocado.

A listagem a seguir resume cada uma das funcionalidades implementadas, agrupadas em dois sistemas, o de *feedback* e o de acompanhamento:

1. Sistema de *feedback*
   (a) Lembrar a dupla de trocar o piloto;
   (b) Permitir à dupla informar ao programa que eles realizaram uma troca;
   (c) Lembrar a dupla de executar os testes;
   (d) Detectar a execução de testes automaticamente;
   (e) Lembrar a dupla de fazer *commits*;

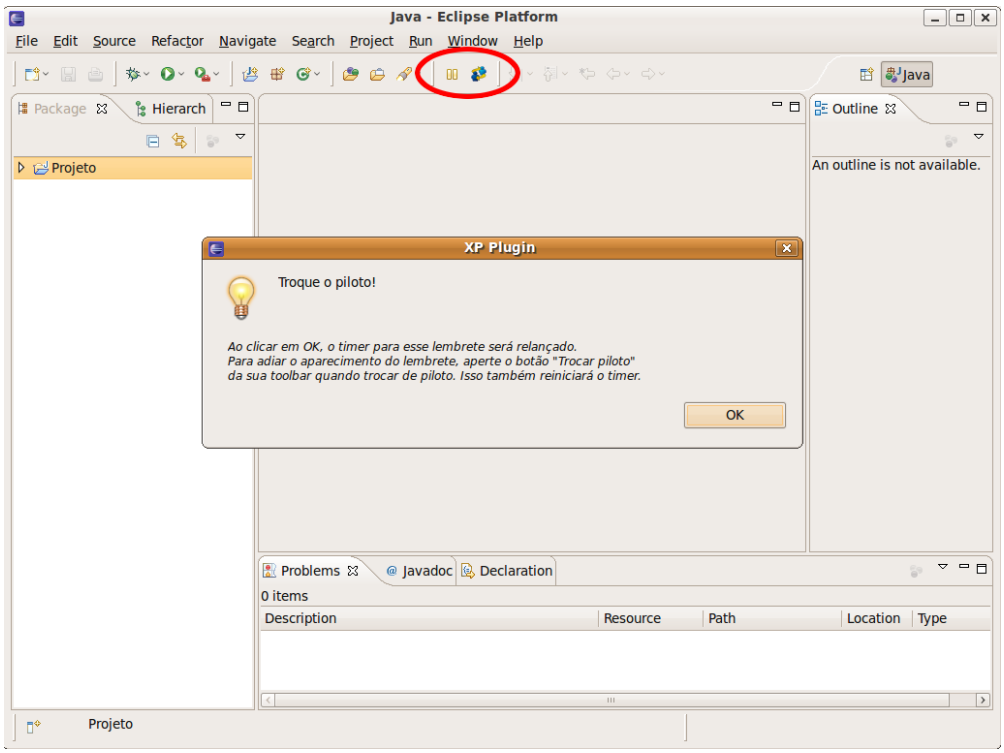


Figura 4.1: *Lembrete para trocar o piloto. Destaque na barra de ferramentas para os botões de pausar as notificações e sinalizar troca de piloto.*

(f) Detectar *commits* automaticamente;

(g) Permitir pausar todos os lembretes.

2. Sistema de acompanhamento

(a) Gravar os eventos de execução de testes;

(b) Gravar os eventos de sincronização de projetos;

(c) Gravar os eventos de troca de piloto;

(d) Gravar os eventos de aparição dos lembretes;

(e) Gravar os eventos de início e pausa da ferramenta de lembretes.

4.2 Ambiente de desenvolvimento

Para implementação das funcionalidades acima descritas, escolhemos o Eclipse[1] como ferramenta base. O Eclipse é um ambiente integrado de desenvolvimento (IDE) muito popular para a linguagem Java[2], desenvolvido por uma comunidade baseada no modelo de software livre. Por baixo da IDE Java existe uma plataforma genérica que dá apoio a uma grande variedade de ferramentas para diferentes linguagens e sistemas. Essa plataforma baseia-se num modelo de componentes altamente flexível que permite combinar e integrar as funcionalidades fornecidas pelos componentes conforme a necessidade [13, 28].

A unidade básica de função no Eclipse é um ***plug-in***, um módulo que descreve sua própria estrutura e suas dependências de outros *plug-ins*. Cada *plug-in* contribui para a aplicação de uma

[1] http://eclipse.org. Último acesso em Novembro 2010.
[2] http://java.sun.com/. Último acesso em Novembro 2010.

maneira estruturada, podendo tanto usar serviços providos por outros *plug-ins* como fornecer seus próprios serviços para que outros *plug-ins* aproveitem.

Na Figura 4.2, apresentamos uma visão geral dos componentes da plataforma Eclipse. Cada componente da figura pode corresponder a um ou mais *plug-ins*. Veremos a seguir a descrição dos principais componentes.

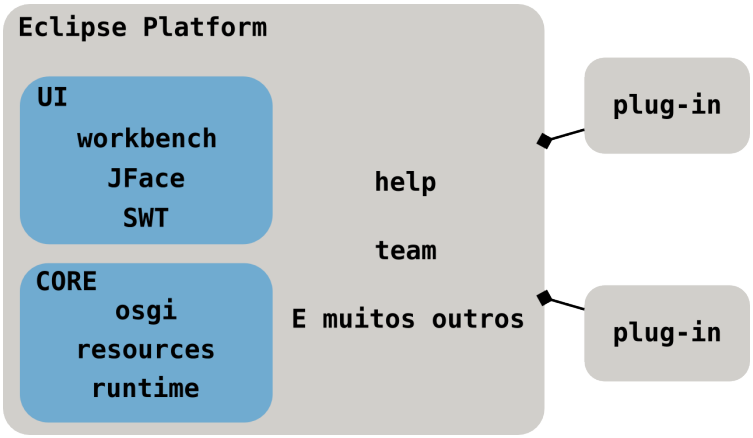


Figura 4.2: *Visão geral da plataforma Eclipse.*

O componente `core osgi` é uma implementação da especificação de mesmo nome[3] que descreve uma plataforma de serviços e módulos (ou "*bundles*") em Java [16]. Um "*bundle*" do OSGi corresponde a um *plug-in* do Eclipse. Esse componente do Eclipse é responsável pelo sistema de dependências entre os *plug-ins* e por carregar as classes conforme a necessidade.

O `core runtime` é o componente que executa o programa de fato. Ele carrega uma aplicação e mantém um registro dos *plug-ins* detectados, quais funcionalidades eles provém e seu estado atual. Com o objetivo de não penalizar o usuário em memória e desempenho por *plug-ins* que estão instalados mas não estão em uso, o `core runtime` em geral só carrega e executa as classes de um *plug-in* quando o usuário precisa delas.

O `core resources` define o modelo de recursos do Eclipse. Toda execução do Eclipse possui um `workspace`, a área de trabalho do usuário, que corresponde a uma pasta no sistema de arquivos. Ela contém projetos, que por sua vez podem conter pastas e arquivos. Os projetos não necessariamente correspondem a pastas dentro da área de trabalho, mas no `core resources` eles sempre correspondem a alguma pasta no sistema de arquivos local.

A interface gráfica do Eclipse é baseada nos componentes `SWT`, `JFace` e `workbench`. O `SWT`, ou *Standard Widget Toolkit*, é a biblioteca de componentes visuais que fazem a integração do Java com os elementos gráficos do sistema operacional. O `JFace` é uma série de classes desenhadas para facilitar o uso do `SWT`, provendo uma interface de programação de mais alto nível para realizar tarefas comuns em interfaces gráficas com o usuário. Ambas essas bibliotecas podem ser usadas independentemente da plataforma Eclipse. Por fim, o `workbench` define o modelo visual da aplicação, cuidando do ciclo de vida das janelas, *menus*, barras de ferramentas e assim por diante.

Além disso, a plataforma dispõe de vários outros componentes que cuidam de funcionalidades específicas, como o sistema de ajuda, o sistema de atualização, o sistema de equipe, etc.

A seguir, descreveremos com mais detalhes o que é um *plug-in*, sua estrutura e como ele contribui com funcionalidades na plataforma.

[3] http://osgi.org. Último acesso em Novembro 2010.

4.2.1 Definição e estrutura de um *plug-in*

Como vimos, um *plug-in* é a unidade básica de composição da plataforma Eclipse. Seu papel é prover funcionalidade à plataforma de maneira estruturada. Isso pode ser feito na forma de bibliotecas de código, extensões para a plataforma ou até documentação – um *plug-in* não precisa ter código. Um *plug-in* também pode definir pontos de extensão para que outros *plug-ins* possam incrementar ou completar sua funcionalidade.

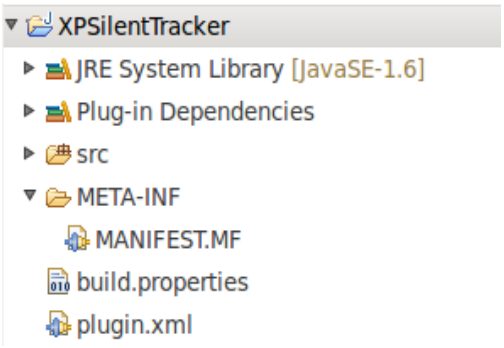


Figura 4.3: *A estrutura geral de um plug-in do Eclipse.*

Na Figura 4.3, vemos a estrutura de um dos *plug-ins* desenvolvidos para este trabalho. O único elemento obrigatório desta estrutura é o arquivo `META-INF/MANIFEST.MF`, também chamado de manifesto do *plug-in*. Esse arquivo faz parte da especificação OSGi e é responsável por descrever como o *plug-in* se relaciona com o *framework* e com os outros *plug-ins*. Na Figura 4.4, vemos um exemplo de manifesto de um dos *plug-ins* desenvolvidos neste trabalho, com destaque para as informações mais importantes deste arquivo, descritas a seguir:

- **Nome simbólico e versão**: o nome simbólico (por exemplo, `org.eclipse.ui`) e a versão (por exemplo, `3.6.1`), juntos, formam um identificador único de cada *plug-in* perante a plataforma;
- **Localização do código**: especifica o caminho onde se encontram as classes deste *plug-in*;
- **Dependências**: lista *plug-ins* ou pacotes dos quais este *plug-in* depende. Apenas as classes dos pacotes listados estarão disponíveis em tempo de execução;
- **Pacotes exportados**: lista os pacotes que são explicitamente exportados por este *plug-in*. Pacotes que não estiverem nessa lista nunca serão acessíveis para outros *plug-ins*. Pacotes que estiverem nessa lista serão acesíveis para *plug-ins* que dependam deste.

O manifesto do *plug-in* fornece informações que ajudam o *framework* a determinar quais classes devem estar disponíveis para cada *plug-in* em tempo de execução. Para que o *plug-in* possa interagir com a plataforma, em geral é preciso definir outro tipo de informação: em quais pontos e de qual forma isso deve acontecer. Para isso, existe o arquivo `plugin.xml`. Neste arquivo, cada *plug-in* pode definir **extensões** e **pontos de extensão** para a aplicação.

Ao definir um **ponto de extensão**, um *plug-in* documenta uma maneira pela qual outros *plug-ins* podem contribuir com seu funcionamento. Nessa definição, são especificados parâmetros obrigatórios ou opcionais para que a contribuição funcione. Na outra ponta, um *plug-in* declara uma **extensão** fornecendo as informações pedidas e dessa forma contribuindo com o primeiro *plug-in*.

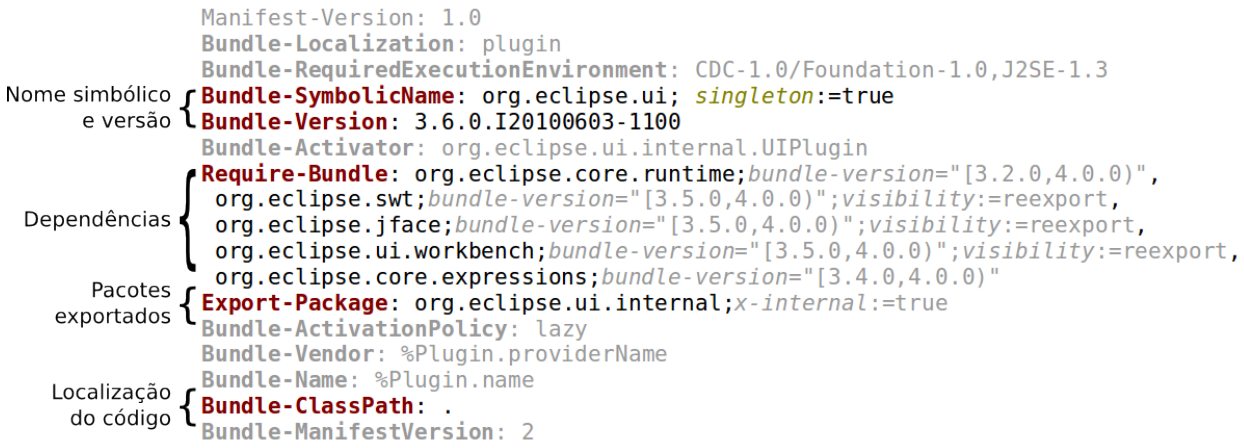


Figura 4.4: *Um exemplo de `MANIFEST.MF` com as seções descritas em destaque.*

Por exemplo, o plugin `org.eclipse.ui` define o ponto de extensão `org.eclipse.ui.editors` que permite contribuir um editor para o programa. Para tal, basta definir um identificador, um nome e prover uma classe que implementa a interface `IEditorPart`; podendo também fornecer um ícone para o editor, especificar extensões de arquivos com as quais ele lida, entre outros parâmetros opcionais. Na Figura 4.5 vemos o código `XML` para a declaração de um ponto de extensão. Nota-se que o `org.eclipse.ui` não depende de plugins que definem editores, mas sim o contrário. O `org.eclipse.ui` acessa as extensões através do `core runtime`, pedindo ao registro de extensões que forneça os parâmetros e instancie as classes necessárias para abrir um editor.

```
<schema targetNamespace="org.eclipse.ui" xmlns="http://www.w3.org/2001/XMLSchema">
   ...

   <element name="editor">
      <complexType>
         <attribute name="id" type="string" use="required" />
         <attribute name="name" type="string" use="required" />
         <attribute name="icon" type="string">
            <annotation>
               <appinfo>
                  <meta.attribute kind="resource"/>
               </appinfo>
            </annotation>
         </attribute>
         <attribute name="class" type="string">
            <annotation>
               <appinfo>
                  <meta.attribute kind="java" basedOn="org.eclipse.ui.part.EditorPart"/>
               </appinfo>
            </annotation>
         </attribute>
         ...
      </complexType>
   </element>

   ...
</schema>
```

Figura 4.5: *Um exemplo de ponto de extensão do `org.eclipse.ui.editors`.*

Para definir um editor para o Eclipse, basta então fornecer uma extensão para o `org.eclipse.ui.editors`. Na Figura 4.6, vemos o código `XML` que define o editor de código Java do próprio Eclipse. É importante notar que esse modelo de extensões promove um baixo acoplamento entre as classes de *plug-ins* diferentes, pois elas comunicam-se em geral apenas através de interfaces.

```
<plugin>
   ...
   <extension
         id="javaeditor"
         point="org.eclipse.ui.editors">
      <editor
            name="%CompilationUnitEditorName"
            default="true"
            icon="$nl$/icons/full/obj16/jcu_obj.gif"
            contributorClass="org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditorActionContributor"
            class="org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor"
            symbolicFontName="org.eclipse.jdt.ui.editors.textfont"
            id="org.eclipse.jdt.ui.CompilationUnitEditor">
            <contentTypeBinding
               contentTypeId="org.eclipse.jdt.core.javaSource"
            />
      </editor>
      ...
   </extension>
   ...
</plugin>
```

Figura 4.6: *Um exemplo de extensão do* `org.eclipse.ui.editors`.

4.3 Descrição do código

A implementação dos sistemas descritos na Seção 4.1 foi realizada em 7 plugins para o Eclipse[4]:

- **XPTrackerPlugin** oferece uma interface simples para os outros *plug-ins* gravarem um *log* na área de trabalho;
- **XPMainPlugin** cuida do ciclo de vida do sistema de notificação e oferece um ponto de extensão que recebe contribuições com diferentes lembretes para as diversas práticas;
- **XPPairProgrammingPlugin** estende o *plug-in* principal para oferecer lembretes para trocar de piloto;
- **XPTestsPlugin** estende o *plug-in* principal para oferecer lembretes para rodar os testes;
- **XPSVNListener** é um *plug-in* auxiliar que oferece funcionalidades para monitorar o estado de mudanças e *commits* na área de trabalho. Para simplificar o desenvolvimento da ferramenta, optamos por suportar apenas o Subversion[5] (ou SVN) como sistema de controle de versões, apesar da integração com outros sistemas ser possível;
- **XPTeamPlugin** estende o *plug-in* principal para oferecer lembretes de fazer *commits*;
- **XPSilentTracker** é um *plug-in* para uso do sistema de acompanhamento sem o sistema de notificações. Ele monitora a execução de testes e a realização de *commits*.

Na Figura 4.7 mostramos as relações de dependência que existem entre os 7 *plug-ins*. Veremos a seguir, com alguns detalhes, como cada um deles funciona.

XPTrackerPlugin - O *plug-in* de acompanhamento

Este *plug-in* provê um serviço que permite gravar eventos na área de trabalho. Ele expõe duas classes: `TrackingPlugin` e `Tracker`. A primeira permite a criação de um `Tracker` através

[4] O código dos *plug-ins* pode ser obtido em http://www.ime.usp.br/~marivb/XPPlugin. Último acesso em Fevereiro 2011.

[5] http://subversion.tigris.org. Último acesso em Novembro 2010.

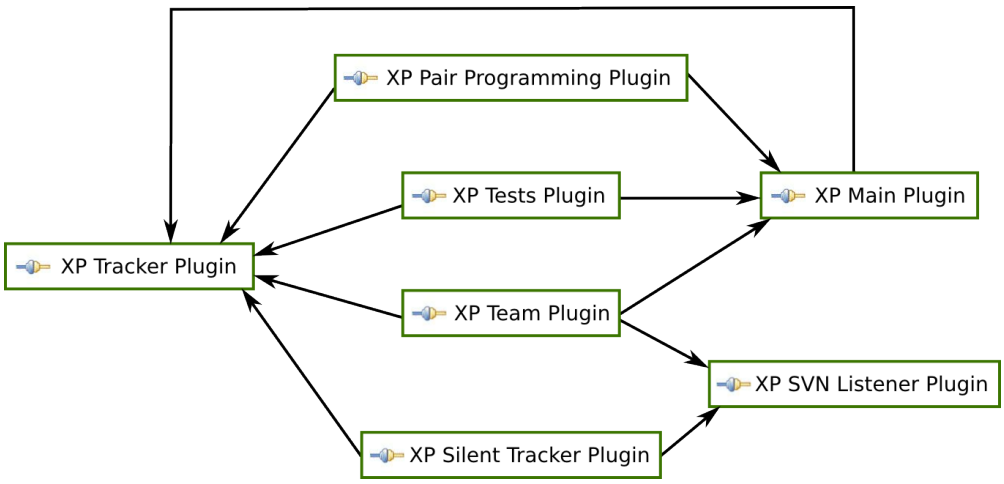


Figura 4.7: *Relações de dependência entre os plug-ins da ferramenta de feedback automático.*

do método `getTracker(String name)`. O parâmetro `name` corresponde a um nome para o conjunto de eventos que serão gravados com esse `Tracker`. O `Tracker` é uma interface que possui os seguintes métodos:

- `log(String message)` grava uma mensagem para todos os projetos abertos na área de trabalho;
- `log(String projectName, String message)` grava uma mensagem para um projeto específico na área de trabalho.

Atualmente, a única implementação da interface `Tracker` disponível é `WorkspacePractice-Tracker` que, para cada projeto, grava as mensagens em um arquivo texto dentro de uma pasta `.tracker` na raiz do projeto. O nome do arquivo é o nome do conjunto de eventos concatenado com um número representando o momento em que o `Tracker` foi criado. Todas as mensagens são gravadas acompanhadas do momento em que foram emitidas. Abaixo apresentamos como exemplo um trecho de um dos *logs* gravados pela ferramenta.

```
start at 12/05/2009 - 09:17
open dialog at 12/05/2009 - 09:54
open dialog at 12/05/2009 - 10:24
open dialog at 12/05/2009 - 10:54
open dialog at 12/05/2009 - 11:40
restart (not by timer) at 12/05/2009 - 11:47
stop at 12/05/2009 - 11:51
```

XPMainPlugin - O *plug-in* principal

O *plug-in* principal é o mais complexo, contendo várias extensões para o Eclipse e até uma definição de ponto de extensão própria. O propósito deste *plug-in* é simplificar a adição de novos tipos de lembretes ao sistema de notificação e centralizar funcionalidades comuns a todos os tipos de lembrete. Para isso, ele define o ponto de extensão `practiceMonitor` que espera como parâmetro

uma classe que implemente a interface `IPracticeMonitor`. Essa interface é bem simples, contendo apenas os métodos `start()` e `stop()` que informam ao monitor quando a ferramenta de lembretes está em execução e quando está pausada.

Para simplificar ainda mais, o *plug-in* também expõe a classe abstrata `PracticeMonitor`, uma implementação do `IPracticeMonitor` que usa o padrão *Template Method* [22, pp. 325–330] para realizar operações semelhantes a todos os monitores. Ao receber a mensagem `start()`, esse monitor lança um *timer* para esperar que um determinado intervalo de tempo passe. O intervalo em si é determinado pelas subclasses específicas. A partir do lançamento do *timer*, existem três resultados possíveis:

1. **Fim do intervalo**: caso o intervalo de tempo acabe sem interrupções, o monitor abre um diálogo *pop-up* que bloqueia a janela principal do Eclipse até que o usuário clique em *OK*. O conteúdo desse diálogo é dado pelas subclasses. Uma vez que o usuário clicou em *OK*, o *timer* é reinicializado;

2. **Reinício**: caso o monitor receba a mensagem `restart()`, que pode ser chamada por suas subclasses, o *timer* é reiniciado com o mesmo intervalo que estava rodando antes. Ou seja, isso adia o aparecimento do diálogo de lembrete;

3. **Pausa**: caso o monitor receba a mensagem `stop()`, o *timer* é cancelado.

Além de cuidar do funcionamento do *timer*, o `PracticeMonitor` também recebe um `Tracker` e grava nele cada um dos eventos descritos acima.

O *plug-in* principal também é responsável pelo ciclo de vida da ferramenta de lembretes através de duas contribuições para o Eclipse. A primeira é através do ponto de extensão `startUp`, que é usado para carregar um *plug-in* assim que o Eclipse for iniciado. Isso é necessário para carregar todos os monitores e dar início a seus *timers*, através do método `start()`. A outra contribuição é na forma de uma ação para a barra de ferramentas do Eclipse que permite ao usuário pausar ou retomar a execução dos monitores.

XPPairProgrammingPlugin - O *plug-in* de programação em pares

Este *plug-in* fornece um `practiceMonitor` para a troca de pilotos na programação em pares, o `PairProgrammingMonitor`. Ele também contribui uma ação para a barra de ferramentas do Eclipse que reinicia o monitor ao ser acionada.

XPTestsPlugin - O *plug-in* de testes

Além de fornecer um `practiceMonitor` para a execução de testes, este *plug-in* usa o *plug-in* do JUnit[6] para Eclipse a fim de observar a execução de testes na área de trabalho. A cada caso de teste terminado, o monitor reinicia o *timer*.

XPSVNListener - O *plug-in* auxiliar de SVN

Este *plug-in* define uma classe que permite observar mudanças no estado de sincronização dos projetos da área de trabalho. Ele trabalha em colaboração com o *plug-in* Subclipse[7] que faz inte-

[6] http://junit.org. Último acesso em Novembro 2010.
[7] http://subclipse.tigris.org. Último acesso em Novembro 2010.

gração do Subversion com o Eclipse. Essa classe mantém o estado atual de cada projeto que está sob controle de versão:

- ***"clean"***: sem mudanças para enviar ao repositório;
- ***"dirty"***: com mudanças para enviar ao repositório.

Conforme o estado é modificado, ela chama um ou mais métodos da classe `SVNStateHandler`:

- `projectAdded(IProject project, boolean configured)`: um projeto foi conectado a um repositório;
- `projectRemoved(IProject project)`: um projeto foi desconectado de seu repositório;
- `allProjectsCommited()`: todos os projetos estão "clean";
- `projectCleanedUp(IProject project)`: um projeto mudou de estado para "clean";
- `projectDirty(IProject project)`: um projeto mudou de estado para "dirty".

Por padrão, esses métodos simplesmente gravam cada evento em um `Tracker` com nome "SVN".

XPTeamPlugin - O *plug-in* de integração contínua

Este *plug-in* faz uso do `XPSVNListener` para observar quando os projetos mudam de estado na área de trabalho. Ele usa um `SVNStateHandler` que, além de gravar os eventos ocorridos como já foi descrito, reinicia o *timer* do monitor de práticas ao receber o evento `allProjectsCommited()`.

XPSilentTracker - O *plug-in* de acompanhamento sem notificações

Este *plug-in*, de maneira similar ao *plug-in* principal, usa o ponto de extensão `startUp` do Eclipse para ser iniciado assim que o programa carregar. Com isso, ele instala observadores para a execução de testes e para mudanças de estado dos projetos na área de trabalho, gravando em `Trackers` todos os eventos ocorridos, mas sem lembrar o usuário de nenhuma prática.

Capítulo 5

Estudo sobre *Dojo* de programação

Como visto no Capítulo 3, algumas práticas ágeis são usadas frequentemente em sessões de *Dojo* de programação. Com o objetivo de conhecer a percepção dos participantes com relação a seu aprendizado dessas práticas, formulamos um questionário a ser preenchido por participantes de *Dojos*. No questionário, temos perguntas investigando o conhecimento prévio do respondente e sua percepção de aprendizado para cada prática. Além disto, perguntamos também em quantas sessões de *Dojo* o respondente participou de forma a saber a influência disso no aprendizado. Finalmente, também investigamos quais os motivos que levam as pessoas a participarem de *Dojos*.

O questionário foi disponibilizado *online*[1] e enviado para as principais listas de *Dojo* brasileiras. O texto completo do questionário pode ser encontrado no Apêndice A. Ao todo, obtivemos 91 respostas, 95% delas vindas de 21 cidades do Brasil (ver Figura 5.1).

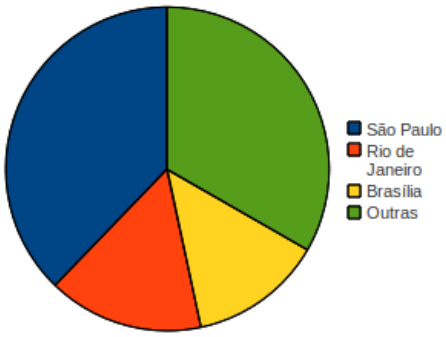


Figura 5.1: *Regiões de onde vieram respostas para o questionário.*

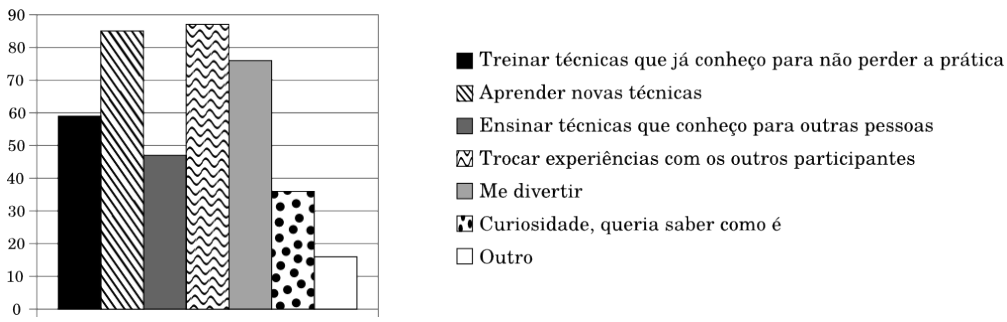


Figura 5.2: *Motivos para participar de sessões de Dojo de programação.*

[1] http://www.ime.usp.br/~marivb/DojoLearning/. Último acesso em Novembro 2010.

Na Figura 5.2 vemos os motivos que levam os respondentes a participar do *Dojo*. Mais de 90% afirmaram que trocar experiências e aprender novas técnicas são fatores importantes para sua participação. Nota-se também que 52% citam como um dos motivos o ensino de técnicas para outros participantes. Essas respostas ajudam a caracterizar o *Dojo* como um ambiente de aprendizado e colaboração.

Tabela 5.1: *Frequência de uso das práticas ágeis em sessões de Dojo de programação.*

Prática	**Número de respostas**
Programação em pares	87
TDD	89
Commits frequentes	30
Retrospectiva	75
Refatoração	84

Na Tabela 5.1 vemos quantos participantes afirmaram que cada prática foi usada nas sessões de *Dojo* das quais ele participou. As práticas mais incomuns são retrospectiva e *commits* frequentes. Isso pode ser explicado pelo fato de que ambas as práticas foram iniciadas no *Dojo* do IME-USP e podem não ter se espalhado pelo resto dos *Dojos*. Além disso, *commits* frequentes é uma prática que foi usada apenas em uma época no *Dojo* do IME-USP e depois foi deixada de lado.

Para a análise da percepção de aprendizado em cada prática, desconsideramos as respostas nas quais a prática não aparece como sendo usada. Ou seja, para a prática de retrospectiva, por exemplo, apenas 75 das respostas são válidas. A percepção do aprendizado de cada prática deveria ser classifica em alguma das opções abaixo:

- **Atrapalhou** - participar do *Dojo* de Programação atrapalhou o meu aprendizado
- **Indiferente** - participar do *Dojo* de Programação não teve nenhum efeito no meu aprendizado
- **Interessou** - participar do *Dojo* de Programação serviu para despertar meu interesse na prática, mas aprendi mesmo em outro lugar
- **Pouco** - participar do *Dojo* de Programação ajudou meu aprendizado, mas pouco
- **Médio** - participar do *Dojo* de Programação ajudou moderadamente, tive outras fontes para aprender também
- **Muito** - aprendi praticamente só com a participação no *Dojo* de Programação

Na Figura 5.3 mostramos, para cada prática, qual foi a percepção do grau de aprendizado dos respondentes. Em todos os casos, mais de 50% afirmou que participar do *Dojo* ajudou "médio" ou "muito" em seu aprendizado. Porém, algumas diferenças entre as práticas são notáveis.

TDD é a prática que apresentou maior aprendizado no *Dojo*. Pouco menos de 8% consideraram que participar do *Dojo* foi indiferente ao aprendizado ou apenas despertou seu interesse. Por outro lado, a retrospectiva foi a prática que mais despertou indiferença dos respondentes – 16% afirmaram não ter aprendido nada sobre ela com o *Dojo*. Por fim, a prática de *commits* frequentes foi a que menos propiciou aprendizado aos participantes. Apenas 63% afirmaram terem aprendido "médio" ou "muito" sobre essa prática no *Dojo*.

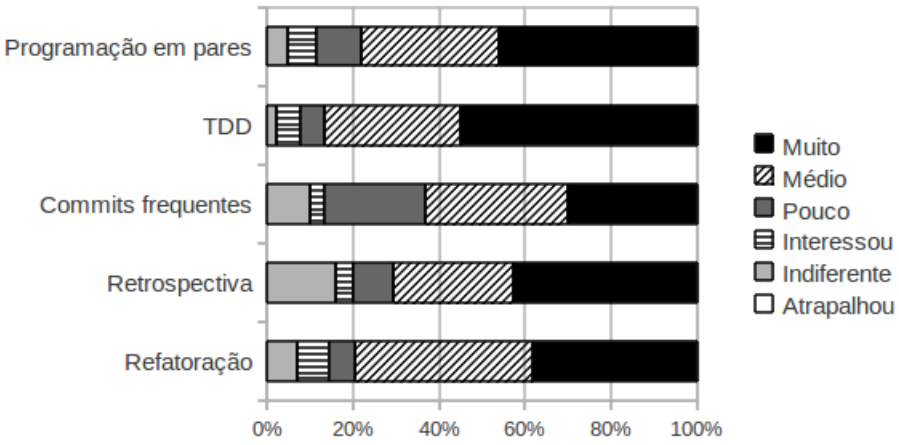


Figura 5.3: *Resumo da percepção de aprendizado para cada prática, em porcentagem de respostas.*

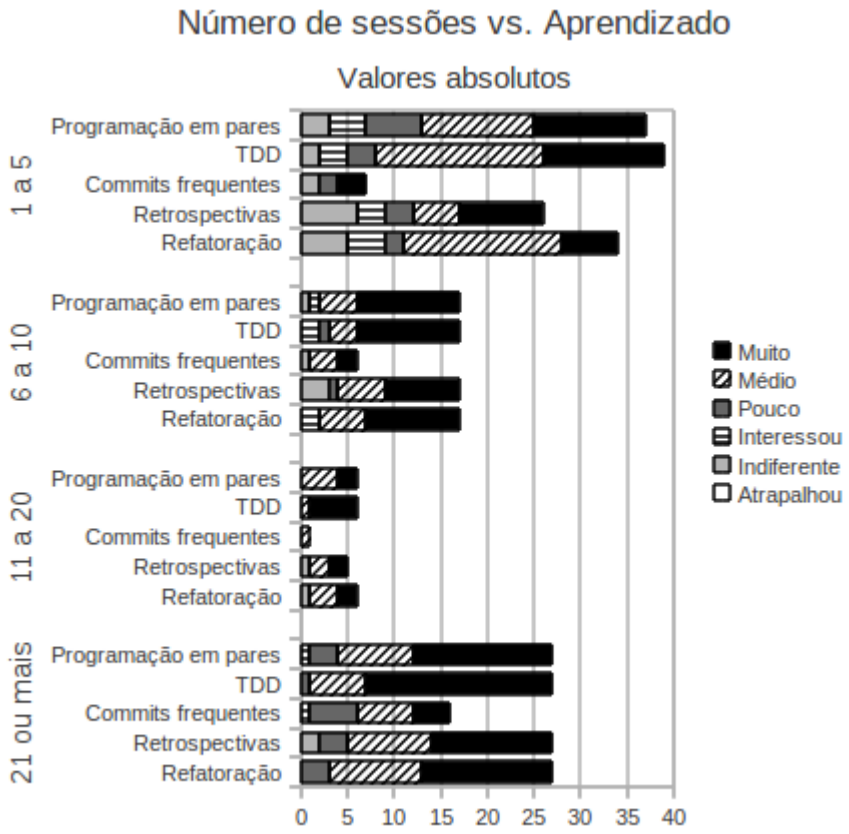


Figura 5.4: *A relação entre o número de sessões e a percepção de aprendizado, por prática, em número absoluto de respostas.*

Na Figura 5.4 apresentamos a relação entre o número de sessões das quais o respondente participou e sua percepção do aprendizado de cada prática. Desconsideramos as respostas de participantes que nunca foram a uma sessão de *Dojo*. Podemos ver nesse gráfico que a maioria dos respondentes que consideraram o *Dojo* indiferente para o aprendizado em alguma prática participou de apenas algumas sessões (1 a 5). Entre esse grupo também notamos, no geral, a menor porcentagem de respostas em que o aprendizado foi "muito". Essa tendência leva a supor que é preciso participar de mais sessões de *Dojo* para poder aprender as práticas utilizadas nele.

A terceira análise de interesse é a relação entre o conhecimento do respondente sobre uma prática antes de sua primeira sessão de *Dojo* e sua percepção de aprendizado desta prática (ver Figura 5.5). No questionário, o conhecimento prévio sobre cada prática poderia ser classificado em uma das seguintes opções:

- **Não conhecia** - Nunca tinha ouvido falar dessa prática
- **Teórico** - Já tinha lido a respeito em livros e na Internet
- **Iniciante** - Já tinha brincado e experimentado a prática em algumas ocasiões
- **Fluente** - Já usava a prática com frequência
- **Especialista** - Já dominava muito bem a prática, tendo mais de um ano de experiência com ela

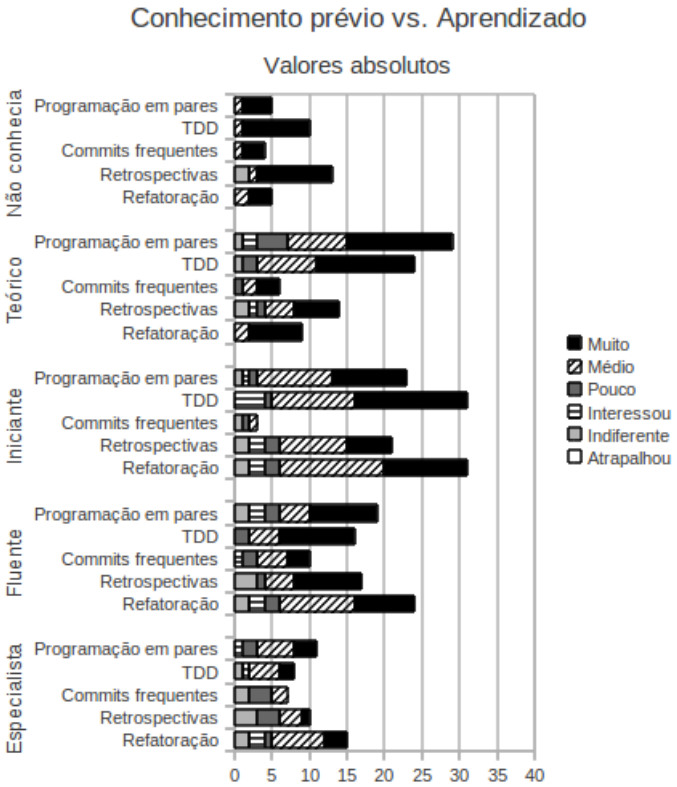


Figura 5.5: *A relação entre conhecimento prévio dos participantes e sua percepção de aprendizado, por prática, em número absoluto de respostas.*

As poucas respostas que apontam para indiferença no aprendizado aparecem espalhadas pelos níveis de conhecimento prévio, o que parece indicar uma baixa relação entre esses fatores. A maioria dos respondentes que não conheciam uma prática afirmou que aprendeu muito sobre ela – pelo menos 75% para todas as práticas, exceto refatoração. Apenas 60% dos que não conheciam refatoração disseram aprender muito sobre ela enquanto os outros 40% afirmaram aprender moderadamente sobre ela. No outro lado do intervalo, entre aqueles que afirmaram serem especialistas em uma prática, vemos que o *Dojo* propiciou menos aprendizado percebido: no máximo 27% afirmou ter aprendido muito e 50% afirmou ter aprendido moderadamente.

Capítulo 6

Avaliando as ferramentas de *feedback* automático

Veremos agora os resultados obtidos ao usarmos as ferramentas propostas no Capítulo 4. Para avaliar seu uso, escolhemos o curso de Programação Extrema do IME-USP, durante o primeiro semestre de 2009. Neste capítulo, descreveremos como é montado o curso, qual era o contexto no semestre escolhido, qual a metodologia que usamos para avaliar o uso da ferramenta e quais os resultados obtidos com isso.

6.1 Contexto e metodologia

O curso de Programação Extrema no IME-USP é formado por aulas em laboratório onde os alunos são divididos em grupos e cada grupo trabalha em um projeto diferente. A turma tem 2 semanas de aulas teóricas cobrindo as principais práticas de XP. Depois disso, os alunos começam a trabalhar em seus respectivos projetos em laboratório. A maioria dos grupos é formada por alunos com pouca ou nenhuma experiência com as práticas de XP. Cada grupo tem um *coach* que é escolhido entre os alunos mais experientes matriculados – de preferência que já tenha feito a matéria ou tenha tido contato com métodos ágeis.

A turma de 50 alunos inscritos no começo do semestre foi dividida em 7 grupos, dos quais 4 usariam Java como linguagem de programação e os outros 3 usariam outras linguagens. Devido à escolha do Eclipse como plataforma de desenvolvimento e integração das ferramentas, apenas os 4 grupos usando Java poderiam usar a ferramenta de *feedback* automático. Para facilitar a observação do experimento, os 4 grupos foram colocados numa mesma sala, enquanto os outros 3 grupos dividiam outra sala. Os grupos podem ser brevemente descritos da seguinte forma:

1. **Grupo 1**: 5 alunos, aplicação Java para *desktop*, base de código existente
2. **Grupo 2**: 8 alunos, aplicação Java para *web*, base de código existente
3. **Grupo 3**: 6 alunos, aplicação Java para *desktop*, começando do zero
4. **Grupo 4**: 10 alunos, aplicação Java para *web*, começando do zero

O experimento foi organizado em duas fases. Durante a primeira fase, os Grupos 1 e 2 usaram a ferramenta de *feedback* automático mas os Grupos 3 e 4 não usaram. O objetivo dessa fase era

verificar o efeito do uso da ferramenta em comparação com o efeito de não usar a ferramenta. Ao final dessa fase, todos os grupos responderam um questionário sobre as práticas (ver a Seção B.1) e os Grupos 1 e 2 responderam um questionário sobre o uso da ferramenta (ver a Seção B.2).

Durante a segunda fase, todos os grupos usaram a ferramenta, porém com diferentes intervalos de tolerância para os lembretes das práticas. Na Tabela 6.1, apresentamos os intervalos usados durante a primeira fase e por cada grupo durante a segunda fase. O objetivo dessa fase era tentar identificar o intervalo de tolerância mais apropriado para cada prática. Ao final dessa fase, todos os grupos responderam um questionário a respeito dos intervalos (ver a Seção B.3).

Tabela 6.1: *Intervalos de tolerância para os lembretes usados durante a primeira fase e por cada grupo durante a segunda fase.*

Lembrete para...	**Trocar o piloto**	**Fazer *commit***	**Executar os testes**
Primeira fase	30 min.	30 min.	10 min.
Grupo 1	40 min.	60 min.	5 min.
Grupo 2	10 min.	100 min.	55 min.
Grupo 3	55 min.	10 min.	30 min.
Grupo 4	25 min.	30 min.	10 min.

Além disso, de forma a avaliar os resultados mais objetivamente, todos os grupos usaram a ferramenta de acompanhamento das ações do usuário descrita no Capítulo 4 ao longo do semestre. Como vimos, essa ferramenta acompanhava os seguintes eventos:

- A frequência de execução de testes;
- A frequência de *commits* de um projeto inteiro;
- A frequência de troca de piloto através do botão disponível na interface;
- A aparição dos lembretes para o usuário.

Vale observar que os dois primeiros itens puderam ser acompanhados para todos os grupos, porém os dois últimos itens dependiam da presença da ferramenta de *feedback* automático. Por isso, eles não foram acompanhados para os Grupos 3 e 4 durante a primeira fase.

6.2 Resultados

Veremos a seguir os resultados obtidos durante a avaliação dos efeitos do uso da ferramenta e dos intervalos de tempo. A análise foi feita com base em dois conjuntos de dados: um conjunto objetivo, fornecido pelos *logs* gravados pela ferramenta enquanto ela era usada; e um conjunto subjetivo, consistente das respostas a questionários fornecidas pelos alunos ao final de cada fase.

Para simplificar a explicação dos resultados, vale à pena oferecer mais detalhes sobre como os *logs* foram gravados. Os arquivos de *log* foram separados de acordo com duas dimensões: as **categorias de eventos** e as **sessões de programação**. Ou seja, para cada categoria e cada sessão um arquivo diferente foi criado. Por esse motivo, não foi possível cruzar dados de diferentes categorias mas de uma mesma sessão. As categorias de *logs* que foram gravadas são as seguintes (no Capítulo 4 os eventos são explicados com mais detalhes):

- **Lembretes para troca de piloto** – os eventos de abertura de diálogo e reinício do *timer* (clique no botão de 'Trocar piloto') são gravados nesta categoria, assim como os eventos de início e pausa da ferramenta de lembretes;
- **Lembretes para execução de testes** – os eventos de abertura de diálogo e reinício do *timer* (execução de testes) são gravados nesta categoria, assim como os eventos de início e pausa da ferramenta de lembretes;
- **Eventos de execução de testes** – os eventos relacionados a execução de testes, bem como seus resultados, são gravados nesta categoria;
- **Lembretes para fazer *commit*** – os eventos de abertura de diálogo e reinício do *timer* (nenhum projeto da área de trabalho com mudanças) são gravados nesta categoria, assim como os eventos de início e pausa da ferramenta de lembretes;
- **Eventos de controle de versões** – os eventos relacionados ao estado de sincronização dos projetos da área de trabalho são gravados nesta categoria.

Além disso, para efeitos do *log*, uma sessão de programação corresponde ao intervalo de tempo entre o primeiro e o último evento gravado. No caso das categorias de lembretes, o primeiro evento gravado sempre corresponde ao momento em que o Eclipse foi aberto, mas no caso das categorias de eventos isso não é verdade. O primeiro evento gravado diz respeito aos eventos observados por aquela categoria – por exemplo, a primeira execução de testes. Além disso, o último evento gravado não corresponde ao momento em que o Eclipse fechou em nenhuma das categorias pois esse evento não foi observado.

A análise dos dados objetivos foi feita com base em diferentes relatórios gerados a partir desses dados:

- **Quantidade de ocorrências** de cada evento por sessão;
- **Duração estimada** de cada sessão a partir da diferença de tempo entre o primeiro e o último evento gravado. Não foi possível ter a duração exata pois não registramos o momento em que o Eclipse foi encerrado e em alguns casos também não registramos o momento em que ele foi iniciado;
- **Frequência** de cada evento por sessão, com base nas informações citadas acima;
- **Linha do tempo** dos eventos de cada sessão. Esse relatório nos permite observar não apenas a frequência ou número dos eventos por sessão mas também a ordem em que eles aconteceram, dessa forma descobrindo alguns padrões que não apareceriam com simples números;
- **Evolução** de todos relatórios listados acima ao longo do tempo para várias sessões ordenadas cronologicamente. Esse relatório nos permite descobrir se o uso da ferramenta causou alguma alteração no comportamento dos usuários.

Por fim, algumas sessões foram consideradas casos especiais e analisadas de maneira diferente ou removidas da análise. Em primeiro lugar, as sessões com duração estimada em zero são sessões onde apenas um evento aconteceu ou todos os eventos aconteceram no mesmo minuto. Elas foram

analisadas de maneira diferente pois dificultam, por exemplo, o cálculo da frequência dos eventos. Também tratamos de maneira especial as sessões nas quais não ocorreu nenhum evento de interesse – por exemplo, as sessões das categorias de lembretes nas quais ocorreram apenas eventos de início e pausa das notificações. Para essas sessões, é preciso considerar qual é o significado dessa falta de eventos importantes. Conforme as análises são apresentadas, explicaremos quantas sessões foram desconsideradas e por quais motivos.

6.2.1 Efeitos de usar a ferramenta (Fase 1)

Como descrevemos na Seção 6.1, o objetivo da primeira fase era comparar os efeitos do uso da ferramenta com grupos que não a usaram. Apresentaremos a seguir o que pudermos observar para cada prática.

Ao final da Fase 1, os alunos responderam dois questionários fornecendo dados subjetivos para análise. A estrutura geral de cada questionário será descrita a seguir, mas seu conteúdo completo pode ser encontrado nas Seções B.1 e B.2.

O primeiro questionário foi respondido por todos os grupos que participaram do experimento e tinha o objetivo de investigar diferenças na percepção dos alunos a respeito de cada prática. Com esse propósito, para cada prática foram feitas algumas perguntas semelhantes:

- **Intensidade ideal e atual da prática**

  Para essas perguntas, eram apresentados 5 cenários diferentes de uso da prática (enumerados de a a e) com intensidade decrescente. Isto é, a opção a descrevia o cenário de uso mais intenso da prática e a opção e descrevia o cenário no qual a prática era menos usada. Apresentadas essas opções, o respondente deveria apontar qual delas ele considerava ser o ideal para uma equipe XP e qual delas ele considerava ser o estado atual para a sua equipe.

- **Frequência subjetiva**

  Nessa pergunta, o respondente deveria informar a sua percepção da frequência do evento relacionado a esta prática por aula.

O segundo questionário foi respondido apenas pelos grupos que usaram a ferramenta e tinha como objetivo levantar as reações e impressões que os usuários tiveram com ela. Novamente, para cada prática foram feitas algumas perguntas semelhantes:

- **Ordem das reações**

  Para essa pergunta, primeiro era apresentada uma lista de reações possíveis à abertura do lembrete de uma dada prática. Em geral, as opções continham 'ignorar' e 'fazer o que o lembrete pede', entre outras. O respondente deveria, então, ordenar essas reações pela frequência com a qual elas aconteciam, da mais frequente para a menos frequente.

- **Aprendizado**

  Nessa pergunta, o respondente deveria escolher como ele considerou que a ferramenta de lembretes ajudou no aprendizado da prática em questão, dadas as opções: ajudou, foi indiferente e não ajudou.

Programação em pares

As informações vindas dos *logs* que temos a respeito de programação em pares são limitadas pois os *logs* puderam ser gravados apenas para os grupos que usaram as ferramentas. Foram gravadas ao todo 108 sessões, das quais 37 tinham apenas o evento de início (ou seja, duração zero) e 17 tinham apenas eventos de início e pausa (ou seja, nenhum evento de interesse). Essas sessões foram consideradas inválidas e não foram usadas na análise apresentada a seguir. Na Tabela 6.2 mostramos as informações sobre as sessões válidas e inválidas de cada grupo.

Tabela 6.2: *Dados das sessões válidas e inválidas dos Grupos 1 e 2 para a prática* ***programação em pares****.*

	Grupo 1	**Grupo 2**
Sessões válidas	46	8
Média de duração	1:41:00	2:07:14
Sessões inválidas	42	12
Média de duração	0:06:16	0:02:25

Lembramos que a forma de reiniciar o *timer* deste lembrete era clicar no botão 'Trocar piloto' disponível na interface, uma atitude manual que poderia ser esquecida facilmente. Isso significa que não conseguimos saber a relação direta entre os cliques e as trocas de piloto em si. Tanto a ocorrência de uma troca sem haver um clique de botão quanto a ocorrência de um clique no botão sem ter havido uma troca são eventos possíveis. Ao observar a frequência dos eventos gravados nessa prática, não é surpreendente constatar que o botão 'Trocar piloto' foi clicado em média $0,32$ vezes por hora enquanto que o lembrete para trocar de piloto aparaceu em média $1,56$ vezes por hora.

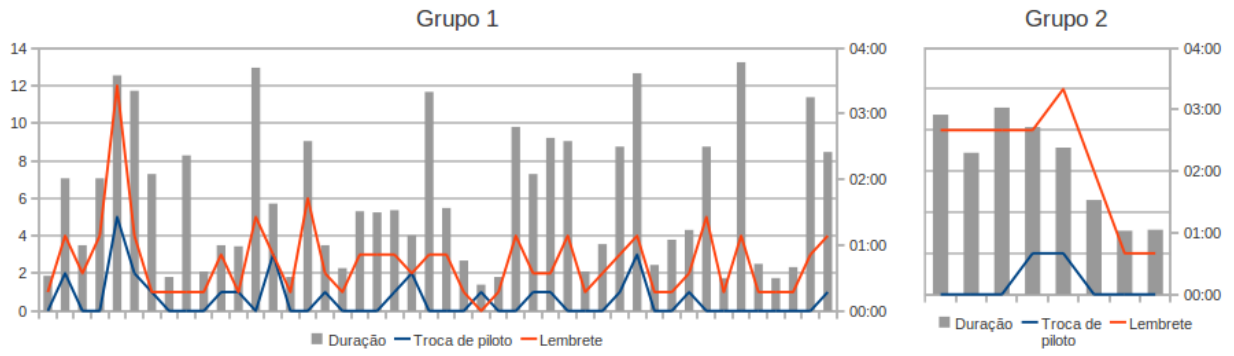


Figura 6.1: *Para cada sessão dos lembretes de trocar o piloto, a quantidade de cada evento e a duração estimada. As sessões aparecem em ordem cronológica, separadas por grupo.*

Além disso, ao observar a evolução das quantidades de ocorrências de cada evento em conjunto com a duração estimada das sessões (ver Figura 6.1), não se pode perceber nenhuma alteração de comportamento ao longo do tempo. Essa observação foi comprovada também pela análise da evolução das linhas do tempo. Isso indica simplesmente que os usuários não adquiriram um costume de clicar no botão 'Trocar piloto'.

Essa suposição é condizente com as respostas subjetivas para uma pergunta na parte sobre programação em pares no segundo questionário da Fase 1 (ver Figura 6.2). Quase 55% dos respondentes afirmaram nunca clicar no botão para indicar troca de piloto. Se considerarmos que a duração de uma aula é de aproximadamente 2 horas, podemos atribuir valores estimados para a frequência de cliques por hora das opções apresentadas nesta questão (ver Tabela 6.3). Com esses valores, é

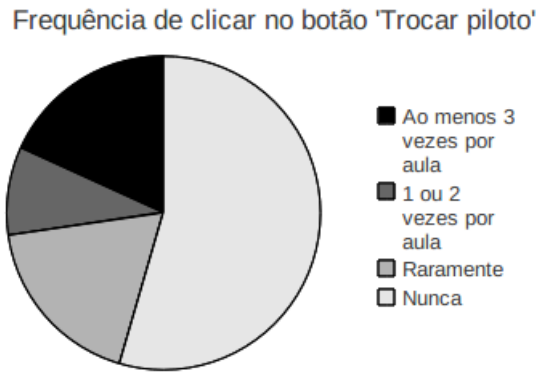


Figura 6.2: *Frequência de cliques por aula no botão 'Trocar piloto'.*

possível estimar que a média ponderada relatada pelos respondentes é de $0,39$ cliques por hora, um valor bem próximo dos $0,32$ cliques por hora apontados pela observação objetiva.

Tabela 6.3: *Relação das opções na questão sobre frequência de cliques no botão 'Trocar piloto' com as estimativas numéricas de frequência usadas para calcular a média.*

Ao menos 3 vezes por aula	$\Rightarrow$	$1,5$ cliques por hora
1 ou 2 vezes por aula	$\Rightarrow$	$0,75$ cliques por hora
Raramente (1 vez por semana ou menos)	$\Rightarrow$	$0,25$ cliques por hora
Qual botão 'Trocar piloto'?	$\Rightarrow$	0 cliques por hora

Os resultados vistos até aqui levam a concluir que os alunos não adquiriram o costume de clicar no botão 'Trocar piloto', de forma que na maioria das sessões o lembrete para trocar de piloto aparecia a cada 30 minutos. Quando questionados sobre a ordem de suas reações quando este aviso aparecia, 36% dos respondentes afirmaram que trocar de piloto era a reação mais comum e 55% deles afirmaram que ignorar o aviso era a reação mais comum. A Figura 6.3 mostra quantas vezes cada opção oferecida apareceu em cada posição. A única reação "outra" que apareceu nessas respostas foi "A gente terminava a ideia (se fosse rápida) e trocava".

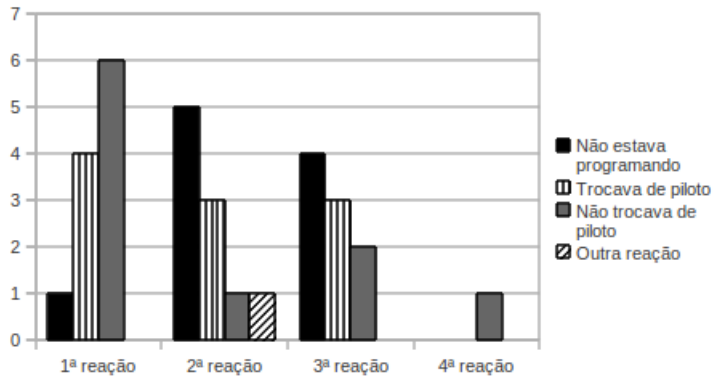
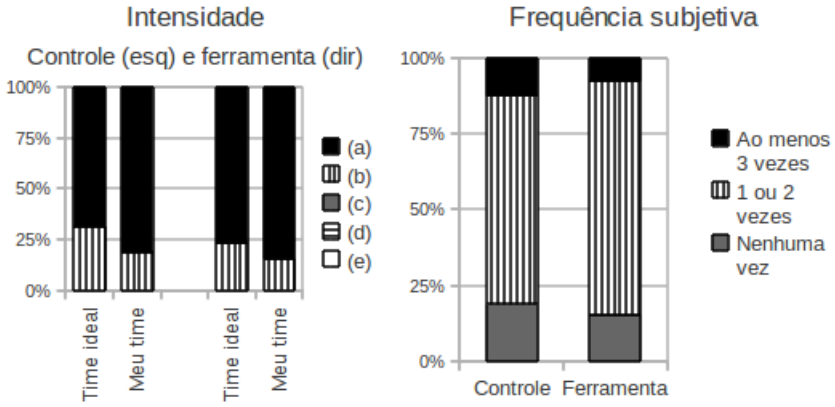


Figura 6.3: *Respostas para a ordem das reações ao lembrete de trocar de piloto. Cada colocação exibe quantas vezes cada resposta apareceu nela.*

Com relação à percepção dos alunos sobre quanto a ferramenta ajudou no aprendizado de programação em pares, 50% dos respondentes afirmaram que ela foi indiferente, $41,7\%$ afirmaram que ela ajudou e os outros $8,3\%$ afirmaram que ela atrapalhou.

Por fim, podemos comparar as percepções dos alunos do grupo de controle com os do grupo que usou a ferramenta de lembretes para a programação em pares. Na Figura 6.4 vemos a comparação entre as intensidades ideal e atual relatadas pelos alunos (esquerda) e a comparação entre as frequências subjetivas relatadas por eles (direita).



Esquerda: Intensidade ideal e atual da programação em pares, como relatada pelos alunos dos diferentes grupos.

Direita: Frequência da troca de pilotos por aula, como relatada pelos alunos dos diferentes grupos.

Figura 6.4: *Comparação entre as intensidades ideal e atual relatadas pelos alunos (esquerda) e a comparação entre as frequências subjetivas relatadas por eles para a* ***programação em pares****.*

Para a questão a respeito da intensidade da prática, apenas duas opções foram utilizadas:

(a) Nunca ou quase nunca fazemos *commits* de código que não foi feito em par. Nós fazemos o rodízio constante de pares.

(b) Nós geralmente trabalhamos em pares, mas de vez em quando fazemos *commits* de trabalho feito sozinho.

A diferença entre as porcentagens de cada resposta é de menos de 8% para o time ideal e menos de 4% para o estado atual, o que não representa uma diferença significativa. Similarmente, no lado direito da figura vemos que a frequência da troca de piloto considerada pelos respondentes é parecida entre os dois lados. Considerando novamente que 'uma aula' dura aproximadamente 2 horas, podemos atribuir valores de frequência para cada uma das possíveis respostas como na Tabela 6.4. Com isso, é possível estimar a média ponderada da frequência de troca de piloto, concluindo que os respondentes relataram $0,7$ trocas por hora no grupo de controle e $0,69$ trocas por hora no grupo que usou a ferramenta – ou seja, os números praticamente iguais indicam que ocorre uma troca a aproximadamente cada 1 hora e meia.

Tabela 6.4: *Relação das opções na questão sobre frequência subjetiva de trocas de piloto com as estimativas numéricas de frequência usadas para calcular a média.*

Trocamos de piloto pelo menos 3 vezes em uma aula	$\Rightarrow$	$1,5$ cliques por hora
Trocamos de piloto 1 ou 2 vezes em uma aula	$\Rightarrow$	$0,75$ cliques por hora
Em uma aula, o piloto não muda	$\Rightarrow$	0 cliques por hora

De forma geral, podemos resumir os resultados observados a respeito da prática **programação em pares** com as seguintes conclusões:

- O uso da ferramenta de lembretes não criou um costume de clicar no botão 'Trocar piloto', independente de acontecerem trocas de piloto ou não;
- A aparição dos lembretes era na maioria das vezes ignorada pelos usuários, porém também levou a uma troca de pilotos em várias ocasiões;
- Não há diferenças perceptíveis entre o grupo de controle e o grupo que usou a ferramenta no que diz respeito à frequência subjetiva de troca de piloto.

Integração contínua

A prática integração contínua tinha duas categorias de eventos associadas a ela: lembretes para fazer *commits*, que teve 99 sessões gravadas para 2 grupos; e eventos de controle de versões, que teve 272 sessões gravadas para os 4 grupos. Para a categoria de lembretes, as sessões com duração zero (28 ao todo) e as sessões com apenas eventos de início e pausa (20 ao todo) foram desconsideradas (ver Tabela 6.5). Para a categoria de eventos de controle de versões, não se gravou o momento de início da sessão portanto várias sessões onde apenas um evento ocorreu têm duração zero (156 ao todo, ver Tabela 6.6). Apresentamos alguns resultados a respeito destas sessões porém elas são analisadas separadamente das outras.

Além disso, houve 3 sessões do Grupo 3 que foram analisadas também de forma particular pois elas tinham 14, 23 e 58 horas de duração. Como os *plug-ins* foram disponibilizados para que os alunos pudessem baixá-los em seus computadores pessoais, supomos que essas 3 sessões correspondem ao uso prolongado do Eclipse em uma máquina que foi 'hibernada' ao longo de várias horas. Como não é possível saber a quantas horas de programação essas sessões correspondem, elas não puderam ser analisadas em conjunto com as outras.

Tabela 6.5: *Dados das sessões válidas e inválidas dos Grupos 1 e 2 para a categoria de lembretes de fazer commit.*

	Grupo 1	**Grupo 2**
Sessões válidas	43	8
Média de duração	1:43:26	1:58:44
Sessões inválidas	36	12
Média de duração	0:17:01	0:02:25

Tabela 6.6: *Dados das sessões com duração mais que zero e com duração zero dos 4 grupos para os eventos de controle de versões.*

	Grupo 1	**Grupo 2**	**Grupo 3**	**Grupo 4**
Duração maior que zero	19	7	54	30
Média de duração	1:04:56	0:34:25	1:51:47	3:26:44
Duração zero	62	13	34	50

No que diz respeito aos eventos relacionados ao lembrete de fazer *commit*, não podemos perceber nenhuma evolução do comportamento ao longo do tempo (ver Figura 6.9). Na média, as duplas fizeram *commit* de todos os projetos $0,23$ vezes por hora enquanto o lembrete para fazer *commit* apareceu $1,50$ vezes por hora. Esse número indica que as duplas faziam um *commit* a aproximadamente cada 4 horas de trabalho – ou, como no geral nenhum dia de trabalho corresponde a mais de 4 horas, eles faziam *commit* uma vez por aula.

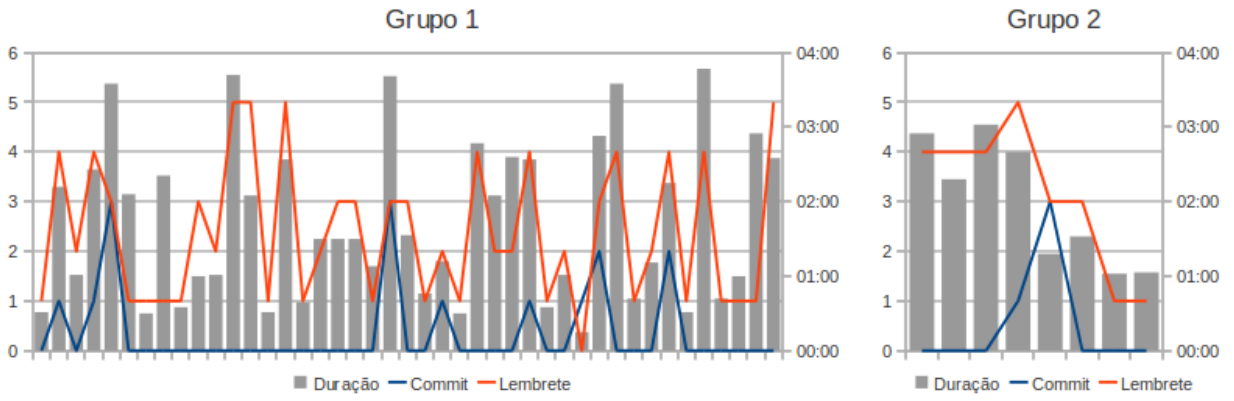


Figura 6.5: *Para cada sessão dos lembretes de fazer commit, a quantidade de cada evento e a duração estimada. As sessões aparecem em ordem cronológica, separadas por grupo.*

Ao observarmos as reações que os usuários tinham quando aparecia o lembrete de fazer *commit* (Figura 6.6), de fato percebemos que a principal reação era ignorar o lembrete e não fazer o *commit*. Além disso, três alternativas foram mencionadas para a posição de reação mais frequente: "Clicava em OK, terminava o que estava fazendo e fazia commit", "Faziamos o commit a cada passo importante na funcionalidade ou no testes" (essa alternativa não é na verdade uma possível reação ao lembrete, mas sim uma explicação de quando o *commit* era feito – aparentemente a pergunta não foi entendida corretamente) e "Não tinha o que comitar".

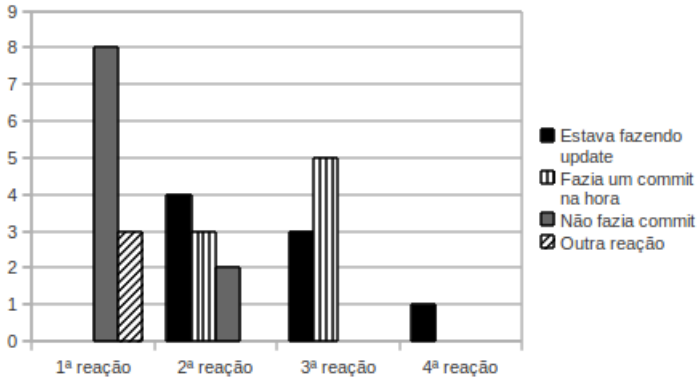


Figura 6.6: *Respostas para a ordem das reações ao lembrete de fazer commit. Cada colocação exibe quantas vezes cada resposta apareceu nela.*

Quanto à percepção que os usuários da ferramenta tiveram sobre a ajuda que ela teve no aprendizado, 64% deles disse que a ferramenta foi indiferente mas 27% afirmaram que ela atrapalhou. Apenas 9% considerou que a ferramenta ajudou no aprendizado de integração contínua.

Para continuar a análise do uso deste lembrete, vamos observar os dados comparativos com os grupos que não usaram a ferramenta.

Na análise dos dados objetivos, já vimos que algumas sessões tiveram duração maior que zero enquanto outras não. Vemos na Figura 6.7 a quantidade com que cada evento apareceu nas sessões que tiveram duração maior que zero. As outras sessões não foram inclusas na figura pois tornaria sua leitura difícil e elas tem em geral apenas um ou dois eventos (que ocorreram no mesmo minuto).

A diferença entre os eventos '*Commit* de tudo' e '*Commit* de projeto' é que alguns grupos tinham mais de um projeto em sua área de trabalho. Uma das restrições que colocamos na ferramenta de

lembretes é que ela apenas reiniciava seu *timer* quando **todos** os projetos da área de trabalho fossem *commitados*; por isso esses eventos foram diferenciados no *log*. Porém, ambos correspondem a eventos de *commit*. O evento 'Mudança' representa o momento em que uma alteração foi feita em algum projeto que estava previamente "*clean*".

Pelo gráfico percebemos uma diferença significativas entre as atividades de todos os grupos. O Grupo 1 teve no máximo 4 eventos de *commit* por sessão. O Grupo 2 teve apenas 7 sessões com duração maior que zero, das quais as duas primeiras foram intensas em atividades mas as outras não tiveram mais que 3 eventos. O Grupo 3 é o que parece ter maior intensidade de *commits* por sessão, tendo inclusive algumas delas que tiveram mais de 10 eventos de *commit*. Por fim o Grupo 4 apresentou uma intensidade de atividades de *commit* menor que o 3 mas ainda sim maior que o 1 e o 2.

Tabela 6.7

	***Commit* de tudo**	***Commit* de projeto**	**Mudança**
Grupo 1	0,20	0,95	0,18
Grupo 2	0,25	0,75	1,00
Grupo 3	1,34	2,10	1,59
Grupo 4	0,41	0,77	1,21

Para calcular a média de ocorrências de cada tipo de evento por grupo, consideramos também as sessões que tiveram duração zero. Ao invés de calcular a média de ocorrências por hora, calculamos a média de ocorrências por sessão uma vez que as estimativas de duração das sessões gravadas não é boa. Apresentamos essas médias na Tabela 6.7. Apesar de cada grupo ter várias sessões com duração zero, as médias seguem as mesmas tendências apresentadas acima.

Vamos ver a seguir os resultados do questionário a respeito das práticas respondido por todos os grupos. Na Figura 6.8 vemos a comparação entre as intensidades ideal e atual relatadas pelos alunos (esquerda) e a comparação entre as frequências subjetivas relatadas por eles (direita).

Nas questões sobre a intensidade da prática, chegaram a ser citadas 4 opções diferentes:

(a) Quando trabalho em uma funcionalidade, sincronizo e disponibilizo o código diversas vezes por dia.

(b) Quando trabalho em uma funcionalidade, sincronizo e disponibilizo o código uma vez por dia.

(c) Quando trabalho em uma funcionalidade, sincronizo e disponibilizo o código uma vez por semana.

(d) Algumas semanas podem se passar sem que eu disponibilize o código. Só disponibilizamos código quando a tarefa está pronta.

As diferenças entre as respostas de cada lado chega a ser significativa e reflete as diferenças que já analisamos nos dados objetivos. Dentre os usuários que não usaram a ferramenta, 94% afirmaram que o ideal é que o código seja sincronizado diversas vezes por dia, enquanto que entre os que usaram apenas 69% concordam com essa afirmação. Similarmente, quando indagados sobre o estado atual de sua equipe, 50% dos respondentes do grupo de controle afirmaram sincronizar diversas vezes por dia, em comparação com os 31% que deram a mesma resposta entre os respondentes que usaram a ferramenta.

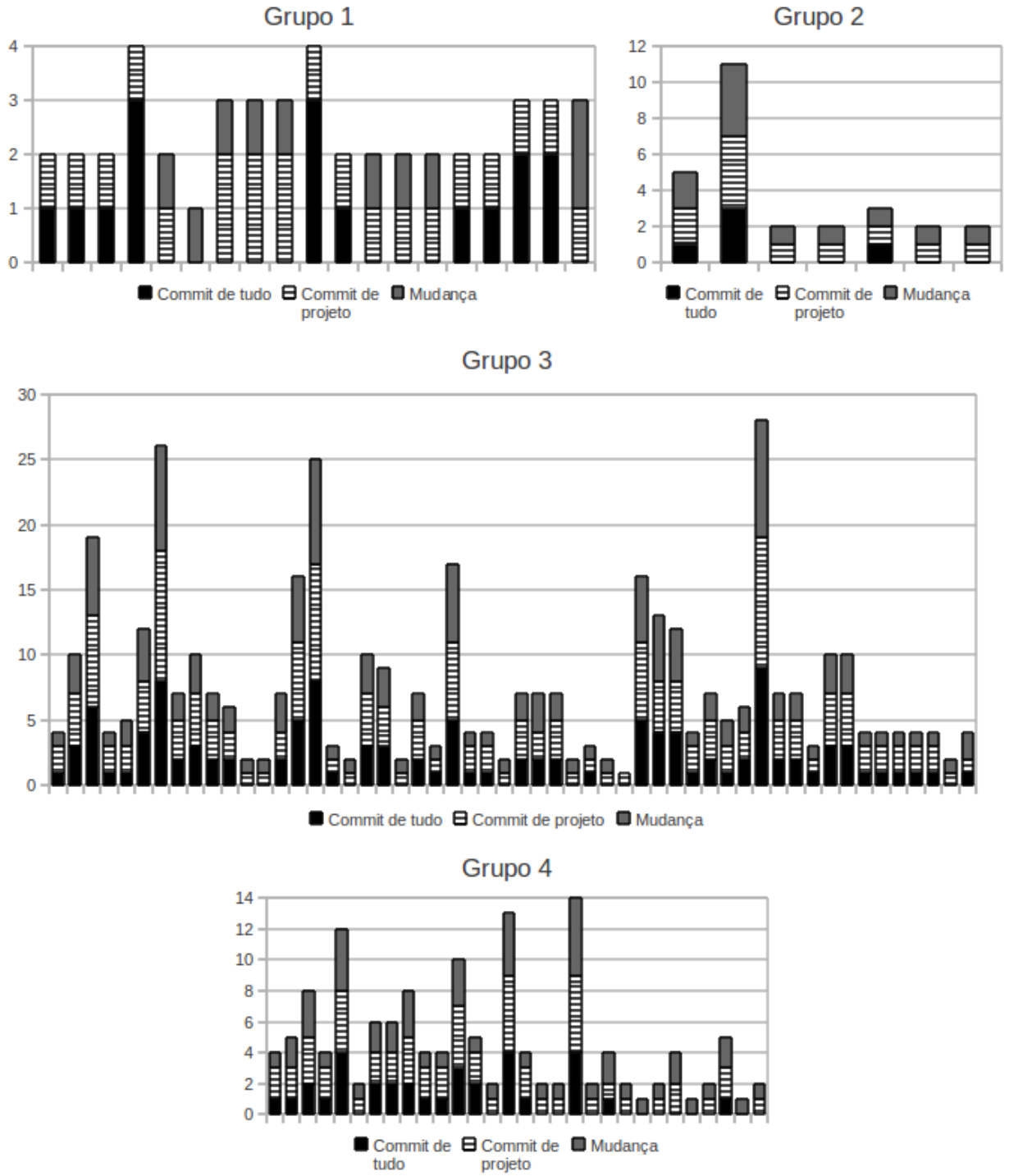
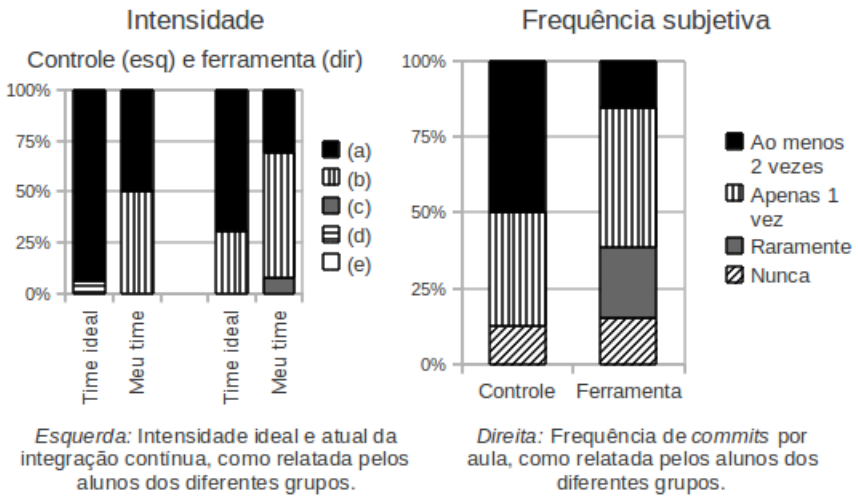


Figura 6.7: *Para cada sessão dos eventos de controle de versões, a quantidade que cada evento apareceu. As sessões aparecem em ordem cronológica, separadas por grupo.*



Esquerda: Intensidade ideal e atual da integração contínua, como relatada pelos alunos dos diferentes grupos.

Direita: Frequência de *commits* por aula, como relatada pelos alunos dos diferentes grupos.

Figura 6.8: *Comparação entre as intensidades ideal e atual relatadas pelos alunos (esquerda) e comparação entre as frequências subjetivas relatadas por eles para a* ***integração contínua****.*

Com relação à frequência com que os alunos relataram fazer *commit* de todos os projetos em sua área de trabalho, novamente vemos uma grande diferença entre os grupos que usaram e que não usaram a ferramenta. Em torno de 14% em ambos os lados afirmaram que existem arquivos sob controle de versão que nunca são enviados para o repositório. Por outro lado, 50% dos respondentes que não usaram a ferramenta afirmaram que fazem *commit* de tudo ao menos duas vezes por aula, comparados aos apenas 15% que disseram fazer isso entre os alunos que usaram a ferramenta.

No entanto, analisando os comentários feitos pelos alunos no próprio questionário, acreditamos que essas respostas não retratam a situação realmente percebida pelos alunos devido a um mau-entendido com relação a esta questão. Isso é porque 4 dos 13 alunos que usaram a ferramenta fizeram comentários do tipo "A pergunta 11 me pareceu estranha. Existem alguns trechos do código que eu nunca alterei, ou algumas partes específicas que raramente são alteradas.". Supomos que não ficou claro, na questão, que a pergunta se referia apenas a fazer *commit* de mudanças que estivessem pendentes, isto é, que caso um projeto ou partes dele não tivesse mudanças com relação ao repositório, isso não contaria como uma oportunidade perdida de fazer *commit*.

De forma geral, podemos resumir os resultados observados a respeito da prática **integração contínua** com as seguintes conclusões:

- A aparição dos lembretes para fazer *commit* era quase sempre ignorada pelos usuários;
- A frequência de *commits* por sessão variou entre os quatro grupos, com uma tendência para menos *commits* entre os grupos que usaram a ferramenta e mais *commits* entre os que não usaram;

Código e testes

Assim, como a integração contínua, a prática código e testes tem duas categorias de eventos: lembretes para executar os testes, com um total de 103 sessões gravadas para os grupos 1 e 2, e eventos de execução de testes, com um total de 106 sessões gravadas para os 4 grupos. Novamente para a categoria de lembretes, consideramos inválidas as sessões com duração zero (24 ao todo) e as sessões com apenas eventos de início e pausa (7 ao todo, ver Tabela 6.8). Para a categoria de

eventos de execução dos testes, analisamos todas as sessões porém destacamos na Tabela 6.9 as quantidades de sessões com duração zero e maior que zero, para cada grupo.

Tabela 6.8: *Dados das sessões válidas e inválidas dos Grupos 1 e 2 para a categoria de lembretes de executar os testes.*

	Grupo 1	**Grupo 2**
Sessões válidas	56	16
Média de duração	1:31:20	1:14:18
Sessões inválidas	27	4
Média de duração	0:02:11	0:00:00

Tabela 6.9: *Dados das sessões com duração mais que zero e com duração zero dos 4 grupos para os eventos de execução de testes.*

	Grupo 1	**Grupo 2**	**Grupo 3**	**Grupo 4**
Duração maior que zero	26	1	33	27
Média de duração	1:01:39	1:07:00	1:43:43	1:29:02
Duração zero	5	1	1	12

No que diz respeito aos eventos relacionados ao lembrete de fazer *commit*, não podemos perceber nenhuma evolução do comportamento ao longo do tempo (ver Figura 6.9), porém percebemos uma clara diferença entre os comportamentos dos grupos 1 e 2. Na média, as duplas do Grupo 1 executaram os testes $3,24$ vezes por hora e receberam o aviso $3,79$ vezes por hora, enquanto as duplas do Grupo 2 executaram os testes apenas $0,4$ vezes por hora e receberam o aviso $5,4$ vezes por hora. Vale lembrar que os avisos para executar os testes apareciam a cada 10 minutos, ou seja, 3 vezes mais que os avisos para trocar piloto ou fazer *commit*.

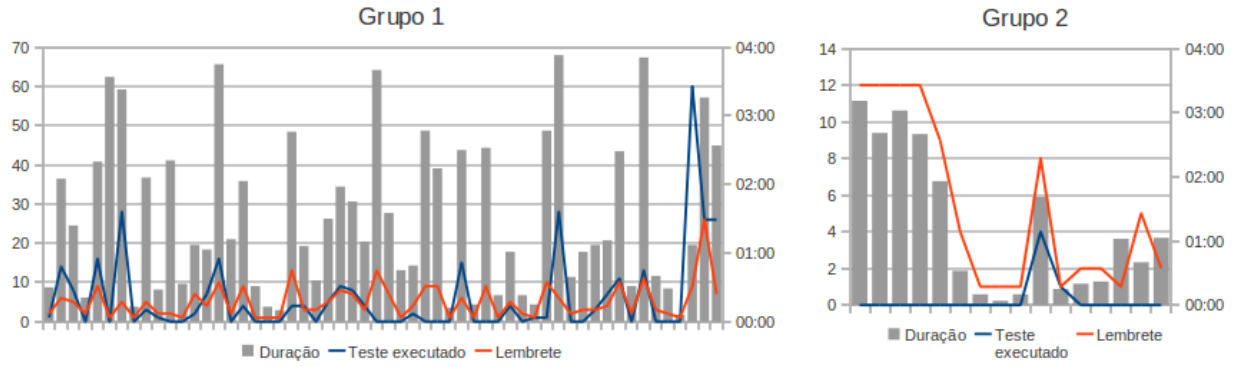


Figura 6.9: *Para cada sessão dos lembretes de executar os testes, a quantidade de cada evento e a duração estimada. As sessões aparecem em ordem cronológica, separadas por grupo.*

A reação que os usuários relataram ser a mais frequente ao receber o aviso de executar os testes foi de ignorar os avisos, correspondendo a 73% das reações em primeira opção (ver Figura 6.10). Além disso, dois respondentes mencionaram uma reação alternativa. "Não fazia sentido executar os testes, pois estávamos mexendo em algo relacionado à eles" foi mencionado como reação mais comum por um deles; e "Estávamos programando de fato e executávamos os testes independente do aviso" foi mencionado como segunda reação mais comum por outro, logo após a opção "não executava os testes".

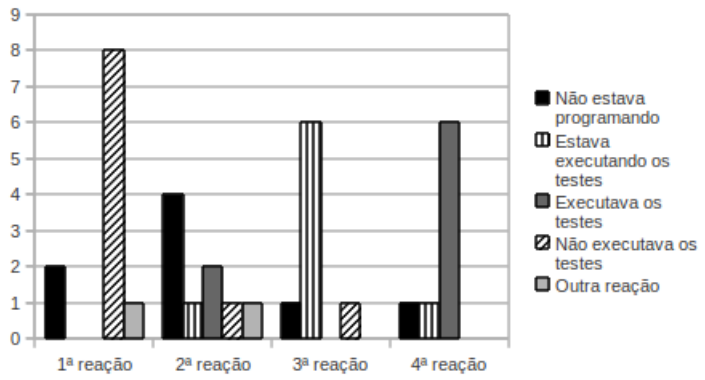


Figura 6.10: *Respostas para a ordem das reações ao lembrete de fazer commit. Cada colocação exibe quantas vezes cada resposta apareceu nela.*

Quanto à percepção que os usuários da ferramenta tiveram sobre a ajuda que ela teve no aprendizado, 55% deles disse que a ferramenta foi indiferente. Outros 27% afirmaram que ela atrapalhou e 18% consideraram que ela ajudou no aprendizado sobre testes.

Para continuar a análise do uso deste lembrete, vamos observar os dados comparativos com os grupos que não usaram a ferramenta.

Na Figura 6.11, vemos a quantidade de vezes que os testes foram executados em cada sessão para os grupos 1, 3 e 4. O grupo 2 foi omitido por brevidade, já que ele teve apenas duas sessões gravadas nessa categoria. Uma delas teve 4 execuções de testes ao longo de aproximadamente uma hora enquanto a outra teve apenas uma execução de testes.

Pela figura, podemos perceber que o comportamento dos grupos com relação a execução dos testes ao longo do tempo nas diferentes sessões é similar: todos os grupos apresentam alternância entre sessões onde são realizadas várias execuções de testes e sessões onde são realizadas poucas. Esse comportamento não mudou ao longo do tempo. Porém, a média de execução de testes por hora calculada excluindo as sessões onde apenas um teste é executado varia: o Grupo 1 teve uma média de $10,75$ enquanto o Grupo 3 teve uma média de $22,74$ e o Grupo 4 teve uma média de $20,26$ execuções de testes por hora.

Vamos ver a seguir os resultados do questionário a respeito das práticas respondido por todos os grupos. Na Figura 6.12 vemos a comparação entre as intensidades ideal e atual relatadas pelos alunos (esquerda) e a comparação entre as frequências subjetivas relatadas por eles (direita).

Das opções apresentadas para a intensidade da prática, foram utilizadas apenas as seguintes:

(a) Os testes automatizados são o design. Se apagar qualquer linha do código, algum teste ficará vermelho. Temos testes de aceitação.

(b) Após pensar no design e escrever um pouco de código, nós escrevemos o teste automatizado.

(c) Nós tomamos o cuidado de escrever alguns testes de unidade para o nosso código depois que terminamos uma história.

As diferenças entre as respostas dos que não usaram a ferramenta e dos que usaram não é muito grande. Porém os que usaram apresentam uma leve tendência a apontar como ideal e atual cenários um pouco menos intensos de uso da prática do que os que não usaram a ferramenta.

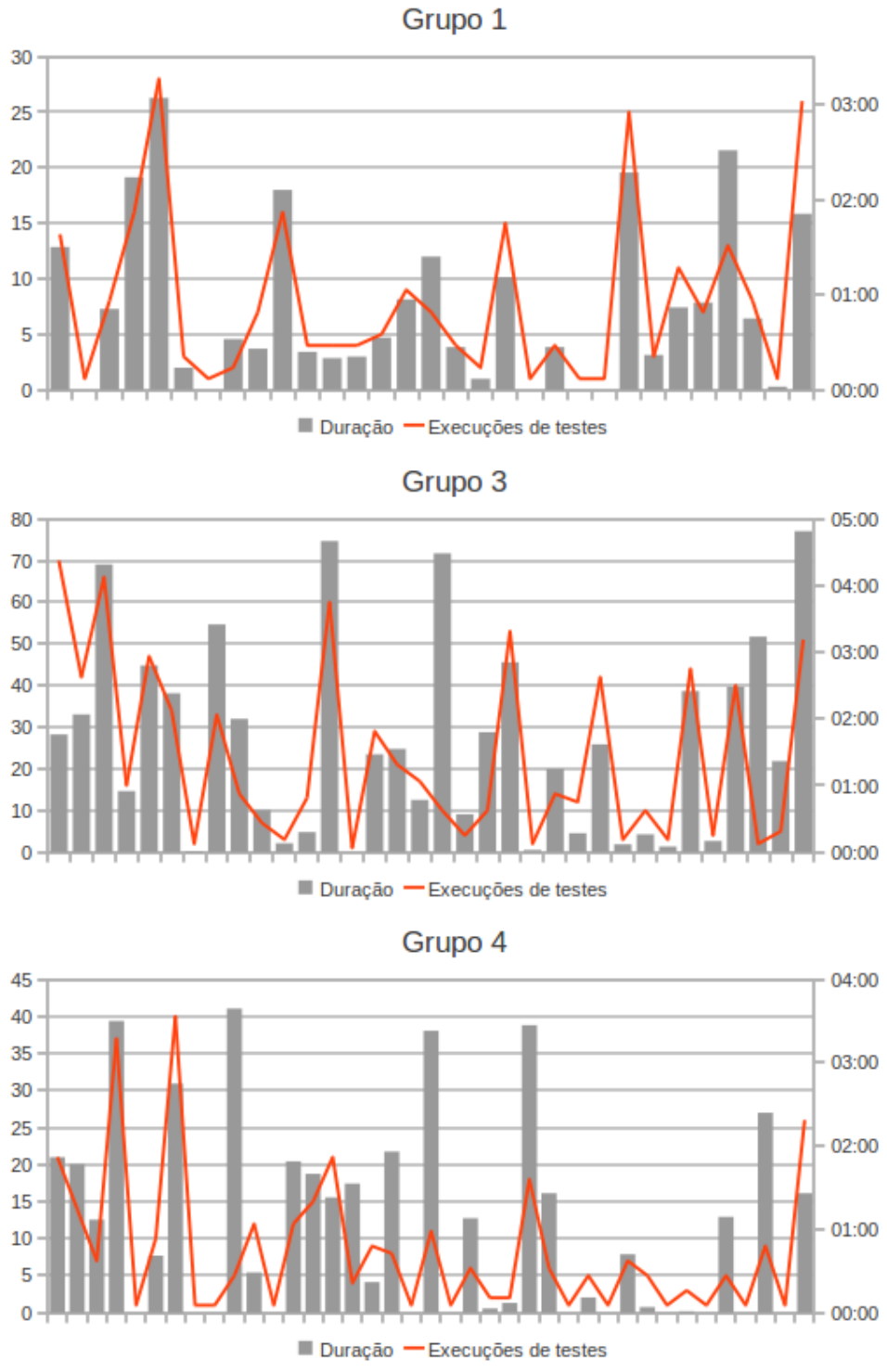
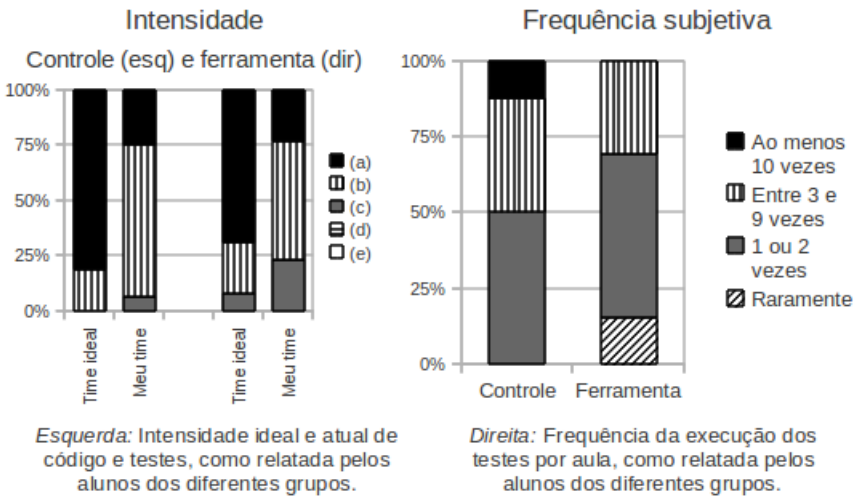


Figura 6.11: *Para cada sessão dos eventos de execução de testes, a quantidade do evento de executar os testes e a duração estimada da sessão. As sessões aparecem em ordem cronológica, separadas por grupo para os grupos 1, 3 e 4. A linha pontilhada indica a média da execução de testes.*



Esquerda: Intensidade ideal e atual de código e testes, como relatada pelos alunos dos diferentes grupos.

Direita: Frequência da execução dos testes por aula, como relatada pelos alunos dos diferentes grupos.

Figura 6.12: *Comparação entre as intensidades ideal e atual relatadas pelos alunos (esquerda) e comparação entre as frequências subjetivas relatadas por eles para* ***código e testes****.*

Com relação à frequência com que os alunos relataram executar os testes, é possível observar uma diferença que reflete a observação feita anteriormente: os respondentes que não usaram a ferramenta relatam executar os testes com um pouco mais de frequência do que os que usaram. De fato, se usarmos a Tabela 6.10 para converter as respostas subjetivas em números aproximados de execução de testes por hora, podemos calcular que os respondentes do grupo de controle relataram executar os testes na média $2,13$ vezes por hora e os respondentes do grupo da ferramenta relataram executar os testes na média $1,37$ vezes por hora.

Tabela 6.10: *Relação das opções na questão sobre frequência subjetiva de trocas de piloto com as estimativas numéricas de frequência usadas para calcular a média.*

Executo os testes pelo menos umas 10 vezes em 2 horas	$\Rightarrow$	5 execuções por hora
Executo os testes pelo entre 3 e 9 vezes em 2 horas	$\Rightarrow$	3 execuções por hora
Executo os testes pelo apenas 1 ou 2 vezes em 2 horas	$\Rightarrow$	$0,75$ execuções por hora
Raramente: executo os testes no máximo 1 vez por semana	$\Rightarrow$	$0,25$ execuções por hora

De forma geral, podemos resumir os resultados observados a respeito da prática **código e testes** com as seguintes conclusões:

- A aparição dos lembretes para executar os testes era quase sempre ignorada pelos usuários;
- Todos os grupos apresentaram alternância entre sessões nas quais muitos testes são executados e sessões nas quais poucos testes são executados, porém a quantidade e frequência de execução dos testes varia de acordo com o grupo;
- O Grupo 2 praticamente não executou testes durante a fase 2;
- Os grupos que não usaram a ferramenta mostraram executar os testes com um pouco mais de frequência do que os grupos que a usaram.

6.2.2 Análise dos intervalos (Fase 2)

Para procurar descobrir qual seria o tempo ideal de cada lembrete, um dos dados foi obtido do questionário aplicado ao final da Fase 2 no qual os alunos responderam sua opinião a esse respeito

(ver Figura 6.13). Para o lembrete de trocar o piloto, a maioria dos respondentes (67%) sugerem um intervalo de 1 hora. Para o lembrete de fazer *commit*, os respondentes ficaram divididos: 33% afirmaram que esse lembrete nunca deveria aparecer ao passo que 29% acredita que um intervalo de 1 hora seria o ideal. Por fim para o lembrete de executar os testes, 46% dos respondentes acha que 30 minutos é o tempo ideal para o intervalo e 20% prefere um intervalo de 1 hora.

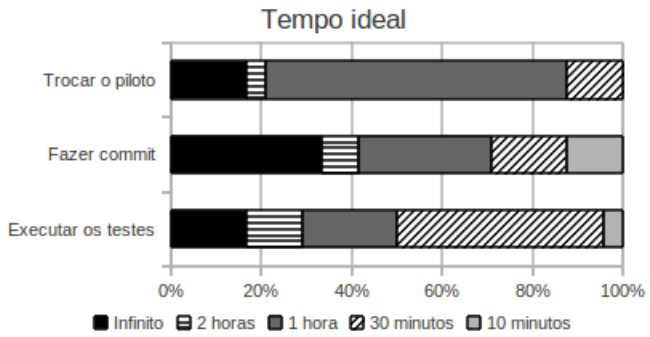


Figura 6.13: *O tempo ideal para cada lembrete de acordo com os usuários da ferramenta.*

Esses dados são completamente subjetivos mas parecem indicar uma vontade de que a aparição dos lembretes seja menos frequente, já que nessa segunda fase todos os grupos tinham lembretes que apareciam a cada 5 ou 10 minutos se nada fosse feito a respeito.

6.2.3 Conclusão do experimento

Como parte do questionário do final da Fase 2, os alunos responderam algumas questões a respeito do uso da ferramenta como um todo. Na Figura 6.14 vemos que 71% deles recomendaria a ferramenta com cuidado. Apenas 8% afirmaram que não recomendariam a ferramenta de maneira alguma.

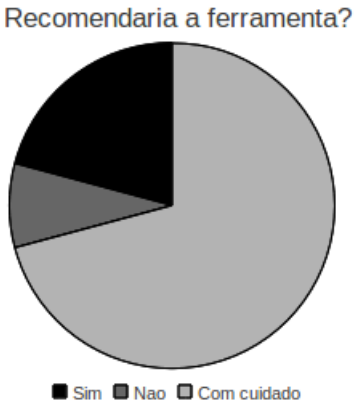


Figura 6.14: *Respostas para a questão sobre se os alunos recomendariam o uso da ferramenta para alguém aprendendo as práticas de XP.*

Na Figura 6.15, vemos quais foram as funcionalidades mais pedidas pelos usuários. A possibilidade de ignorar arquivos que nunca são enviados para o repositório nos lembretes de fazer *commit* recebeu 21% dos votos enquanto os outros 79% se dividiram quase que igualmente entre poder configurar os intervalos dos lembretes e ter notificações que não travem a tela do Eclipse.

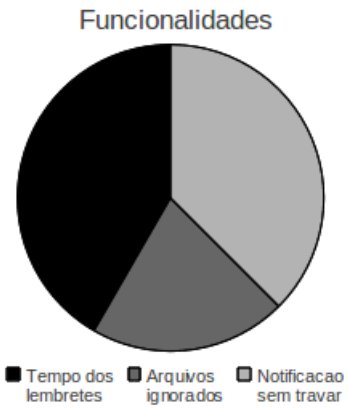


Figura 6.15: *Funcionalidades pedidas pelos usuários.*

Além disso, os alunos escreveram comentários a respeito da ferramenta, seus possíveis benefícios, seus atuais defeitos e sugestões para sua melhoria. Abaixo apresentamos alguns desses comentários:

> "Nem sempre dá para seguir à risca os avisos do *plug-in*. Os avisos podem servir de sinal para algo errado na aplicação da metodologia, mas o tempo entre as atividades que os avisos lembram varia muito de acordo com a tarefa a ser feita. Por exemplo, quando se está lendo/explorando o código nem sempre faz sentido rodar os testes."

> "PRÓS: os lembretes ajudam a não esquecer de práticas importantes de XP, e a pô-las em prática. CONTRAS: (1) os tempos para cada lembrete poderiam ser configuráveis, provavelmente pelo *coach*, para se adequar ao trabalho de cada equipe pois, se aparecerem com muita frequência, desestimula as pessoas a fazer o que é lembrado pelas mensagens, podendo chegar ao ponto de irritá-las de tal modo que elas ignorariam as mensagens; (2) o tempo até que o próximo lembrete de *commit* apareça na tela poderia ser iniciado quando alguma mudança num arquivo de código-fonte fosse feita, pois às vezes as mudanças são feitas em arquivos não tão importantes para o projeto em si (arquivos nunca *comitados* para o repositório)."

> "Pensar em indicadores menos agressivos para o usuário, que possibilite que este os ignore e siga em frente com o desenvolvimento. Por exemplo: trocar cor das abas abertas no editor ou pintar o projeto no *package explorer* de vermelho indicando que já faz um tempo (acho que mostrar o tempo é interessante) que o projeto não foi *comitado* ou que testes não foram executados. Seria interessante colocar também no *plug-in* a geração de relatórios e estatísticas do uso do repositório ou da execução de testes para que a equipe possa definir tempos considerados razoáveis para as notificações, afinal, tal medida subjetiva não deveria ser definida de forma arbitrária. Esses dados também podem ser utilizados como indicativos, no *tracking* da equipe."

> "Talvez a geração de estatísticas para o *coach* fosse uma boa *feature*; assim, o *coach* poderia identificar duplas com problemas e orientá-las, bem como ignorar os casos em que a razão pelo problema identificado é conhecida."

> "Não consigo ver a utilidade do lembrete de *commit*. Para mim, não faz sentido mandar algo que não esteja funcionando para o repositório, a não ser em caso de emergência."

A maioria dos comentários reforça os dados vistos na Figura 6.15, identificando dois problemas que as funcionalidades mais votadas poderiam resolver.

O primeiro problema e, aparentemente, o mais grave é o fato dos lembretes em seu formato atual interromperem o fluxo de trabalho e raciocínio da dupla. Uma notificação deste tipo que tem a intenção não de forçar uma ação mas apenas de lembrá-la não deveria causar essa quebra para o usuário. Os alunos relatam que essa quebra no fluxo causa irritação e, consequentemente, faz com que os avisos sejam ignorados, perdendo o seu propósito. Eles também oferecem ideias e sugestões de como obter uma usabilidade melhor de forma a evitar esse problema.

O segundo problema é o fato de que o intervalo ideal para cada lembrete pode variar de acordo com o grupo e, como observou um dos alunos, pode variar até de acordo com a tarefa ou o tipo de atividade sendo realizada. Boa parte dos alunos sugere que isso pode ser resolvido permitindo a configuração do intervalo de cada lembrete pela equipe ou pelo *coach*.

De forma geral, ficou evidente pelo experimento que a ferramenta de lembretes com as limitações que possui atualmente não traz um benefício significativo ao aprendizado das práticas de XP. Porém, há indícios de que isso poderia ser alterado caso algumas melhorias fossem feitas na ferramenta.

Além disso, pudemos perceber algumas falhas na maneira de organizar o experimento que poderiam ser melhoradas para uma melhor avaliação dos resultados. O primeiro ponto é a existência de fatores que afetam os comportamentos dos grupos com relação às práticas. Por exemplo, o fato de os grupos que usaram a ferramenta na primeira fase trabalharem em uma base de código existente enquanto os grupos que não usaram terem começado seu projeto do zero pode ser um fator que contribuiu para as diferenças observadas nas práticas de integração contínua e código e testes. Outro ponto é que a ferramenta de acompanhamento das ações do usuário poderia ter algumas melhorias no que diz respeito aos *logs* gravados. Algumas delas já foram citadas, como gravar o momento de abertura e fechamento do Eclipse para fornecer a duração real da sessão ao invés de estimada.

Capítulo 7

Conclusão

Neste trabalho, propusemos e avaliamos duas maneiras de facilitar o ensino de algumas práticas ágeis: programação em pares, código e testes, desenvolvimento dirigido por testes (TDD), refatoração, integração contínua e retrospectiva.

A primeira maneira proposta foi o uso de *Dojos* de programação para propiciar um ambiente seguro onde os participantes possam treinar as práticas e receberem *feedback*. Através de um questionário, analisamos a percepção que os participantes de *Dojos* têm a respeito do aprendizado que essa técnica propicia para as seguintes práticas: programação em pares, TDD, refatoração, *commits frequentes* e retrospectiva.

Os resultados indicam que o *Dojo* é percebido como um bom ambiente para tal aprendizado.Também notamos uma tendência de que quanto menos o participantes souber sobre uma prática antes de participar de um *Dojo*, mais ele pode aprender sobre ela. Além disso, TDD se destacou como a prática sobre a qual os participantes mais aprendem com o *Dojo*.

A segunda maneira proposta foi o uso de ferramentas para observar um indivíduo ou dupla enquanto eles trabalham e fornecer *feedback* a respeito das práticas. Para avaliar a potencialidade desta ideia como uma técnica que facilite o ensino de métodos ágeis, desenvolvemos um protótipo dessa ferramenta que acompanha a frequência de algumas ações do usuário na área de trabalho e abre um lembrete para executar a ação caso ela fique um determinado intervalo de tempo sem aparecer. Esse protótipo focou em 3 práticas de XP: programação em pares com a ação troca de piloto, código e testes com a ação execução dos testes e integração contínua com a ação fazer *commit* de todos os projetos sob controle de versões.

Esse protótipo foi utilizado por estudantes do Laboratório de Programação Extrema no IME-USP em um experimento que analisou os efeitos de uso e de diferença nos intervalos. Nesse experimento, constatamos que a ferramenta implementada possui algumas limitações que dificultaram avaliar se ela pode, de fato, ajudar no aprendizado das práticas. Apesar disso, o estudo apontou indícios de que, com uma melhor usabilidade, a ferramenta de *feedback* automatizado poderia ser útil para o ensino das práticas. Além disso, o mecanismo desenvolvido para observar e gravar as ações realizadas pelo usuário enquanto trabalha mostrou-se uma ferramenta que pode ajudar a entender melhor como as duplas trabalham.

No decorrer deste trabalho foram publicados dois artigos com os resultados obtidos:

- Danilo T. Sato, Hugo Corbucci e Mariana V. Bravo. Coding dojo: An environment for learning and sharing agile practices. Em *AGILE '08: Proceedings of the Agile 2008* [33].

- Mariana Bravo e Alfredo Goldman. Reinforcing the Learning of Agile Practices Using Coding *Dojos*. Em Agile Processes in Software Engineering and Extreme Programming [11].

Fica claro que essas duas ferramentas tem potencial para o ensino das práticas de programação extrema abordadas neste estudo. Porém algum trabalho ainda é necessário para entender melhor qual seria o seu papel no aprendizado. Com relação ao *Dojo* de programação, seria interessante conduzir um estudo controlado com participantes de diferentes níveis de conhecimento e motivação, avaliando de forma mais objetiva o seu conhecimento antes e depois de participar do *Dojo*, bem como qual a influência que o número de sessões tem no aprendizado. Quanto às ferramentas de *feedback* automatizado, seria necessário em primeiro lugar melhorar sua usabilidade e acrescentar algumas funcionalidades ao sistema de acompanhamento. Após isso, seria possível conduzir novamente um estudo comparativo avaliando os efeitos do uso da ferramenta.

Apêndice A

Questionário sobre *Dojo* de programação e aprendizado

0. **Indique o país e a cidade onde você mora:**
   Cidade: _____
   País: _____

1. **De quantas sessões de *Dojo* de Programação você já participou?**
   (Escolha apenas uma opção)

   (a) Nenhuma
   (b) Entre 1 e 5
   (c) Entre 6 e 10
   (d) Entre 11 e 20
   (e) Mais que 20

2. **Marque as práticas utilizadas nas sessões de *Dojo* de Programação das quais você já participou:**
   (Escolha todas que se aplicam)

   (a) Programação em pares
   (b) Desenvolvimento dirigido por testes (TDD)
   (c) Commits frequentes: fazer commits do código (ao longo da sessão)
   (d) Retrospectiva
   (e) Refatoração

3. **De acordo com as definições abaixo, indique para cada uma das práticas o seu grau de familiaridade com ela ANTES de sua primeira sessão de *Dojo* de Programação.**

   (a) **Não conhecia** - Nunca tinha ouvido falar dessa prática
   (b) **Teórico** - Já tinha lido a respeito em livros e na internet
   (c) **Iniciante** - Já tinha brincado e experimentado a prática em algumas ocasiões
   (d) **Fluente** - Já usava a prática com frequência
   (e) **Especialista** - Já dominava muito bem a prática, tendo mais de um ano de experiência com ela

3.1	**Programação em pares**	(a)	(b)	(c)	(d)	(e)
3.2	**TDD**	(a)	(b)	(c)	(d)	(e)
3.3	***Commits* frequentes**	(a)	(b)	(c)	(d)	(e)
3.4	**Retrospectiva**	(a)	(b)	(c)	(d)	(e)
3.5	**Refatoração**	(a)	(b)	(c)	(d)	(e)

4. **Como você diria que participar em sessões de *Dojo* de Programação ajudou no seu aprendizado/evolução em cada uma das práticas?**

   (a) **Atrapalhou** - participar do *Dojo* de Programação atrapalhou o meu aprendizado

   (b) **Indiferente** - não teve nenhum efeito no meu aprendizado

   (c) **Interessou** - despertou meu interesse na prática, mas aprendi mesmo em outro lugar

   (d) **Pouco** - ajudou, mas pouco

   (e) **Médio** - ajudou médio, tive outras fontes para aprender também

   (f) **Muito** - aprendi praticamente só com o *Dojo* de Programação

4.1	**Programação em pares**	(a)	(b)	(c)	(d)	(e)	(f)
4.2	**TDD**	(a)	(b)	(c)	(d)	(e)	(f)
4.3	***Commits* frequentes**	(a)	(b)	(c)	(d)	(e)	(f)
4.4	**Retrospectiva**	(a)	(b)	(c)	(d)	(e)	(f)
4.5	**Refatoração**	(a)	(b)	(c)	(d)	(e)	(f)

5. **Quais os motivos para você participar do *Dojo* de Programação?**
   (Escolha todas que se aplicam)

   (a) Treinar técnicas que já conheço para não perder a prática

   (b) Aprender novas técnicas

   (c) Ensinar técnicas que conheço para outras pessoas

   (d) Trocar experiências com os outros participantes

   (e) Me divertir

   (f) Curiosidade, queria saber como é

   (g) Outro (preencha a seguir): _____

6. **Comentários, sugestões, observações? Escreva abaixo.**

_____
_____
_____
_____
_____

Apêndice B

Questionários sobre as ferramentas de *feedback* automático

B.1 Fase 1 – Sobre as Práticas

0. **Qual o seu grupo no Laboratório de Programação Extrema?**

   (a) Grupo 1
   (b) Grupo 2
   (c) Grupo 3
   (d) Grupo 4

Parte I. Programação em pares

Considerando as opções abaixo:

(a) Nunca ou quase nunca fazemos *commits* de código que não foi feito em par. Nós fazemos o rodízio constante de pares.

(b) Nós geralmente trabalhamos em pares, mas de vez em quando fazemos *commits* de trabalho feito sozinho.

(c) Nós geralmente fazemos discussões no quadro branco, conversamos por *chat* ou no laboratório. Algumas pessoas programam em pares, mas outras preferem não tentar.

(d) Nós tentamos parear mas não conseguimos devido a desencontro de horários e reuniões. Algumas pessoas são muito rápidas ou devagares para que eu tenha paciência de sentar junto.

(e) Eu uso fones de ouvido para que as pessoas não me interrompam. Na verdade, eu prefiro mesmo é trabalhar em casa.

1. **Qual delas você considera ser o ideal para uma equipe XP?**
   (a) (b) (c) (d) (e)

2. **Qual delas você considera ser o que acontece na sua equipe XP?**
   (a) (b) (c) (d) (e)

Na programação em pares, dizemos que o piloto é aquele que usa o teclado e navegador é aquele que observa, opina e procura melhorias ou erros no código.

3. **Considerando um período de 2 (duas) horas de trabalho, quando você programa em par você diria que, em geral:**

   (a) Trocamos de piloto pelo menos 3 vezes.

   (b) Trocamos de piloto 1 ou 2 vezes.

   (c) Nesse período de tempo, o piloto não muda.

4. **Comentários ou sugestões? Preencha abaixo:**

_____
_____
_____
_____
_____

Parte II. Testes e Código

Considerando as opções abaixo:

(a) Os testes automatizados são o design. Se apagar qualquer linha do código, algum teste ficará vermelho. Temos testes de aceitação.

(b) Após pensar no design e escrever um pouco de código, nós escrevemos o teste automatizado.

(c) Nós tomamos o cuidado de escrever alguns testes de unidade para o nosso código depois que terminamos uma história.

(d) Nós ouvimos falar de ferramentas como o JUnit, RUnit, PyUnit, mas nunca utilizamos.

(e) Nós não temos nenhum tipo de teste formal. Os clientes geralmente nos avisam se encontrarem algum problema.

5. **Qual delas você considera ser o ideal para uma equipe XP?**
   (a) (b) (c) (d) (e)

6. **Qual delas você considera ser o que acontece na sua equipe XP?**
   (a) (b) (c) (d) (e)

7. **Quando você está trabalhando, com qual frequência executa os testes:**

   (a) Executo os testes pelo menos umas 10 vezes em 2 horas.

   (b) Executo os testes entre 3 e 9 vezes em 2 horas.

   (c) Executo os testes apenas 1 ou 2 vezes em 2 horas.

   (d) Executo os testes no máximo 1 vez por semana.

8. **Comentários ou sugestões? Preencha abaixo:**

_____
_____
_____
_____
_____

Parte III. Integração Contínua

Considerando as opções abaixo:
Quando trabalho em uma funcionalidade, sincronizo e disponibilizo o código:

(a) Diversas vezes por dia.

(b) Uma vez por dia.

(c) Uma vez por semana.

(d) Algumas semanas podem se passar. Só disponibilizamos código quando a tarefa está pronta.

(e) Geralmente tenho problemas porque muitas mudanças acontecem entre o momento em que eu faço *checkout* e quando tento sincronizar. Tenho muitos conflitos e preciso tomar cuidado pois geralmente posso esquecer algumas coisas.

9. **Qual delas você considera ser o ideal para uma equipe XP?**
(a) (b) (c) (d) (e)

10. **Qual delas você considera ser o que acontece na sua equipe XP?**
(a) (b) (c) (d) (e)

11. **Quando você está trabalhando, com qual frequência faz *commit* de todo(s) o(s) projeto(s)?**

(a) Pelo menos 2 vezes num período de 2 horas.

(b) Apenas 1 vez num período de 2 horas.

(c) Posso ficar dias sem fazer *commit* de tudo.

(d) Existem arquivos sob controle de versão que eu nunca faço *commit*.

12. **Comentários ou sugestões? Preencha abaixo:**

_____
_____
_____
_____
_____

B.2 Fase 1 – Sobre a Ferramenta

0. **Qual o seu grupo no Laboratório de Programação Extrema?**

   (a) Grupo 1
   (b) Grupo 2
   (c) Grupo 3
   (d) Grupo 4

Parte I. Programação em Pares

1. **Com qual frequência você clicou no botão "Trocar piloto"?**

   (a) Num período de 2 horas, pelo menos 3 vezes.
   (b) Num período de 2 horas, 1 ou 2 vezes.
   (c) Muito raramente (1 vez por semana ou menos)
   (d) Qual botão "Trocar piloto"?

2. **Quando aparecia o aviso para trocar de piloto, você e seu par:**

   1. A gente nem estava programando de fato.
   2. A gente trocava de piloto.
   3. A gente clicava em OK mas não trocava de piloto.
   4. Outra opção: _____ (Opcional)

   Ordene os itens acima do mais frequente para o menos frequente: _____
   Exemplo: 2-3-1 quer dizer que o item 2 era mais frequente, seguido pelo 3 e por fim o 1, menos frequente de todos. Se você preencher o item 4, coloque ele na ordem também.

3. **Eu acho que o lembrete para trocar de piloto:**

   (a) Ajudou a aprender programação em pares.
   (b) Não fez diferença alguma para o meu aprendizado de programação em pares.
   (c) Não ajudou a aprender programação em pares e ainda atrapalhou a produtividade da minha dupla.

4. **Comentários ou sugestões? Preencha abaixo:**

   _____
   _____
   _____
   _____
   _____

Parte II. Testes e Código

5. **Quando aparecia o aviso para excutar os testes, você e seu par:**

   1. A gente nem estava programando de fato.
   2. A gente estava no meio de uma execução dos testes.
   3. A gente executava os testes.
   4. A gente clicava em OK mas não executava os testes.

5. Outra opção: _____ (Opcional)

Ordene os itens acima do mais frequente para o menos frequente: _____
Exemplo: 2-4-3-1 quer dizer que o item 2 era mais frequente, seguido pelo 4, seguido pelo 3 e por fim o 1, menos frequente de todos. Se você preencher o item 5, coloque ele na ordem também.

6. **Eu acho que o lembrete para executar os testes:**

   (a) Ajudou a aprender sobre testes, dar importância a eles e lembrar de fazer bons testes.

   (b) Não fez diferença alguma para o meu aprendizado de testes e sua importância.

   (c) Não ajudou a aprender sobre testes e ainda atrapalhou a produtividade da minha dupla.

7. **Comentários ou sugestões? Preencha abaixo:**

_____
_____
_____
_____
_____

Parte III. Integração Contínua

8. **Quando aparecia o aviso para fazer commit, você e seu par:**

   1. A gente estava fazendo update.
   2. A gente fazia um commit na hora.
   3. A gente clicava em OK mas não fazia commit algum.
   4. Outra opção: _____ (Opcional)

   Ordene os itens acima do mais frequente para o menos frequente: _____
   Exemplo: 2-3-1 quer dizer que o item 2 era mais frequente, seguido pelo 3 e por fim o 1, menos frequente de todos. Se você preencher o item 4, coloque ele na ordem também.

9. **Eu acho que o lembrete para fazer commits:**

   (a) Ajudou a aprender integração contínua de verdade.

   (b) Não fez diferença alguma para o meu aprendizado de integração contínua.

   (c) Não ajudou a aprender sobre integração contínua e ainda atrapalhou a produtividade da minha dupla.

10. **Comentários ou sugestões? Preencha abaixo:**

_____
_____
_____
_____
_____

B.3 Fase 2 – Sobre os Tempos

0. **Qual o seu grupo no Laboratório de Programação Extrema?**

   (a) Grupo 1
   (b) Grupo 2
   (c) Grupo 3
   (d) Grupo 4

Veja na tabela abaixo os tempos de lembrete usados no seu grupo:

Lembrete para...	**Executar os testes**	**Trocar o piloto**	**Fazer *commit***
Grupo 1	5 min.	40 min.	60 min.
Grupo 2	55 min.	10 min.	100 min.
Grupo 3	30 min.	55 min.	10 min.
Grupo 4	10 min.	25 min.	30 min.

1. **Qual tempo você considera adequado para o lembrete de rodar os testes?**

   (a) 10 minutos
   (b) 30 minutos
   (c) 60 minutos
   (d) 120 minutos
   (e) Infinito - o lembrete mais atrapalha do que ajuda

2. **Qual tempo você considera adequado para o lembrete de trocar o piloto?**

   (a) 10 minutos
   (b) 30 minutos
   (c) 60 minutos
   (d) 120 minutos
   (e) Infinito - o lembrete mais atrapalha do que ajuda

3. **Qual tempo você considera adequado para o lembrete de fazer commit?**

   (a) 10 minutos
   (b) 30 minutos
   (c) 60 minutos
   (d) 120 minutos
   (e) Infinito - o lembrete mais atrapalha do que ajuda

4. **Você recomendaria o uso desses plugins para alguém aprendendo as práticas de XP?**

   (a) Não recomendaria. Não consigo imaginar como a ferramenta poderia ser útil.
   (b) Recomendaria muito. Achei a ferramenta muito boa para aprendizado das práticas.
   (c) Recomendaria com cuidado, observando prós e contras (Preencha abaixo):

   _____
   _____
   _____
   _____
   _____

5. **A ferramenta é um trabalho em andamento, várias funcionalidades estão faltando. Qual das sugestões abaixo você considera a mais importante de ser implementada?**

   (a) Configurar o tempo de cada lembrete.

   (b) Ignorar alguns arquivos que nunca são comitados para o repositório.

   (c) Notificar sem travar a tela do eclipse.

   (d) Lembrete para outra prática: _____

   (e) Outra funcionalidade: _____

6. **Comentários ou sugestões? Preencha abaixo:**

_____
_____
_____
_____
_____

Referências Bibliográficas

[1] Coding dojo principles. http://codingdojo.org/cgi-bin/wiki.pl?CodingDojoPrinciples. Último acesso: Abril 2010. 15

[2] Manifesto for agile software development. http://agilemanifesto.org/, 2001. 3

[3] Lyssa Adkins. *Coaching Agile Teams: A Companion for ScrumMasters, Agile Coaches, and Project Managers in Transition.* Addison-Wesley Professional, 1st edition, 2010. ISBN 0321637704, 9780321637703. 8

[4] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, us ed edition, 1999. 3, 15

[5] Kent Beck. *Test driven development: By example.* Addison-Wesley Professional, us ed edition, 2003. 6, 18

[6] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, 2nd Edition.* Addison-Wesley Professional, 2nd edition, 2004. 4

[7] Petra Becker-Pechau, Holger Breitling, Martin Lippert, and Axel Schmolitzky. Teaching team work: An extreme week for first-year programmers. In Michele Marchesi and Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003. URL http://dx.doi.org/10.1007/3-540-44870-5_59. 11

[8] Vieri Del Bianco and Giordano Sassaroli. Agile teaching of an agile software process. In Michele Marchesi and Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 25-29, 2003 Proceedings*, volume 2675 of *Lecture Notes in Computer Science*, pages 402–405. Springer, 2003. ISBN 3-540-40215-2. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2675&spage=402. 10

[9] Laurent Bossavit. Good programming. http://www.bossavit.com/pivot/pivot/entry.php?id=207. Último acesso: Agosto 2010., 2003. 17

[10] Laurent Bossavit. Object dojo. http://www.bossavit.com/pivot/pivot/entry.php?id=253. Último acesso: Agosto 2010., 2004. 17

[11] Mariana Bravo and Alfredo Goldman. Reinforcing the learning of agile practices using coding dojos. In Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Alberto Sillitti, Angela Martin, Xiaofeng Wang, and Elizabeth Whitworth, editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 48 of *Lecture Notes in Business Information Processing*, pages 379–380. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-13054-0. URL http://dx.doi.org/10.1007/978-3-642-13054-0_41. 10.1007/978-3-642-13054-0_41. 58

[12] Christian Bunse, Raimund L. Feldmann, and Jörg Dörr. Agile methods in software engineering education. In Jutta Eckstein and Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings*, volume 3092 of *Lecture Notes in Computer Science*, pages 284–293. Springer, 2004. ISBN 3-540-22137-9. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3092&spage=284. 10

[13] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison Wesley Professional, 3rd edition, 2009. 23

[14] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. pages 223–243, 2001. 6

[15] Rachel Davies and Liz Sedley. *Agile Coaching*. Pragmatic Bookshelf, 1st edition, 2009. ISBN 1934356433, 9781934356432. 8

[16] Scott Delap. Understanding how eclipse plug-ins work with osgi. http://www.ibm.com/developerworks/library/os-ecl-osgi/index.html. Último acesso: Dezembro 2010., 2006. 24

[17] Esther Derby and Diana Larsen. *Agile Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, 2006. ISBN 0977616649. 8

[18] Yael Dubinsky and Orit Hazzan. Roles in agile software development teams. In *5th Int. Conf. on eXtreme Programming and Agile Processes in Software Engineering*, pages 157–165, 2004. 12

[19] K Anders Ericsson. Deliberate practice and the acquisition and maintenance of expert performance in medicine and related domains. *Academic Medicine*, 79(10):89–93, 2004. 19

[20] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, August 1999. ISBN 0201485672. 6

[21] Alexandre Freire, Alfredo Goldman, and Fábio Kon. Three antipractices while teaching agile methods. Technical report, IME-USP, 2007. 5

[22] Erich Gamma, Richard Helm, John Vlissides, and Ralf Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, first edition, 01 1995. 6, 29

[23] Görel Hedin, Lars Bendix, and Boris Magnusson. Teaching extreme programming to large groups of students. *J. Syst. Softw.*, 74:133–146, January 2005. ISSN 0164-1212. doi: 10.1016/j.jss.2003.09.026. URL http://portal.acm.org/citation.cfm?id=1045926.1045930. 12

[24] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X. 6

[25] Andrew Hunt and David Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003. ISBN 0974514012. 6

[26] David G. Jones. From 0 to 1,169,600 in 3 minutes: nine months of testing by the devcreek team. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 731–731, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: http://doi.acm.org/10.1145/1297846.1297857. 7

[27] James B. Fenwick Jr. Adapting XP to an academic environment by phasing-in practices. In Frank Maurer and Don Wells, editors, *Extreme Programming and Agile Methods - XP/Agile*

Universe 2003, Third XP and Second Agile Universe Conference, New Orleans, LA, USA, August 10-13, 2003, Proceedings, volume 2753 of *Lecture Notes in Computer Science*, pages 162–171. Springer, 2003. ISBN 3-540-40662-X. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2753&spage=162. 11

[28] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2nd edition, 2010. 23

[29] Dawn McKinney, Julie Froeseth, Jason Robertson, Leo F. Denton, and David Ensminger. Agile CS1 labs: extreme programming practices in an introductory programming course. In Carmen Zannier, Hakan Erdogmus, and Lowell Lindstrom, editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2004, 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004, Proceedings*, volume 3134 of *Lecture Notes in Computer Science*, pages 164–174. Springer, 2004. ISBN 3-540-22839-X. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3134&spage=164. 11

[30] Rick Mugridge, Bruce MacDonald, Partha S. Roop, and Ewan D. Tempero. Five challenges in teaching XP. In Michele Marchesi and Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 25-29, 2003 Proceedings*, volume 2675 of *Lecture Notes in Computer Science*, pages 406–409. Springer, 2003. ISBN 3-540-40215-2. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2675&spage=406. 11

[31] Roger A. Müller. Extreme programming in a university project. In Jutta Eckstein and Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings*, volume 3092 of *Lecture Notes in Computer Science*, pages 312–315. Springer, 2004. ISBN 3-540-22137-9. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3092&spage=312. 9

[32] Jack W. Reeves. What is software design? http://www.developerdotstar.com/mag/articles/reeves_design_main.html. Último acesso: Abril 2010. 15

[33] Danilo Toshiaki Sato, Hugo Corbucci, and Mariana Vivian Bravo. Coding dojo: An environment for learning and sharing agile practices. In *AGILE '08: Proceedings of the Agile 2008*, pages 459–464, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3321-6. doi: http://dx.doi.org/10.1109/Agile.2008.11. 18, 57

[34] James Shore and Shane Warden. *The art of agile development*. O'Reilly, first edition, 2007. ISBN 9780596527679. 7

[35] Jaime Spacco and William Pugh. Helping students appreciate test-driven development (tdd). In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 907–913, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. doi: http://doi.acm.org/10.1145/1176617.1176743. URL http://doi.acm.org/10.1145/1176617.1176743. 12

[36] Dave Thomas. Code kata. http://codekata.pragprog.com/2007/01/code_kata_backg.html. Último acesso: Agosto 2010., . 17

[37] Dave Thomas. Code kata – how it started. http://codekata.pragprog.com/2007/01/code_katahow_it.html. Último acesso: Agosto 2010., . 17

[38] Michael Wainer. Adaptations for teaching software development with extreme programming: An experience report. In Frank Maurer and Don Wells, editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2003, Third XP and Second Agile*

Universe Conference, New Orleans, LA, USA, August 10-13, 2003, Proceedings, volume 2753 of *Lecture Notes in Computer Science*, pages 199–207. Springer, 2003. ISBN 3-540-40662-X. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2753&spage=199. 10

[39] Carol Wellington, Thomas Briggs, and C. Dudley Girard. The impact of agility on a bachelor's degree in computer science. In *Proceedings of the conference on AGILE 2006*, pages 400–404, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2562-8. doi: 10.1109/AGILE.2006.54. URL http://portal.acm.org/citation.cfm?id=1155439.1155497. 12

[40] Laurie A. Williams and Robert R. Kessler. All I really need to know about pair programming I learned in kindergarten. *Commun. ACM*, 43(5):108–114, 2000. URL http://doi.acm.org/10.1145/332833.332848. 6