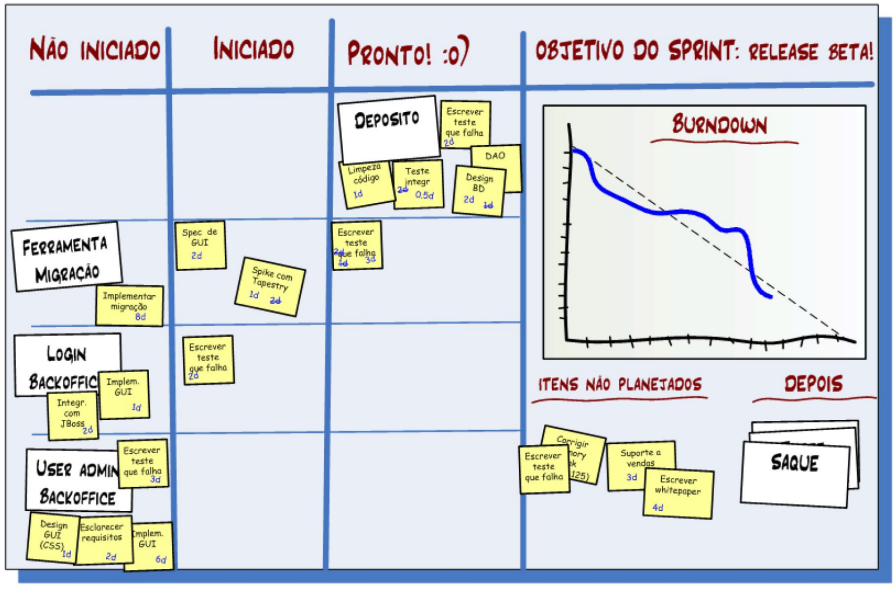
Uma história ágil de guerra

Scrum e XP direto das Trincheiras

Como nós fazemos Scrum



Henrik Kniberg

Prefácios por Jeff Sutherland, Mike Cohn

InfoQ Enterprise Software Development Community

Série Desenvolvimento de Software Corporativo

FREE ONLINE EDITION

(non-printable free online version)

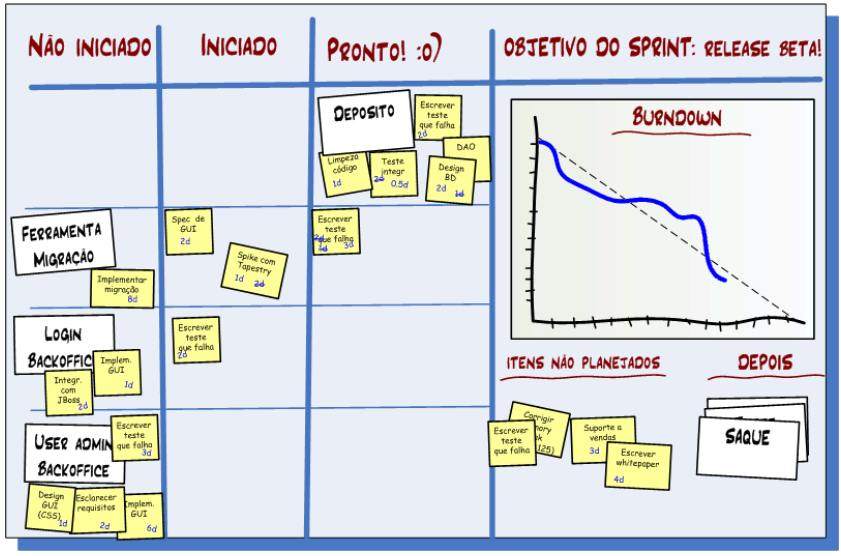
Trazido para você como

Cortesia de

Visite a página deste livro em:
http://infoq.com/br/minibooks/scrum-xp-from-the-trenches

Scrum e XP direto das Trincheiras

Como fazemos Scrum



Escrito Por:
Henrik Kniberg

C4Media, Publisher of InfoQ.com.

Este livro é parte da série InfoQ de livros sobre Desenvolvimento de Software Corporativo.

Para informações sobre compra desse ou outros livros InfoQ, entre em contato com books@c4media.com.

Editor: Diana Plesa / Felipe Rodrigues

Tradutores: Vários voluntários gerenciados pela SEA Tecnologia

Dados de Publicação-em-Catálogo da Biblioteca do Congresso Norte-Americano:

ISBN: 978-1-4303-2264-1

Dedicatória

O primeiro rascunho desse livro demorou apenas um final de semana para ser digitado, mas com certeza foi um final de semana intenso! Fator de foco de 150% :o)

Obrigado à minha esposa Sophia e aos meus filhos Dave e Jenny por agüentar minha falta de sociabilidade naquele fim de semana. E obrigado também aos pais de Sophia, Eva e Jörgen por virem ajudar a tomar conta da família.

Obrigado também aos meus colegas da Crisp, em Estocolmo, e às pessoas do grupo de discussão do yahoo scrumdevelopment por revisarem e me ajudarem a melhorar este trabalho.

E, finalmente, obrigado a todos os leitores que forneceram um canal constante de feedback bastante útil. Eu estou particularmente satisfeito por ouvir que esse trabalho tem impactado tantos de vocês a ponto de darem uma chance ao desenvolvimento ágil de software.

Conteúdo

Prefácio por Jeff Sutherland

As equipes precisam conhecer o básico do Scrum. Como você cria e estima um *product backlog*? Como você o transforma num *sprint backlog*? Como você gerencia um gráfico *burndown* e calcula a velocidade de sua equipe? O livro do Henrik é um *kit* para iniciantes com práticas basicas que ajudam equipes a irem além de apenas tentativas de praticar o *Scrum* para executá-lo bem.

Uma boa execução do *Scrum* está se tornando mais importante para as equipes que buscam investimento de fundos externos. Como um mentor Ágil para um grupo de capital de risco, eu os ajudo a investirem em empresas que executam as práticas Ágeis com eficiência. O Parceiro Sênior do grupo tem perguntado a todas as empresas do portfólio se elas conhecem a velocidade de suas equipes. Elas têm dificuldade em responder a essa questão de imediato. Oportunidades futuras de investimento irão exigir que as equipes de desenvolvimento saibam sua velocidade de produção.

Por que isso é tão importante? Se as equipes não conhecem sua própria velocidade, o *product owner* não pode criar um guia de produto com datas de releases com credibilidade. E sem datas de release interdependentes, a empresa poderá falhar e os investidores poderão perder seu dinheiro!

Esse problema é enfrentado por companhias grandes e pequenas, novas ou antigas, com investimento próprio ou externo. Numa discussão recente sobre a implantação do *Scrum* no Google, numa conferência em Londres, eu perguntei às 135 pessoas presentes quantos deles estavam praticando *Scrum*, e 30 responderam positivamente. Então eu perguntei se eles estavam fazendo o desenvolvimento iterativo usando os padrões da Nokia. Desenvolvimento iterativo é fundamental no Manifesto Ágil – entregue software funcionando cedo e com freqüência. Depois de anos de retrospectivas com centenas de equipes de *Scrum*, a Nokia desenvolveu alguns requisitos básicos para o desenvolvimento iterativo:

- As iterações devem ter um tempo fixo com menos de seis semanas de duração.
- Ao final da iteração, o código deve ser testado pelo CQ e funcionar corretamente.

Das 30 pessoas que responderam estar usando o *Scrum*, apenas metade disse que estava atendendo o primeiro princípio do Manifesto Ágil pelos padrões da Nokia. Então eu perguntei se eles atendiam aos padrões da Nokia para o *Scrum*:

- Uma equipe de *Scrum* deve ter um *product owner* e saber quem ele é.
- O *product owner* deve ter um *product backlog* com estimativas criadas pela equipe.
- A equipe deve ter um gráfico *burndown* e saber sua velocidade.
- Não deve haver nenhuma interferência externa sobre a equipe durante um *sprint*.

Das 30 pessoas praticantes do *Scrum*, apenas 3 atendiam o teste da Nokia para identificar uma equipe de *Scrum*. Essas seriam as únicas que receberiam futuros investimentos dos meus parceiros.

O valor do livro do Henrik é que se você seguir as práticas que ele recomenda, você terá um *product backlog*, estimativas para o *product backlog*, um gráfico *burndown*, e saberá a velocidade de sua equipe, além de outras práticas fundamentais para um *Scrum* altamente funcional. Você passará no teste da Nokia e estará apto a receber investimento no seu trabalho. Se você for uma empresa iniciante, poderia até receber investimento de um grupo de capital de risco. Você pode ser o futuro do desenvolvimento de software e o criador da próxima geração dos produtos líderes de mercado.

Jeff Sutherland,
Ph.D., Co-Criador do Scrum

Prefácio por Mike Cohn

Ambos *Scrum* e *Extreme Programming* (XP) pregam que a equipe complete alguma parte palpável de trabalho que seja entregável ao fim de cada iteração. Essas iterações são pensadas para serem curtas e com espaço de tempo definido. Este foco em entregas de código funcionando em um curto espaço de tempo significa que equipes *Scrum* e XP não têm tempo para teorias. Eles não perseguem o desenho do diagrama UML perfeito em ferramentas case, a escrita do documento de requsitos perfeito, ou a escrita do código que poderá acomodar todas as mudanças futuras imagináveis. Ao invés disso, equipes *Scrum* e XP focam em ter as coisas prontas. Essas equipes aceitam que cometem erros ao longo do caminho, mas também compreendem que o melhor modo de identificar esses erros é parar de pensar sobre o software em um nível teórico de análise e design e mergulhar fundo, sujar as mãos e começar a construir o produto.

É esse mesmo foco em fazer ao invés de teorizar que distingue esse livro. E aparentemente Henrik Kniberg entende isso direito, desde o princípio. Ele não oferece uma longa descrição sobre o que é *Scrum*; ele nos remete a alguns sites simples para isso. Ao invés disso, Henrik dá um salto e começa imediatamente descrevendo como suas equipes gerenciam e trabalham com seu *product backlog*. Daí, ele parte para todos os outros elementos e práticas de um projeto ágil bem executado. Sem teorias, sem referências. Sem notas de rodapé. Nada disso é necessário. O livro de Henrik não é uma explanação filosófica sobre por que *Scrum* funciona ou por que você poderia querer experimentá-lo. É uma descrição de como uma boa equipe ágil funciona.

Eis o porquê do subtítulo do livro, “Como fazemos Scrum”, ser tão apropriado. Pode não ser o modo como *você* faz *Scrum*, mas é o modo como as equipes do Henrik fazem *Scrum*. Você pode perguntar por que deveria se importar com a forma como outra equipe faz *Scrum*. Você deveria se importar porque todos podemos aprender como fazer melhor *Scrum* ouvindo histórias de como foi feito por outros, especialmente aqueles que estão fazendo direito. Não existe, e nunca existirá uma lista de “melhores práticas em *Scrum*”, porque o contexto dos projetos e equipes descarta todas as outras considerações. Ao invés de melhores práticas, o que precisamos conhecer são boas práticas e o contexto onde foram bem sucedidas. Leia histórias suficientes de equipes bem sucedidas e como elas fizeram as coisas e você estará preparado para os obstáculos que se apresentarem a você e ao seu uso de *Scrum* e XP.

Henrik provê as boas práticas e o contexto necessário para nos ajudar a aprender melhor como fazer *Scrum* e XP, nas trincheiras dos seus projetos.

Mike Cohn
Autor de *Agile Estimating and Planning* e *User Stories Applied for Agile Software Development.*

Prefácio – Ei, Scrum funcionou!

Scrum funcionou! Pelo menos para nós (me refiro ao meu cliente atual em Stockholm, cujo nome não pretendo mencionar aqui). Espero que ele funcione pra você também. Talvez esse livro te ajude ao longo do caminho.

Esta é a primeira vez que vi uma metodologia de desenvolvimento (desculpe Ken, um *framework*) funcionar diretamente de um livro. *Plug 'n Play*. Todos estamos felizes com isso – desenvolvedores, testadores, gerentes. Nos ajudou a sair de situações difíceis e nos permitiu manter o foco e momentum, apesar das severas turbulências de mercado e reduções de pessoal.

Eu não deveria dizer que estava surpreso, mas estava. Após digerir inicialmente alguns livros sobre *Scrum*, me parecia bom, mas quase tão bom para ser verdade (e todos nós conhecemos o ditado "quando alguma coisa parece muito boa para ser verdade..."). Então eu estava justificadamente um pouco cético. Mas após fazer *Scrum* por um ano, estou suficientemente impressionado (e a maioria das pessoas nas minhas equipes também) que continuarei a usar *Scrum* como padrão em novos projetos sempre que não haja uma forte razão para não usá-lo.

1

Introdução

Você está para começar a utilizar *Scrum* em sua empresa. Ou talvez você esteja usando *Scrum* há alguns meses. Conhece os princípios, leu alguns livros, ou talvez até já tenha feito a certificação de *scrum master*. Parabéns!

Mas ainda se sente confuso.

Como disse Ken Schwaber, *Scrum* não é uma metodologia, é um *framework*. O que significa que *Scrum* não vai te dizer exatamente o que fazer.

A boa notícia é que vou te dizer exatamente como faço *Scrum*, em um nível de detalhes excruciantemente doloroso. A má noticia é, bem, esta é a maneira como EU faço *Scrum*. E isto não significa que *Você* deva faze-lo exatamente da mesma maneira. De fato, eu também faria de maneira diferente, se encontrasse situações diferentes.

O melhor e o pior do *Scrum* é que você é forçado a adaptar o processo para sua situação especifica.

Minha abordagem atual para o *Scrum* é o resultado de um ano de experiências numa equipe de desenvolvimento de aproximadamente 40 pessoas. A empresa estava em uma situação difícil com tarefas levando mais tempo que o previsto, problemas de qualidade severos, constantes incêndios, deadlines estourados, etc. A empresa decidiu utilizar *Scrum* mas não completou a implantação, o que foi como minha tarefa. Para a maioria das pessoas na equipe de desenvolvimento naquela época, *Scrum* era uma *buzzword* que escutavam de vez em quando nos corredores, mas que não tinha nenhuma implicação para o trabalho diário.

Após um ano, implementamos *Scrum* em todas as camadas da empresa, tentamos diferentes tamanhos de equipes (3-12 pessoas), diferentes tamanhos de sprint (2-6 semanas), diferentes formas de definir “pronto”, diferentes formatos para os *product backlog* e de *sprint backlogs* (Excel, Jira, cartões), diferentes estratégias de testes, diferentes formas de realizar

apresentações de *sprint*, diferentes formas de sincronização de múltiplas equipes *Scrum*, etc. Também fizemos experimentos com práticas de XP – diferentes formas de realizar a build contínua, programação em pares, desenvolvimento orientado a testes, etc, e como combiná-los com *Scrum*.

Este é um processo de aprendizado contínuo, de modo que a estória não termina aqui. Estou convencido que esta empresa continuará aprendendo (se continuarem a realizar as retrospectivas dos *sprints*) e terem novos insights sobre como melhor implementar *Scrum* em seus contextos particulares.

Termo de Responsabilidade

Este documento não assume representar a "forma correta" de usar *Scrum*! Ele apenas representa uma forma de usar *Scrum*, o resultado da melhoria contínua ao longo de um ano. Você pode até mesmo decidir que nós entendemos tudo errado.

Tudo o que há neste documento reflete minhas próprias opiniões pessoais e subjetivas e não é de forma alguma uma declaração oficial da Crisp ou do meu cliente atual. Por esta razão eu intencionalmente evitei mencionar quaisquer produtos ou pessoas específicas.

Porque eu escrevi isso

Quando estava aprendendo *Scrum*, li os livros relevantes sobre *Scrum* e agile, acessei muitos sites e fóruns sobre *Scrum*, fiz a certificação do Ken Schwaber, perturbei-o com perguntas e passei muito tempo conversando com meus colegas. Uma das mais valiosas fontes de informação, entretanto, foram as histórias de guerra. As histórias de guerra transformam princípios e práticas em... bem... Como você realmente faz isso. Elas também me ajudaram a identificar (e algumas vezes evitar) erros comuns de iniciantes em *Scrum*.

Logo, essa é minha chance de devolver algo. Aqui está minha história de guerra.

Eu espero que este documento provoque algum retorno útil daqueles que estejam na mesma situação. Por favor, me iluminem!

Mas o que é Scrum?

Oh, me desculpe. Você é completamente novo no *Scrum* ou XP? Neste caso talvez você queira dar uma olhada nos seguintes links:

- http://agilemanifesto.org/
- http://www.mountaingoatsoftware.com/scrum
- http://www.xprogramming.com/xpmag/whatisxp.htm

Se você é impaciente demais para fazer isso, sinta-se livre para continuar lendo. Muito do jargão do *Scrum* é explicado conforme avançamos, logo ainda assim você poderá achar isso interessante.

2

Como fazemos product backlogs

O *product backlog* é o coração do *Scrum*. É aqui que tudo começa. O *product backlog* é basicamente uma lista de requisitos, estórias, Coisas que o cliente deseja, descritas utilizando a terminologia do cliente.

Nós as chamamos de *estórias*, ou algumas vezes apenas de *itens* do backlog.

Nossas estórias incluem os seguintes campos:

- **ID** – Uma identificação única, apenas um número com auto-incremento. Isso é para evitar que percamos o controle sobre as estórias quando nós mudamos seus nomes.
- **Nome** – Um nome curto e descritivo para a estória. Por exemplo, "Ver o histórico de transações". Suficientemente claro para que os desenvolvedores e o *product owner* entendam mais ou menos sobre o que estamos falando, e claro o bastante para distingui-la das demais estórias. Normalmente de 2 a 10 palavras.
- **Importância** – a pontuação de importância dessa estória para o *product owner*. Por exemplo 10. Ou 150. Mais pontos = mais importante.
    - Eu tento evitar o termo "prioridade" já que prioridade 1 é tipicamente interpretado como "prioridade mais alta", o que fica feio se mais tarde você decidir que algo é ainda mais importante. Qual pontuação de prioridade esse item deveria receber? Prioridade 0? Prioridade -1?
- **Estimativa inicial** – As estimativas iniciais da equipe sobre quanto tempo é necessário para implementar aquela estória, se comparada a outras estórias. A unidade é pontos por estória e geralmente corresponde mais ou menos a "relação homem/dias" ideal.
        1. Pergunte à equipe "se vocês puderem ter o número ideal de pessoas para esta estória (nem muitas, nem poucas, normalmnte duas), e se trancarem em uma sala cheia de comida e trabalharem sem distúrbio

algum, após quantos dias vocês apresentarão uma implementação pronta, demonstrável e testada? Se a resposta for "com 3 pessoas trancados em uma sala levará aproximadamente 4 dias" então a estimativa inicial é de 12 pontos por estória.

2. O importante não é ter estimativas absolutamente precisas (por exemplo, dizer que uma estória com 2 pontos deverá gastar 2 dias), mas sim obter estimativas relativas corretas (por exemplo, dizer que uma estória com 2 pontos gastará cerca da metade de uma estória com 4 pontos).

Como demonstrar – Uma descrição em alto nível de como a estória será demonstrada na apresentação do sprint. Isso é simplesmente uma simples especificação de teste. "Faça isso, então faça aquilo e então isso deverá acontecer."

- Se você pratica TDD (desenvolvimento orientado a testes) essa descrição pode ser usada como pseudo-código para o seu código de teste de aceitação.

- **Notas –** quaisquer outras informações, esclarecimentos, referências a outras fontes de informação, etc. Normalmente agil bem breve.

PRODUCT BACKLOG (exemplo)					
ID	**Nome**	**Imp**	**Est**	**Como demonstrar**	**Notas**
1	Depósito	30	5	Logar-se, abrir a página de depósito, depositar R$ 10,00, ir para a página do meu saldo e verificar que este aumentou em R$ 10,00.	Precisa de uma diagrama UML de sequência. Não é necessário se preocupar com criptografia por enquanto.
2	Verificar seu próprio histórico de transações	10	8	Logar-se, clicar em "transações". Fazer um depósito. Voltar para transações, verificar se o novo depósito é listado.	Usar paginação para evitar consultas muito grandes ao banco de dados. Projetar de forma similar à página de visualização de usuários.

Nós experimentamos com muitos outros campos, mas ao final do dia os seis campos acima eram os únicos realmente utilizados *sprint* após *sprint*.

Nós normalmente fazemos isso em um documento do Excel com compartilhamento habilitado (isso é, múltiplos usuários podem editá-lo simultaneamente). Oficialmente o *product owner* é o dono do documento, mas nós não queremos deixar os outros usuários de fora. Várias vezes um desenvolvedor desejará abrir o documento para esclarecer algo ou alterar uma estimativa.

Pela mesma razão, nós não mantemos este documento no repositório de controle de versões, nós o colocamos em um *drive* compartilhado na rede. Essa é a forma mais simples de permitir multiplos editores simultâneos sem causar conflitos de *lock* ou *merge*.

Quase todos os outros artefatos, entretanto, são colocados no repositório de controle de versões.

Campos adicionais da estória

Algumas vezes nós usamos campos adicionais no *product backlog*, principalmente como conveniência para ajudar o *product owner* na hora de decidir as priorizações.

- **Track** *(Tilha/Rastro)* – uma categorização básica dessa estória, por exemplo, "*back office*" ou "otimização". Dessa forma, o *product owner* pode facilmente filtrar por todos os itens de "otimização" e definir sua prioridade para baixa, etc.
- **Componentes** – Geralmente são incluídos campos na forma de "*checkboxes*" em um documento no Excel, por exemplo "base de dados, servidor, cliente". Aqui a equipe ou o *product owner* podem identificar quais componentes técnicos estão envolvidos na implementação dessa história. Isto é útil quando se tem várias equipes de *Scrum*, como por exemplo uma equipe de *back office* e outra de cliente, assim facilitando a decisão de cada equipe na escolha das estórias.
- **Solicitante** – o *product owner* pode querer manter o rastro de qual cliente ou o *stakeholder* que solicitou o item, para poder fornecer um *feedback* sobre o progresso desse item.
- **ID do bug** – se houver um sistema separado para registro de erros (*bug tracking*), como nós fazemos com o Jira, é útil manter a rastreabilidade da estória com um ou mais erros reportados sobre ela.

Como nós mantemos o product backlog a nível de negócio

Se o *product owner* tem uma formação técnica, ele pode adicionar estórias como "Adicionar índice na tabela de eventos". Por que ele quer algo assim? O verdadeiro objetivo implícito é provavelmente algo como "acelerar o formulário de pesquisa de eventos no *back office*".

Pode ocorrer que os índices não sejam os gargalos que causam a lentidão no formulário. Pode ser algo completamente diferente. A equipe é normalmente melhor adaptada para descobrir *como* resolver algo, assim o *product owner* deveria focar-se nos objetivos de negócio.

Quando vejo estórias com orientação técnica como essa, normalmente eu pergunto ao *product owner* uma série de questões de "mas *por quê*", até encontrar o objetivo subjacente. Então nós reformulamos a estória nos termos do objetivo subjacente ("acelerar o formulário de pesquisa de eventos no *back office*"). A descrição técnica original acaba virando uma nota ("Indexar a tabela de eventos poderia resolver isso").

3

Como nos preparamos para o planejamento do sprint

Está bem, o dia de planejamento do *sprint* está chegando rápido. Uma lição que aprendemos sempre é:

Lição: Tenha certeza de que o *product backlog* está organizado e fechado *antes* da reunião de planejamento do sprint.

E o que *isso* significa? Que todas as estórias estão perfeitamente bem definidas? Que todas as estimativas estão corretas? Que todas as prioridades devem permanecer fixas? Não, não e não! O que isso significa é:

- O *product backlog* deveria existir! (já pensou nisso?)
- Deveria haver *apenas um product backlog* e *apenas um product owner* (por produto).
- Todos os itens importantes deveriam ter uma escala de importância associada a eles. Cada um com a sua escala diferente da dos outros.
    - Na verdade, não há problema se todos os itens de pouca importância tiverem a mesma escala, já que eles não serão o foco da reunião de planejamento.
    - Qualquer estória que o *product owner* acredite ter a mais remota possibilidade de ser incluída no próximo *sprint*, deve ter uma escala única de importância.
    - O nível de importância só é usado para classificar os itens por importância. Assim, se o item A tem importância de 20 e o item B 100, significa apenas que B é mais importante que A. Não significa que B é cinco vezes mais importante do que A. Se B tivesse com a escala 21, significaria a mesma coisa!
    - É útil deixar espaços vazios entre as numerações para poder inserir novos itens. Supondo que o item C seja descoberto como mais importante do que A, mas menos importante do que B, ele poderia ganhar a escala 30, ao

invés de 20,5. Assim, deixe os espaços. É bem mais prático.

- O *product owner* deveria *entender* cada estória (normalmente ele é o autor, mas em alguns casos outras pessoas adicionam requisitos, mas o *product owner* ainda precisa priorizá-los). Ele não precisa conhecer exatamente o que é necessário para implementar, mas ele deveria entender o porquê dessa estória estar ali.

Nota: Outras pessoas além do *product owner* podem adicionar estórias ao *product backlog*. Mas elas não podem associar uma escala de importância. Esse é um papel exclusivo do *product owner*. Eles também não podem estimar prazos. Essa é uma tarefa exclusiva da equipe.

A seguir, outras abordagens que nós tentamos ou avaliamos:

- Usar Jira (nosso sistema de *bug tracking*) para hospedar o *product backlog*. A maioria de nossos *product owners* acharam que era preciso dar muitos cliques para realizar as tarefas. Excel é bom e fácil de manusear. Você pode usar cores facilmente, reordenar os itens, adicionar novas colunas quando necessário, adicionar notas, importar e exportar dados, etc.
- Usar uma ferramenta de suporte a processos ágeis, como VersionOne, ScrumWorks, XPlanner, etc. Ainda não testamos nenhuma delas, mas provavelmente faremos isso no futuro.

4

Como planejamos sprint

O planejamento de sprint é uma reunião crítica, provavelmente o evento mais importante no *Scrum* (na minha opinião, claro). Um encontro de planejamento de *sprint* mal feito pode bagunçar totalmente um *sprint*.

O propósito da reunião de planejamento do *sprint* é dar à equipe informação suficiente para trabalharem em paz por algumas semanas, e para dar ao *product owner* confiança suficiente para deixá-los fazerem isto.

OK, isto foi confuso. O resultado concreto do encontro de planejamento de *sprint* é:

- Um objetivo de *sprint*.
- Uma lista de membros da equipe (e seus níveis de comprometimento, se não 100%).
- Um *sprint backlog* (= uma lista de estórias inclusas no *sprint*).
- Uma data definida para a apresentação do *sprint*.
- Data e local definidos para a reunião diária.

Por que o product owner deve participar

Às vezes, os *product owners* relutam em despender horas com a equipe fazendo planejamento do *sprint*. "Pessoal, eu já listei o que eu quero. Eu não tenho tempo para estar na sua reunião de planejamento". Isto é um problema muito grave.

A razão pela qual toda equipe *e* o *product owner* devam estar na reunião de planejamento do *sprint* é porque cada estória contém três variáveis que são muito dependentes umas das outras.

Escopo e importância são definidos pelo *product owner*. Estimativa é definida pela equipe. Durante uma reunião de planejamento do *sprint*, estas três variáveis são refinadas continuamente por diálogo cara-a-cara entre equipe e *product owner*.

Normalmente, o *product owner* inicia a reunião resumindo seu objetivo para o *sprint* e as estórias mais importantes. Em seguida, a equipe toma a frente e estima o tempo de cada estória, começando pela mais importante. Ao fazer isto, eles vão se deparar com questões de escopo importantes – "esta estória 'apagar usuário' inclui passar por cada transação pendente para o usuário e cancelar cada uma?" Em alguns casos, as respostas serão surpreendentes para a equipe, fazendo com que eles mudem suas estimativas.

Em alguns casos, a estimativa de tempo para uma estória não será aquela que o *product owner* esperava. Isto poderá fazê-lo mudar a importância da estória. Ou mudar o escopo da estória, que por sua vez fará a equipe re-estimar, etc., etc.

Este tipo de colaboração direta é fundamental para o *Scrum* e, de fato, todo o desenvolvimento ágil de software.

E se o *product owner* ainda insistir que ele não tem tempo para tomar parte das reuniões de planejamento do *sprint*? Eu normalmente tento uma das estratégias, na seguinte ordem:

- Tento ajudar o *product owner* a entender por que a sua participação direta é crucial e espero que ele mude de idéia.
- Tento fazer com que alguém da equipe se voluntarize a ser o intermediador do *product owner*. Diga ao *product owner* "Como você não pode participar de nossa reunião, nós deixaremos Jeff representá-lo como um intermediador. Ele terá plenos poderes para mudar as prioridades e escopo das estórias na sua ausência durante a reunião. Eu sugiro que você sincronize com ele o máximo possível antes da reunião. Se você não quiser que Jeff seja o seu intermediador, por favor sugira alguma outra pessoa,

que possa se juntar a nós durante toda a duração da reunião."
- Tente convencer a equipe de gerenciamento a designar um novo *product owner*.
- Adie o lançamento do *sprint* até que o *product owner* ache tempo para se juntar à reunião. Enquanto isto, se recuse a entregar qualquer coisa. Deixe a equipe passar cada dia fazendo aquilo que sentir ser mais importante para o dia.

Por que qualidade não é negociável

No triângulo acima, eu intencionamente evitei a quarta variável *qualidade*.

Eu tento distinguir entre *qualidade interna* e *qualidade externa*:

- *Qualidade externa* é o que é percebido pelos usuários do sistema. Uma interface de usuário lenta e não intuitiva é um exemplo de baixa qualidade externa.
- *Qualidade interna* refere-se a questões que normalmente não são visíveis ao usuário, mas que têm profundos efeitos na manutenibilidade do sistema. Coisas como consistência do projeto do sistema, cobertura dos testes, facilidade de leitura do código, *refactoring* etc.

No geral, um sistema com alta qualidade interna pode ainda ter uma baixa qualidade externa. Entretanto, um sistema com baixa qualidade interna raramente terá uma qualidade externa alta. É difícil construir algo legal a partir de fundações podres.

Eu trato qualidade externa como parte do escopo. Em alguns casos pode fazer perfeito sentido, negocialmente, lançar uma versão do sistema que tenha uma interface deselegante e lenta, e lançar uma versão melhorada posteriormente. Eu deixo o dilema para o *product owner*, uma vez que ele é responsável por determinar o escopo.
Qualidade interna, porém, não está em discussão. É responsabilidade da equipe manter a qualidade do sistema sob todas as circunstâncias e isso simplesmente não é negociável. Nunca.

(Bem, OK, *quase* nunca)

Então, como mostramos a diferença entre problemas de qualidade interna e problemas de qualidade externa?

Digamos que o *product owner* diga “OK pessoal, eu respeito sua estimativa de tempo de seis pontos de estória, mas eu tenho certeza de que vocês podem fazer algum tipo de quebra-galho para isso na metade do tempo se vocês se concentrarem nisso.”

A-ha! Ele está tentando usar qualidade interna como uma variável. Como eu sei? Porque ele quer que nós reduzamos a estimativa da estória sem “pagar o preço” de reduzir o escopo.. A palavra “quebra-galho” deve ativar um alarme na sua cabeça...

E por que nós não permitimos isso?

Minha experiência é que sacrificar qualidade interna é quase sempre uma idéia muito, muito ruim. O tempo economizado é largamente superado pelo custo, tanto em curto quanto em longo prazo. Uma vez que se permita que uma base de código se deteriore, é muito difícil recuperar a qualidade mais tarde.

Ao invés disso, eu tento levar a discussão para o escopo. “Já que conseguir essa *feature* mais cedo é importante para você, podemos reduzir o escopo de modo que seja mais rápido implementá-la. Talvez possamos simplificar o tratamento de erros e fazer 'Tratamentos de erros avançados' numa estória separada, que reservamos para o futuro. Ou podemos reduzir a prioridade das outras estórias, de modo que possamos focar nessa. Que tal?”

Uma vez que o *product owner* tenha aprendido que qualidade interna não é negociável, ele geralmente se especializa em manipuar as outras variáveis.

Reuniões de planejamento do sprint que se arrastam sem fim

O mais difícil em reuniões de planejamento de *sprint* é o fato de:
1) As pessoas pensarem que elas não irão demorar tanto…
2) ... mas elas demoram!

Tudo no *Scrum* é delimitado pelo tempo. Eu amo essa regra, simples e consistente. Tentamos nos ater a ela.

Então, o que devemos fazer quando o prazo estipulado para a reunião de planejamento do *sprint* está prestes a se esgotar e ainda não temos nem sinal de um objetivo do ou de um *sprint backlog*? Interrompemos a reunião?? Ou melhor estendermos a discussão por mais uma hora? Ou suspendemos temporiamente e continuamos no dia seguinte?

Isso acontece várias vezes, especialmente em equipes novatas no *framework*. Então, o que você faz? Eu não sei. Mas o que podemos fazer? Bem, geralmente eu interrompo a reunião abruptamente. Finalizo-a. Deixa o *sprint* sofrer as consequências. Mais especificamente, eu digo à equipe e ao *product owner*: "então, esta reunião terminará em 10 minutos. Nós não temos ainda o planejamento do *sprint*. Vamos trabalhar com o que temos ou vamos agendar outra reunião de 4 horas para amanhã, a partir das 8 horas da manhã?". Adivinha o que irão escolher… :o)

Já tentei deixar a reunião rolar solta, extrapolando o prazo previsto. Geralmente, não resolve nada, pois as pessoas já estão cansadas. Se a equipe não conseguiu produzir um planejamento decente para o *sprint* em 2 ou 8 horas (ou seja lá qual for a duração escolhida para sua reunião), provavelmente 1 hora a mais não irá resolver em nada o problema. A proposta de agendar uma nova reunião para o dia seguinte, seria bem razoável, se não fosse o fato de as pessoas geralmente já estarem impacientes para iniciar logo o *sprint* e não quererem perder mais tempo com horas de planejamento.

Por isso, minha decisão é de interromper mesmo a reunião. Deixa o *sprint* sofrer com isso. O lado bom disso é que a equipe aprende uma boa lição, e a próxima reunião de planejamento do *sprint* será bem mais eficiente. Além disso, as pessoas serão menos resistentes quando você propor um prazo para a reunião que outrora teria sido considerado muito longo.

Aprenda a respeitar a delimitação do tempo e aprenda a defini-la de forma realística. Isso se aplica tanto para prazos de reuniões quanto para prazos de *sprints*.

Agenda da reunião de planejamento do sprint

Possuir algum tipo de cronograma para a reunião de planejamento do *sprint* reduzirá o risco de ultrapassar o espaço de tempo previsto.

Eis um exemplo de um cronograma típico.

Reunião de planejamento do sprint: 13:00 – 17:00 (10 minutos de intervalo a cada hora)

- **13:00 – 13:30.** O *product owner* repassa o objetivo do *sprint* e sumariza o *product backlog*. Definir lugar, data e hora da demonstração.
- **13:30 – 15:00**. A equipe estima o tempo e quebra os ítens conforme necessário. O *product owner* atualiza as taxas de importância conforme necessário. Os ítens são esclarecidos. "Como demonstrar" é preenchido para todos os ítens de maior importância.
- **15:00 – 16:00**. A equipe escolhe as estórias a serem incluídas no *sprint*. Calculam a velocidade para verificar se o planejamento ficou realista.
- **16:00 – 17:00**. Escolher data e lugar para a reunião diária (se não forem os mesmo do último *sprint*). Complementar a quebra de estórias em tarefas.

Esse cronograma não precisa, de forma alguma, ser seguido à risca. O *scrum master* pode aumentar ou diminuir os tempos conforme necessário, ao longo da reunião.

Definindo o tamanho do sprint

Uma das saídas da reunião de planejamento do *sprint* é uma data definida para a apresentação. Isso significa que vocês precisam definir o tamanho do *sprint*.

Então, qual seria um bom tamanho de *sprint*?

Bem, *sprints* curtos são bons. Eles permitem que a equipe seja "ágil", isto é, mude de direção freqüentemente. *Sprints* curtos = ciclo curto de *feedback* = entregas mais freqüentes = *feedback* mais freqüente do cliente = menos tempo perdido, indo na direção errada = aprender e melhorar rápido, etc.

Entretanto, *sprints* longos são bons também. A equipe tem mais tempo para ganhar ritmo, ela tem mais espaço para se recuperar dos problemas, e conseguir atingir o objetivo do sprint, você tem menos *overhead* em termos de reuniões de planejamento, apresentações, etc.

No geral, *product owners* gostam de sprints curtos e desenvolvedores preferem os longos. Então o tamanho do *sprint* representa um

compromisso. Nós experimentamos bastante isso, e chegamos ao nosso tamanho favorito: 3 semanas. A maioria das nossas equipes (mas não todas) faz *sprints* de 3 semanas. Curtas o bastante para nos dar agilidade corporativa adequada, longas o bastante para a equipe conseguir fluidez e se recuperar de eventuais problemas durante o *sprint*.

Uma coisa que concluímos foi: *faça* experimentos com o tamanho do sprint, no início. Não perca muito tempo *analisando*, apenas escolha um tamanho decente, e dê-lhe uma chance por um *sprint* ou dois, então mude o tamanho.

No entanto, uma vez que vocês decidiram o tamanho que mais gostam, *mantenham ele* por um bom tempo. Depois de alguns meses de experimentação, nós achamos que 3 semanas estava bom. Então nós fazemos *sprints* de 3 semanas, ponto. Algumas vezes parecerá um pouco longo demais, outras vezes parecerá um pouco curto demais. Mas, mantendo o mesmo tamanho, isso se torna como um batimento cardíaco corporativo, com que todos se identificam confortavelmente. Não há discussão quanto às datas de entregas ou coisas do tipo, porque todos sabem que a cada três semanas haverá uma entrega, ponto.

Definindo o objetivo do sprint

Acontece quase toda vez. Em algum momento durante o planejamento do *sprint*, eu pergunto "então, qual é o objetivo deste *sprint*?" e todo mundo fica parado me olhando e o *product owner* franze a sobrancelha e coça o queixo.

Por algum motivo é difícil definir um objetivo para o *sprint*. Mas ainda sim eu percebi que compensa se esforçar para espremer um. Melhor um objetivo meia-boca do que nenhum objetivo. O objetivo poderia ser "ganhar mais dinheiro" ou "completar as 3 estórias mais importantes" ou "impressionar o CEO" ou "tornar o sistema bom o suficiente para distribuir uma versão beta" ou "adicionar funcionalidades básicas de retaguarda" ou qualquer coisa. O importante é que seja em termos de negócio, não termos técnicos. Isto quer dizer, em termos que as pessoas de fora da equipe possam entender.

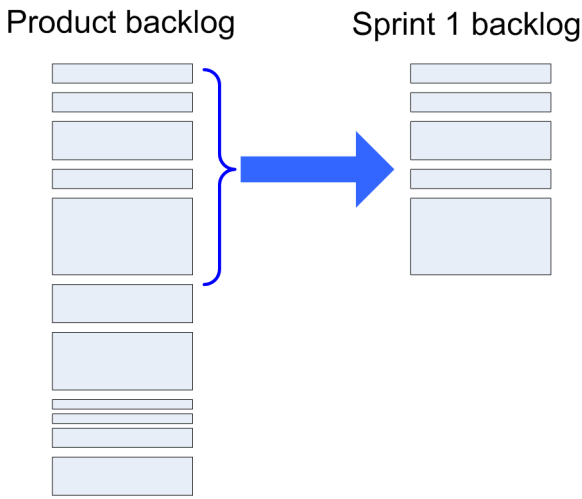
O objetivo do *sprint* deve responder a pergunta fundamental "Por que nós estamos fazendo este *sprint*? Porque não saímos de férias ao invés de fazê-lo?". De fato, uma maneira de extrair o objetivo de um *sprint* do *product owner* é literalmente fazer esta pergunta.

O objetivo deve ser algo que já não tenha sido atingido. “Impressionar o CEO” pode ser um objetivo ótimo, mas não se ele já está impressionado com o sistema atualmente. Neste caso, todos poderiam ir embora para a casa que mesmo assim o objetivo seria atingido.

O objetivo do *sprint* pode parecer bobo e artificial durante o planejamento do sprint, mas freqüentemente se torna útil no meio dele, que é quando as pessoas começam a ficar confusas sobre o que elas deveriam estar fazendo. Se você tem várias equipes *Scrum* (como nós temos) trabalhando em produtos diferentes, é muito útil poder listar os objetivos de todas as equipes em uma única página *wiki* (ou o que seja) e colocá-la em um local evidente para que todos na companhia (não somente a gerência) saibam o que a empresa está fazendo – e por quê!

Decidindo quais estórias incluir no sprint

Uma das principais atividades da reunião de planejamento do *sprint* é decidir quais estórias incluir no *sprint*. Mais especificamente, quais estórias do *product backlog* copiar para o *sprint backlog*.



Observe a figura acima. Cada retângulo representa uma estória, ordenado pela importância. A estória mais importante está no topo da lista. O tamanho de cada retângulo representa o tamanho daquela estória (ou seja, o tempo estimado nos pontos de estória). A altura do braço azul representa a *velocidade estimada* da equipe, ou seja, quantos pontos de estória a equipe acredita poder completar durante o próximo *sprint*.

O *sprint backlog* da direita é um snapshot das estórias do *product backlog*. Ele representa a lista de estórias que a equipe executará para o *sprint*.

A *equipe* decide como as muitas estórias serão incluídas no *sprint*. Não o *product owner* ou qualquer outra pessoa.

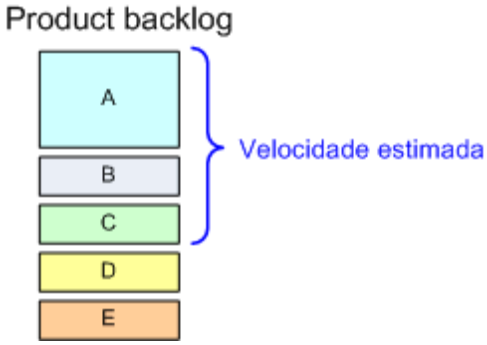
Isso levanta duas questões:

1. Como fazer a equipe decidir quais estórias incluir no *sprint*?
2. Como pode o *product owner* afetar a decisão dela?

Eu começarei com a segunda questão.

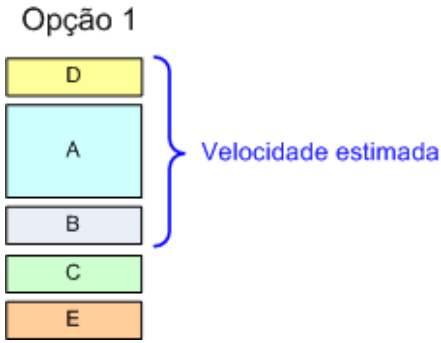
Como o product owner afeta quais estórias estarão neste Sprint?

Digamos que nós tenhamos a seguinte situação durante uma reunião de planejamento de sprint.

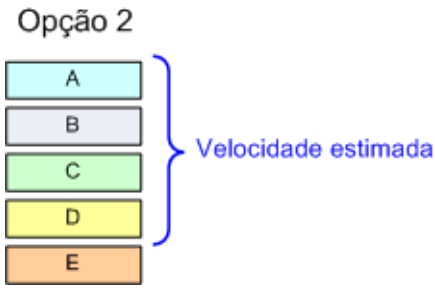


O *product owner* está desapontado com o fato de que a estória D não será incluída no *sprint*. Quais são as suas opções durante a reunião de planejamento do *sprint*?

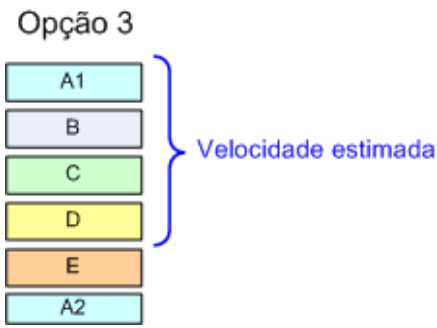
Uma opção é reordenar a priorização. Se ele der à estória D a maior prioridade, a equipe será obrigada a adicioná-la ao *sprint* (neste caso tirando a estória C).



Uma segunda opção é mudar o escopo – reduzir o escopo da estória A até que a equipe acredite que a estória D caberá no *sprint*.



Uma terceira opção é dividir uma estória. O *product owner* pode decidir que existem aspectos na estória A que não são realmente importantes, então ele divide a estória A em A1 e A2 separando-as em prioridades diferentes.



Como vê, mesmo que o *product owner* não possa alterar a velocidade estimada, há várias formas como este pode influenciar quais estórias estarão presentes num sprint.

Como a equipe decide que estórias incluir no sprint?

Utilizamos duas técnicas para isso:

1. No sentimento/instinto
2. Cálculo da Velocidade

Estimativa usando o sentimento ou instinto

- **Scrum master**: "Olá pessoal, podemos terminar a estória A neste Sprint? (apontando para o item mais importante no *backlog* de produtos)
- **Lisa**: "Dã. Claro que podemos. Temos 3 semanas, e é uma funcionalidade bastante trivial"
- **Scrum master**: "OK, e se incluirmos a estória B também? (apontando para o Segundo item mais importante)
- **Tom & Lisa em uníssono**: "Continua fácil"
- **Scrum master**: "OK, e que tal as estórias A, B e C?
- **Sam (ao cliente)**: "a estória C inclui os tratamentos avançados de erros?
- **Cliente**: "não, podemos ignorar por enquanto, somente implemente o tratamento de erros básicos"
- **Sam**: "então C pode entrar também?
- **Scrum master**: "OK, e se incluirmos a estoria D?
- **Lisa**: "Hmm...."
- **Tom**: "acredito que podemos fazê-lo"
- **Scrum master**: "90% certo? 50%?"
- **Lisa & Tom**: "Mais pra 90%".
- **Scrum master**: "OK, então D entra também. Que acham de incluirmos a estória E?
- **Sam**: "Talvez."
- **Scrum master**: "90%? 50%?"
- **Sam**: "Perto de 50%".
- **Lisa**: "Estou na dúvida."
- **Scrum master**: "OK, então vai ficar de fora. Nos comprometemos com A, B, C e D. Se pudermos, terminaremos E também, mas ninguém poderá contar com isso, assim deixaremos fora do plano do *sprint*. Que acham disso?
- **Todos**: "OK!"

O sentimento ou instinto funciona muito bem para pequenas equipes e sprint curtos.

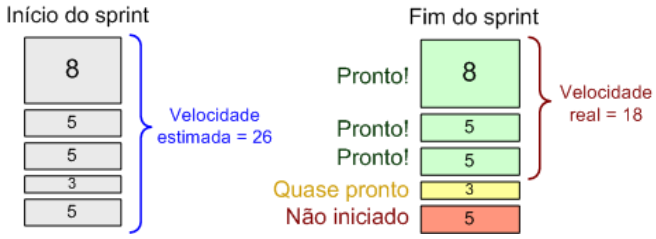
Estimativas usando cálculos de velocidade

Essa técnica envolve dois passos:

1. Defina a *velocidade estimada.*
2. Calcular quantas estórias você pode adicionar sem exceder a velocidade estimada.

Velocidade é uma medida da "quantidade trabalho realizado", onde cada item é medido com base na sua estimativa inicial.

A figura abaixo mostra um exemplo de *velocidade estimada* no início de um sprint e a *velocidade real* no final do *sprint*. Cada retângulo é uma estória, e o número dentro dele é a estimativa inicial dela.



Note que a velocidade real é baseada na estimativa *inicial* de cada estória. Quaisquer modificações na estimativa de tempo da estória durante o sprint devem ser ignoradas.

Já posso ouvir você dizendo: "Como isso é útil? Maior ou menor velocidade depende de uma série de fatores! Programadores inexperientes, estimativas iniciais incorretas, diminuição de escopo, problemas não previstos durante o sprint, etc.!"

Eu concordo, esse é um número cru mesmo. Mas ainda assim é uma medida útil, especialmente quando não for comparado com nada mais. Ele te fornece alguns fatos fixos. "Independentemente dos motivos, aqui está uma diferença aproximada de o quanto nós achávamos que gastaríamos na atividade e o quanto realmente gastamos".

E o que dizer sobre uma estória que ficou *quase* completa durante o sprint? Por que não pegamos medidas parciais em nossa velocidade real? Bem, é para tirar proveito do fato de que *Scrum* (e de fato o desenvolvimento ágil de software e os processos *lean* de fabricação em geral) trata de ter as tarefas realizadas, prontas para serem entregues! O valor de uma coisa feita pela metade é zero (de fato, é até negativo). Leia

o livro "*Managing the Design Factory*" de Donald Reinertsen ou algum dos livros de Poppendieck para saber mais sobre o assunto.

Então, com que varinha mágica estimamos velocidade?

Uma maneira muito simples de fazer isso é olhar para o histórico da equipe. Qual foi a velocidade dela nos sprints mais recentes? Então assuma que a velocidade nesse *sprint* será equivalente.

Essa técnica é conhecida como *tempo de ontem*. Isso só é possível para equipes que já tenham terminado alguns *sprints* (com as estatísticas disponíveis) e farão o próximo praticamente da mesma forma com o mesmo número de membros e mesmas condições de trabalho, etc. Obviamente, nem sempre esse é o caso.

Uma variante mais sofisticada é fazer um cálculo simples de recursos. Vamos supor que estamos planejando um sprint de 3 semanas (15 dias de trabalho) com uma equipe de 4 pessoas. Lisa estará de folga por 2 dias. Dave está disponível somente 50% e estará de folga por 1 dia. Colocando tudo isso junto...

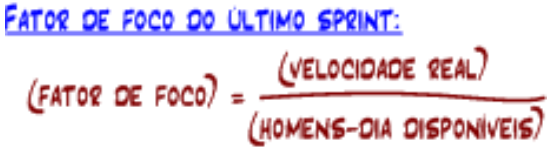
... resulta em 50 homem-dias disponíveis para este *sprint*.

Será esta a nossa velocidade estimada? Não! Porque nossa unidade de estimativa é pontos por estória, os quais, no nosso caso, correspondem a uma aproximação de "homens-dia ideal". Um homem-dia ideal é um dia perfeitamente efetivo, sem distúrbios, o que é raro. Além disso, temos que levar em consideração coisas como o trabalho não planejado que é adicionado ao *sprint*, pessoas que ficam doentes, etc.

Portanto, nossa velocidade estimada será certamente menor que 50. Mas o quanto menor? Utilizamos para isso o termo "fator de foco".

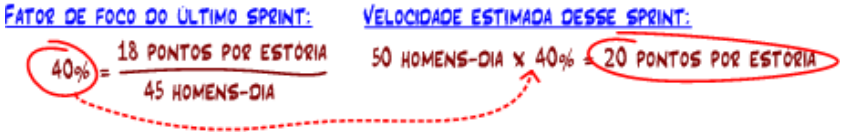
O fator de foco é uma estimativa de como a equipe é focada. Um fator de foco baixo, pode significar que a equipe espera ter muitas interferências ou percebe que suas próprias estimativas de tempo são otimistas.

A melhor maneira para determinar um fator de foco razoável é considerar o último *sprint* (ou melhor ainda, a média de alguns *sprints* anteriores).

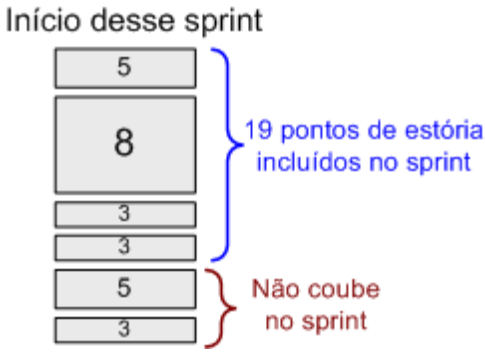


A *velocidade atual* é a soma das estimativas iniciais de todas as estórias que foram finalizadas no último *sprint*.

Vamos supor que o último *sprint* terminou 18 pontos por estória utilizando uma equipe de 3 pessoas, com Tom, Lisa e Sam trabalhando 3 semanas, resultando em um total de 45 homens-dia. E agora nós estamos tentando calcular nossa velocidade estimada para o próximo *sprint*. Para complicar as coisas, um cara novo, o Dave, está se juntando à equipe para esse sprint. Levando em consideração as folgas e as obstruções nós temos 50 homem-dias para o próximo *sprint*.



Portanto, nossa velocidade estimada para o próximo *sprint* é de 20 pontos por estória. O que significa que a equipe deveria adicionar estórias ao *sprint* até atingir uma soma de aproximadamente 20.



Neste caso, a equipe talvez escolha as 4 primeiras estórias para obter um total de 19 pontos por estória, ou as 5 primeiras totalizando 24 pontos por estória. Vamos supor que eles escolham 4 estórias, visto que é o mais próximo de 20 pontos por estória que se chega. Na dúvida, selecione menos estórias.

Como estas 4 estórias adicionaram 19 pontos por estória, a velocidade estimada final para este *sprint* é 19.

Tempo de ontem é uma técnica acessível, mas use-a com uma dose de bom senso. Se o último *sprint* foi excepcionalmente ruim porque a maior parte da equipe esteve doente por uma semana, então pode ser seguro assumir que você não terá esta falta de sorte de novo e você pode estimar um fator de foco alto para o próximo *sprint*. Se a equipe instalou recentemente um novo e rápido sistema de *build* contínuo você pode, provavelmente, aumentar o fator de foco devido a isto também. Se uma pessoa nova está se juntando a este *sprint* você precisa reduzir o fator de foco para considerar seu treinamento. Etc.

Sempre que possível olhe para os *sprints* passados e faça a média dos números para obter uma estimativa mais confiável.

E se a equipe for completamente nova e você não tiver nenhuma estatística? Olhe para o fator de foco de outras equipes em circunstâncias similares.

E se você não tiver nenhuma outra equipe para olhar? Suponha um fator de foco. A boa notícia é que o seu chute será usando somente no primeiro *sprint*. Depois disto você terá estatísticas e pode medir e melhorar continuamente seu fator de foco e a velocidade estimada.

O fator de foco "padrão" que eu uso para novas equipes é usualmente 70%, porque este é o ponto onde muitos de nossas outras equipes chegaram com o tempo.

Qual técnica de estimativa nós usamos ?

Eu mencionei diversas técnicas acima – instinto/sentimento, cálculo de velocidade baseado no tempo de ontem, e cálculo de velocidade baseada no homens-dia disponíveis e fator de foco.

Então, qual técnica nós usamos ?

Geralmente combinamos estas técnicas em certas medidas. Realmente não toma muito tempo.

Olhamos para o fator de foco e na velocidade atual do último *sprint*. Olhamos para a disponibilidade total de nossos recursos disponíveis e estimamos o fator de foco. Discutimos quaisquer diferenças entre estes fatores de foco e fazemos os ajustes necessários.

Uma vez que temos a listagem preliminar das estórias a serem incluídas no *sprint*, eu faço uma revisão baseada no meu sentimento. Eu peço a equipe para ignorar os números por um momento e pensar apenas em suas *percepções* sobre a parte em questão do *sprint*. Se acham que levará bastante tempo então removemos uma ou duas estórias. E vice-versa.

Ao final do dia, o objetivo é simplesmente decidir quais estórias nós incluiremos no *sprint*. Fator de foco, disponibilidade de recursos e velocidade estimada são apenas meios de atingir o objetivo.

Por que nós usamos cartões

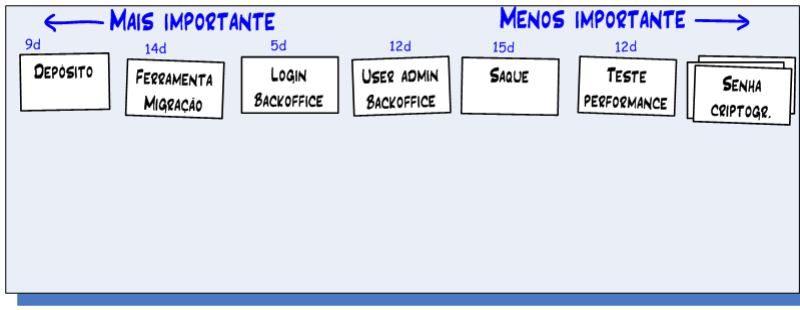
A maior parte da reunião de planejamento do *sprint* é gasto lidando com as estórias no *product backlog*. Estimando-as, repriorizando-as, esclarecendo-as, quebrando-as, etc.

Como fazemos isso na prática ?

Bem, normalmente, as equipes ligam o projetor e mostram um *backlog* feito em Excel, então um cara (tipicamente o *product owner* ou o *scrum master*) pega o teclado, esboça cada estória e estimula uma discussão. A medida que as prioridades e detalhes são discutidas pela equipe e pelo *product owner*, a pessoa ao teclado atualiza a estória direto no Excel.

Parece bom? Bem, não é. Normalmente é um saco. E o que é pior, a equipe normalmente não avisa que é um saco até eles chegarem no fim da reunião e perceberem que ainda não conseguiram passar para lista de estórias!

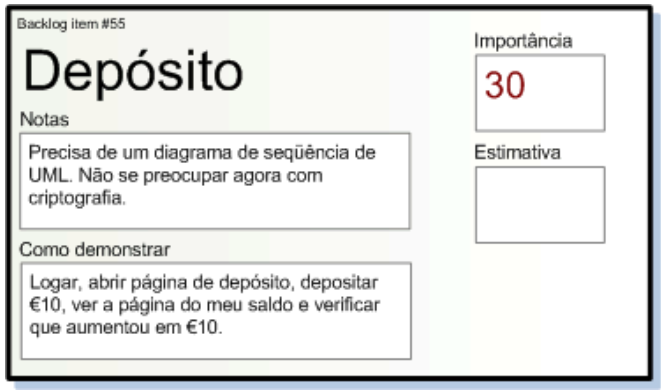
Uma solução que funciona muito melhor é criar cartões e colocá-los na parede (ou numa mesa grande).



Esta é uma interface de usuário superior comparada a computador & projetor porque:

- Pessoas ficam em pé e caminham => elas ficam acordadas e alerta por mais tempo.
- Todos se sentem mais pessoalmente envolvidos (ao invés de só o cara com o teclado).
- Múltiplas estórias podem ser editadas simultaneamente.
- A repriorização é trivial - só trocar a posição dos cartões.
- Após a reunião, os cartões podem ser levados para a sala da equipe e ser usado como um quadro de tarefas na parede (veja pg 49 "Como nós fazemos o *sprint backlog*)

Você pode tanto escrevê-los à mão ou (como geralmente fazemos), quanto usar um *script* simples para gerar cartões imprimíveis diretamente do *product backlog*.



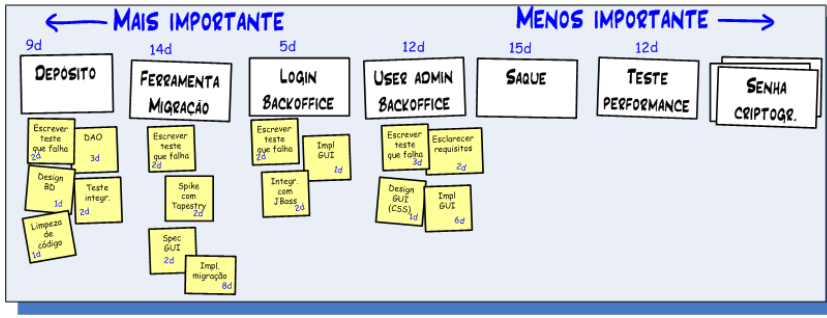
PS – o *script* está disponível no meu blog em http://blog.crisp.se/henrikkniberg.

Importante: Depois da reunião de planejamento do *sprint*, nosso *scrum master* atualiza manualmente o *product backlog* no Excel com respeito a quaisquer mudanças que foram feitas aos cartões físicos de estórias. Sim, isso é um pequeno inconveniente administrativo, mas nós achamos perfeitamente aceitável considerando o quão mais eficiente é a reunião de planejamento do *sprint* com cartões físicos.

Uma nota sobre o campo "Importância". Essa é a importância como especificado no *product backlog* do Excel no momento da impressão. Tê-lo no cartão torna fácil ordenar os cartões fisicamente por importância (normalmente colocamos os itens mais importantes à esquerda, e os menos à direita). Entretanto, uma vez que os cartões estão na parede, você pode ignorar a classificação de importância e usar a ordenação física na parede para indicar importância relativa. Se o *product owner* troca dois itens, não perca tempo atualizando a classificação de importância no papel. Certifique-se de atualizar a classificação de importância no *product backlog* do Excel somente depois da reunião.

Estimativas de tempo são mais fáceis (e mais precisas) de fazer se a estória é quebrada em tarefas. Nós usamos o termo "atividade" pois a palavra "tarefa" significa algo completamente diferente em sueco :o)
Isso também é bom e fácil de fazer com nossos cartões. Você pode dividir a equipe em pares e cada um quebra uma estória, em paralelo.

Fisicamente nós fazemos isso adicionando pequenas notas em post-it embaixo de cada estória, cada post-it refletindo uma tarefa dentro daquela estória.



Nós não atualizamos o *product backlog* do Excel com respeito a nossas quebras em tarefas por duas razões:

- A quebra de tarefas é geralmente bastante volátil, i.e. elas são freqüentemente alteradas e refinadas durante o sprint, então dá muito trabalho manter *product backlog* do Excel sincronizado.
- O *product owner* não precisa estar envolvido neste nível de detalhes, mesmo.

Assim como com os cartões de estórias, os post-its de quebra de tarefas podem ser reusados diretamente no *sprint backlog* (veja pág. 49 “Como fazemos os *sprint backlogs*”).

Definição de “pronto”

É importante que o *product owner* e a equipe concordem com uma definição clara de “pronto”. Uma estória está completa quando todo o código está no repositório? Ou está completa apenas quando foi feito *deploy* em um ambiente de teste e a estória foi verificada por uma equipe de testes de integração? Sempre que possível nós usamos a definição “pronto para *deploy* em produção”, mas algumas vezes precisamos usar a

definição “feito *deploy* em servidor de teste e pronto para os testes de aceitação”.

No início costumávamos ter *checklists* detalhados para isso. Atualmente nós frequentemente dizemos “uma estória está pronta quanto o tester da equipe *Scrum* diz que está”. É responsabilidade do tester se certificar de que a intenção do *product owner* foi compreendida pela equipe e que o item está suficientemente “pronto” para que passe pela definição de “pronto” acordada.

Nós percebemos que nem todas as estórias podem ser tratadas da mesma forma. Uma estória com o nome “Formulário para pesquisa de usuários” será tratada de forma muito diferente de uma estória com o nome “Manual de operações”. No segundo caso, a definição de “pronto” pode simplesmente significar “aceito pela equipe de operações”. É por isso que o senso comum geralmente é melhor do que *checklists* formais.

Se você frequentemente se confunde quanto à definição de “pronto” (o que ocorreu conosco no início) você provavelmente deveria ter um campo “definição de pronto” para cada estória individual.

Estimativa de tempo usando planning poker

Estimar é uma atividade da equipe – cada membro é freqüentemente envolvido pra estimar cada estória. Por que?

- Quando vamos planejar, normalmente nós não sabemos exatamente quem vai implementar quais partes de quais estórias.
- Estórias normalmente envolvem diversas pessoas e diversos tipos de expertise (design de interface de usuário, codificação, teste, etc).
- Para prover uma estimativa, o membro da equipe precisa de algum tipo de entendimento do quê trata a estória. Pedindo para todos estimarem cada item, nós nos certificamos que cada membro da equipe compreende do que cada item se trata. Isso aumenta a probabilidade de que os membros se ajudarão durante o sprint. Isso também aumenta a probabilidade de que questões importantes sobre a estória surjam cedo.
- Quando pedimos que todos estimem uma estória, freqüentemente descobrimos discrepâncias onde duas pessoas da equipe têm estimativas bastante diferentes para a mesma estória. Esse tipo de coisa é melhor ser descoberto e discutido o quanto antes.

Se você pedir à equipe para que gere uma estimativa, normalmente a pessoa que melhor entende a estória será a primeira a revelar a sua. Infelizmente, isso afetará fortemente as estimativas de todo o resto.

Há uma técnica excelente para evitar isso – ela é chamada *planning poker* (cunhada por Mike Cohn, eu acho).

Cada membro da equipe recebe um baralho de 13 cartas como mostrado acima. Sempre que uma estória deve ser estimada, cada membro escolhe uma carta que representa a sua estimativa de tempo (em pontos por estória) e coloca-a virada para baixo sobre a mesa. Quando todos os membros da equipe tiverem feito sua estimativa, as cartas são reveladas simultaneamente. Dessa forma, cada membro da equipe é forçado a pensar por si próprio ao invés de basear-se na estimativa de outra pessoa.

Se houver uma grande divergência entre duas estimativas, a equipe discute as diferenças e tenta chegar a uma visão comum do trabalho envolvido na estória. Eles podem fazer algum tipo de decomposição de tarefas. Depois disso, a equipe faz novamente a estimativa. Esse processo é repetido até que as estimativas de tempo cheguem a uma convergência, isto é, todas as estimativas sejam *aproximadamente* a mesma para cada estória.

É importante lembrar aos membros da equipe que eles devem estimar a quantidade total de esforço envolvido na estória. Não é somente "a sua" parte do trabalho. O testador não deve somente estimar a quantidade de esforço de teste.

Note que a seqüência de números não é linear. For exemplo, não há nada entre 40 e 100. Por quê?

Assim evita-se a falsa sensação de precisão para grandes estimativas de tempo. Se uma estória é estimada em aproximadamente 20 postos por estória, não é relevante se ela deve ser 20 ou 18 ou 21. Todos sabemos que é uma estória grande e que é difícil estimar. Portanto, 20 é o nosso palpite aproximado.

Quer estimativas mais detalhadas? Então, quebre a estória em estórias menores e estime-as!

E não, você não pode trapacear combinando um 5 e um 2 para fazer um 7. Você deve escolher 5 ou 8. Não há um 7 nas cartas.

Existem algumas cartas especiais:

- 0 = “esta estória já está feita” ou “esta estória é tão pequena que leva somente alguns minutos de trabalho”;
- ? = “Eu não faço idéia alguma”;
- Xícara de café = “Estou cansado demais para pensar. Vamos fazer uma pequena pausa”.

Esclarecendo Estórias

O pior é quando uma equipe orgulhosamente demonstra uma nova funcionalidade durante o *sprint* e o *product owner* franze as sombrancelhas e diz: “Bem, está bonito, mas isso *não é o que eu pedi!*”

Como você assegura que a compreensão que o *product owner* tem, casa com a compreensão que a equipe tem sobre a estória? Ou que cada membro da equipe tem a mesma compreensão de cada estória? Bem, você não tem como. Mas aqui vão algumas simples técnicas para identificar as piores confusões.
A técnica mais simples é ter certeza que todos os campos foram preenchidos em cada estória (ou mais especificamente, para cada estória que for mais importante de ser considerada para essa reunião).

Exemplo 1:
A equipe e o *product owner* estão felizes com o *sprint* e prontos para terminar o encontro. O *scrum master* diz “Esperem um segundo, na estória chamada “adiciona usuário”, não há estimativa. Vamos estimar!” Depois de algumas rodadas de *planning poker* a equipe concorda com 20 pontos por história enquanto o *product owner* levanta e enfaticamente diz: -“O queeeee?!”. Depois de alguns minutos de discussão acalourada, ela se transforma numa não compreensão, por parte da equipe, sobre o escopo “adicionar usuário”, eles acreditaram ser “uma boa interface web para adicionar, remover e proucurar por usuários”, enquanto o *product owner* compreendeu “adicionar usuários” como uma chamada manual usando SQL num banco de dados. Eles estimam novamente e concordam em 5 pontos por estória.

Exemplo 2:
A equipe e o *product owner* estão felizes com o plano do *sprint* e prontos para terminar o encontro. O *scrum master* diz: “Esperem um segundo, como a estória adicionar usuários será demonstrada?” algum murmurinho depois e alguém se levanta e diz: “Bem, primeiro logamos no site e aí então...” e o product owner interrompe: “Logamos no site?! Não, não,

não, essa funcionalidade não deverá ser parte do web site de forma alguma, isso deverá ser apenas um script SQL para administradores técnicos".

A descrição de "como demonstrar" a estória pode e deve ser *muito breve*! Ou você nunca vai encerrar a reunião de planejamento do *sprint* a tempo. Isso basicamente é uma descrição por alto de como executar os mais típicos testes de cenário manualmente. "Faça isso, faça aquilo, e verifique isso".

Eu descobri que essa simples descrição cobre *freqüentes* desentendimentos sobre o escopo da estória. Bom descobri-las cedo, certo?

Quebrando estórias em estórias menores

Estórias não deveriam ser muito pequenas nem muito grandes (em termos de estimativas). Se você possui um grupo de estórias de 0.5 ponto, você provavelmente será uma vítima do microgerenciamento. Da mesma forma, uma estória de 40 pontos terá grande risco de terminar parcialmente completa, o que não produz valor à sua empresa e apenas amplia a administração. Ainda assim, se sua velocidade for 70 e as duas estórias de maior prioridade possuirem o peso de 40 pontos por estória cada, o planejamento torna-se um pouco difícil. Você deverá escolher entre comprometer-se pouco (ex. produzir apenas um item) ou comprometer-se demais (ex.produzir os dois itens).

Descobrimos que é quase sempre possível quebrar uma estória em pedaços menores. Apenas tenha certeza de que as partes (estórias menores) ainda representam entregas com valor de negócio.

Nós normalmente nos empenhamos para estórias estimadas em 2 - 8 homens-dia. Nossa velocidade é geralmente entre 40 e 60 para uma equipe típica, e isso nos dá algo em torno de 10 estórias por *sprint*. Algumas vezes serão tão poucas quanto 5 e outras vezes serão tantas quanto 15. Este é um número gerenciável de cartões para se trabalhar.

Dividindo estórias em tarefas

Espere um momento, qual a diferença entre "tarefas" e "estórias"? Uma pergunta muito apropriada.

A distinção é bem simples. Estórias são trabalhos que podem ser entregues e que o *product owner* tem que se preocupar. Tarefas são atividades que não podem ser entregues, ou atividades que o *product owner* não precisa se preocupar.

Exemplo da divisão de uma estória em outras menores:

Exemplo da divisão de uma estória em tarefas:

Seguem aqui algumas observações importantes:

- Equipes novas no *Scrum* são relutantes em gastar tempo dividindo um conjunto de estórias em tarefas como essas. Algumas sentem que essa é uma abordagem em cascata.
- Para estórias que não foram claramente entendidas, é tão simples fazer essa divisão logo, ou mais tarde.
- Esse tipo de divisão freqüentemente revela trabalho adicional que faz com que a estimativa de tempo aumente, mas por outro lado resulta em um planejamento do sprint mais realista.
- Esse tipo de divisão logo cedo torna as reuniões diárias do scrum notavelmente mais eficientes (veja página 64 “Como fazemos as reuniões diárias”).
- Mesmo que a divisão seja imprecisa e mude quando o trabalho iniciar, as vantagens acima ainda aplicam-se.

Assim, nós tentamos fazer o espaço de tempo do planejamento do scrum grande o bastante para acomodá-la, mas se o tempo esgota-se, deixemos ela de lado (veja “Onde desenhar a linha” abaixo).

Nota – nós praticamos TDD (*test driven development* – desenvolvimento orientado a testes) que na prática significa que a primeira tarefa para quase todas as estórias é “escrever um teste de defeito” e a última tarefa é “refazer” (= melhorar a legibilidade do código e remover duplicidades).

Definindo a hora e lugar da reunião diária

Uma saída frequentemente esquecida da reunião de planejamento de *sprint* é “hora e local para a reunião diária do *scrum* definidos”. Sem isto, seu *sprint* terá um mal início. A primeira reunião diária é essencial para o lançamento onde todos decidem começar a trabalhar.

Eu prefiro reuniões matutinas. Mas, eu devo admitir, nós não tentamos fazer reuniões diárias na tarde ou meio-dia..

Desvantagem de reuniões vespertinas: quando você for trabalhar de manhã, você tem que tentar se lembrar do que falou ontem para as pessoas sobre o que faria hoje.

Desvantagem de reuniões matutinas: quando você for trabalhar de manhã, você tem que se lembrar do que fez ontem, pra que possa reportar isto.

Na minha opinião, a primeira desvantagem é pior, porque o mais importante é o que *você vai fazer*, não o que *você fez*.

Nosso procedimento padrão é selecionar o horário mais cedo sem que ninguém reclame. Normalmente 9:00, 9:30 ou 10:00. O mais importante é que seja uma hora que todos da equipe aceitem completamente, de coração.

Onde traçar a linha

OK, então o tempo está acabando. De todas as coisas que queremos fazer durante a reunião de planejamento do *sprint*, o que nós devemos cortar, se o tempo acabar?

Bem, eu uso a seguinte lista de prioridades:

Prioridade 1: Um objetivo para o *sprint* e uma data para a realização da demonstração. Isto é o mínimo que você precisa para iniciar um *sprint*. A equipe tem um objetivo e uma data de término, e eles podem trabalhar a partir do *product backlog*. É ruim, sim, e você deve seriamente considerar o agendamento de uma nova reunião para planejamento do *sprint* para o dia seguinte, mas se você realmente precisa iniciar o *sprint*, isto deverá ajudar. Para ser honesto, eu nunca iniciei um *sprint* com esta pequena informação.

Prioridade 2: Lista de estórias que a equipe aceitou trabalhar neste *sprint*.

Prioridade 3: Preencha a informação de estimativa para cada estória do *sprint*.

Prioridade 4: A informação "Como demonstrar" deve ser preenchida em cada estória do *sprint*.

Prioridade 5: Cálculos de velocidade e recursos, como uma verificação de realidade para o seu plano de *sprint*. Inclui lista de membros da equipe e o comprometimento deles (de outra forma não se consegue calcular velocidade).

Prioridade 6: Hora e local específico para a reunião diária. Se decide isto de forma muito rápida, mas se você não tiver tempo o *scrum master* pode simplesmente decidir isto depois da reunião e pode avisar todos por e-mail.

Prioridade 7: Estórias quebradas em tarefas. Esta quebra pode ser feita diariamente, assim como as reuniões diárias, mas isto vai gerar uma leve interrupção no fluxo do *sprint*.

Estórias técnicas

Eis aqui um problema complexo: Estórias técnicas. Ou itens não funcionais. Ou como você preferir chamá-los.

Estou referindo-me àquelas coisas que precisam ser feitas mas que não fazem parte das entregas, não estão relacionadas diretamente com nenhuma estória específica e não agregam valor para o *product owner*.

Nós as chamamos de "estórias técnicas".

Por exemplo:

- **Instalar um servidor de build**
  - Por que é necessário: porque economiza uma enorme quantidade de tempo para os desenvolvedores e reduz o risco dos problemas de integração ao fim da iteração.

- **Escrever um resumo do projeto do sistema**
  - Por que é necessário: Porque os desenvolvedores insistem em esquecer-se do projeto geral e, em conseqüência, de escrever um código consistente. Precisam de um documento do tipo "resumão" para manter todos na mesma linha de projeto.

- **Refazer a camada DAO**

    - Por que é necessário: Porque a camada DAO tem sido uma total desordem e tem tomado tempo de todos devido a essa confusão e aos bugs desnecessários. Limpar esse código vai economizar tempo de todos e aumentar a robustez do sistema.

- **Fazer o upgrade do Jira** (bug tracker)
    - Por que é necessário: A versão atual tem muitos *bugs* e é lenta. Fazer o upgrade vai economizar tempo de todos.

Essas estórias fazem parte do senso comum? Ou elas estão conectadas com alguma estória específica? Quem as prioriza? O *product owner* deveria envolver-se com essas coisas?

Fizemos várias experiências com diferentes maneiras de lidar com estórias técnicas. Tentamos tratá-las como estórias de primeira classe, como todas as outras. Não funcionou bem. Quando o *product owner* priorizou o *backlog*, foi como comparar maças com laranjas. De fato, por razões obvias as estórias técnicas obtinham sempre prioridade menor devido ao modo de pensar como: "ok, pessoal, Estou certo de que um servidor de *build* contínuo é importante, mas vamos atacar primeiro alguma coisa que possamos faturar! E então podemos adicionar o brinquedo tecnológico mais tarde, OK?"

Em alguns casos o *product owner* tem razão, mas quase sempre não tem. Concluímos que o *product owner* nem sempre esta qualificado para tomar estas decisões. Assim, o que fazemos é:

1) Tentamos evitar as estórias técnicas. Procure transformar uma estória técnica em uma estória normal, com valores de negócio mensuráveis. Desse jeito o product owner vai ter uma chance maior de tomar as decisões corretas.
2) Se não conseguirmos transformar uma estória técnica em uma estória normal, veja se o trabalho pode ser feito como uma tarefa dentro de outra estória. Por exemplo: "refatorar a camada DAO" pode ser uma tarefa dentro da estória "editar usuário", pois ela envolve a camada DAO.
3) Se os dois itens anteriores falharem, defina-o como uma estória técnica, e mantenha uma lista separada destas estórias. Deixe que o product owner a veja, mas não permita que modifique-a. Utilizamos o parâmetros "fator de foco" e "velocidade estimada" para negociar com o *product owner* e separar algum tempo no *sprint* para implementar as estórias técnicas.

Exemplo (um diálogo muito semelhente à este ocorreu durante uma de nossas reuniões de planejamento do *sprint*.)

- **Equipe**: "Nós temos trabalho técnico interno que precisa ser feito. Gostaríamos de dedicar 10% do nosso tempo para fazê-lo, por exemplo, reduzir o fator de foco de 75% para 65%. Tudo bem?"
- **product owner:** "De jeito nenhum! Nós não temos tempo!"
- **Equipe:** : "Bem, olhe para o último sprint (todas as cabeças viram pra curva de velocidade no quadro branco). Nossa velocidade estimada era 80, a atual é 30, certo?"
- **PO:** "Exatamente! Então nós não temos tempo pra serviço interno! Precisamos de novas características!"
- **Equipe:** "Bem, a *razão* de nossa velocidade ter se tornado tão baixa foi que perdemos muito tempo tentando integrar versões funcionais para testes".
- **PO:** "Sim, e?"
- **Equipe:** "Bem, nossa velocidade vai continuar sendo ruim se não fizermos alguma coisa sobre isso."
- **PO:** "Sim, e?"
- **Equipe:** "Então nós propomos subtraírmos aproximadamente 10% deste sprint para configurar um servidor de produção e para cuidar de outras coisas que irão resolver os problemas de integração. Isto provavelmente irá aumentar nossa velocidade de *sprint* em pelo *menos* 20% para cada *sprint* subsequente. Pra sempre!"
- **PO:** "Verdade? Então por quê nós não fizemos isto no *sprint* passado?"
- **Equipe:** "Err… porque você não quis que fizéssemos."
- **PO:** "Oh, hum, muito bem, parece uma boa idéia fazermos isto agora então!"

É claro, a outra opção é só deixar o *product owner* desinformado ou dar-lhe um fator de foco não negociável. Mas não tem desculpa para não *tentar* chegar à um consenso primeiro.

Se o *product owner* for um companheiro competente e razoável (e nós tivemos sorte aqui) sugiro mantê-lo tão informado quanto possível e deixá-lo decidir sobre as prioridades. Transparência é um dos valores centrais do *Scrum*, certo?

Sistema de acompanhamento de bugs vs. product backlog

Eis aqui uma armadilha. Excel é um ótimo formato para o *product backlog*. Mas você ainda precisa de um sistema de acompanhamento de bugs, e o Excel provavelmente não lhe atenderá. Nós usamos Jira.

Então, como nós levamos os defeitos registrados no Jira para as reuniões de planejamento do *sprint*? Eu quero dizer que não devemos apenas ignora-los e focarmos apenas nas estórias.

Nós tentamos várias estratégias:

1) O *product owner* imprime os bugs prioritários registrados no Jira, leva-os à reunião de planejamento do *sprint* e afixa-os na parede junto com as outras estórias (especificando, desse modo, implicitamente, a prioridade desses itens em comparação com as outras estórias).
2) O *product owner* cria estórias que fazem referência aos itens do Jira. Por exemplo: "consertar os *bugs* mais críticos dos relatórios do *back office*, Jira-124, Jira-126 e Jira-180".
3) O concerto de *bugs* é considerado fora do *sprint*, p.ex., a equipe mantém um fator de foco pequeno o suficiente (50%, por exemplo) para garantir que tenham tempo para consertá-los. Assim, assumem que gastarão um certo período de tempo de cada *sprint* consertando os bugs.
4) Colocar o *product backlog* dentro do Jira (p.ex. descartar o Excel). Trate os bugs como qualquer outra estória.

Nós ainda não concluímos qual dessas estratégias é a melhor para nós. De fato, elas variam de equipe para equipe e de *sprint* para *sprint*. Eu sou propenso a apostar na primeira estratégia. Ela é mais legal e mais simples.

A reunião de planejamento do sprint está finalmente terminada!

Uau, Eu nunca imaginei que este capítulo sobre planejamento do *sprint* fosse ser tão longo! Eu acho que isso reflete a minha opinião de que a reunião de planejamento do sprint é a coisa mais importante que se faz no *Scrum*. Empenhe-se bastante para fazê-la corretamente e todo o resto será muito mais fácil.

A reunião de planejamento do *sprint* obteve sucesso se todos (os membros da equipe e o *product owner*) sairem da reunião com um sorriso no rosto, acordam na outra manhã com um sorriso e todos fazem a primeira *reunião diária* com um sorriso.

Então, é claro, tudo pode dar errado no restante do projeto, mas pelo menos você não poderá culpar o *planejamento do sprint* :o)

5

Como comunicamos sprints

É importante manter a empresa inteira informada sobre o que está acontecendo. Embora as pessoas reclamem, ou mesmo pior, façam falsas suposições sobre o que está acontecendo.

Nós usamos a “página de informações do *sprint*” para isso.

As vezes nós incluímos informações sobre como cada estória será demonstrada também.

O mais cedo possível depois da reunião de planejamento do *sprint,* o *scrum master* cria esta página, põe no ar no *Wiki*, e envia para a empresa inteira.

Nós também temos a página ”*dashboard*” no nosso *wiki*, que junta tudo que está acontecendo nos *sprints*.

Além disso, o *scrum master* imprime a página de informação do *sprint* e coloca no mural do lado de fora da sala da equipe. Então todo mundo que passa pode vê-la e descobrir o que a equipe fazendo. Desde que isso inclua o tempo e o lugar da reunião diária e da apresentação do *sprint*, se saberá onde ir para descobrir mais.

Quando o *sprint* está perto do fim, o *scrum master* lembra todo mundo sobre a apresentação que ocorrerá em breve.

Feito tudo isso, ninguém tem desculpa pra *não* saber o que está acontecendo.

6

Como fazemos os sprint backlogs

Você já fez isso alguma vez? Uau! Bom trabalho.

Bem, agora que já completamos a reunião de planejamento do *sprint* e contamos ao mundo sobre nosso *sprint* novinho em folha, é hora do *Scrum master* criar um *sprint backlog*. Ele precisa ser feito *depois* da reunião de planejamento do *sprint*, mas *antes* da primeiro reunião diária do *Scrum*.

Formato do sprint backlog

Nós já tentamos alguns formatos diferentes para o *sprint backlog*, incluindo Jira, Excel e um quadro de tarefas pregado na parede. No princípio nós usamos principalmente Excel. Existem vários modelos prontos de *sprint backlogs* em Excel, incluindo gráficos de *burndown* automáticos e coisas do gênero. Eu poderia falar bastante como nós refinamos nossos *sprint backlogs* em Excel. Mas não farei isso. Não vou nem incluir um exemplo aqui.

Ao invés disso, irei descrever em detalhes o que nós temos visto ser o formato mais produtivo para o *sprint backlog* – um quadro de tarefas pregado na parede!

Encontre uma parede grande que não é utilizada ou que tem coisas inúteis, como o símbolo da empresa, diagramas velhos e quadros feios. Limpe a parede (só peça permissão se for necssário). Cole uma folha de papel bem grande, bem grande mesmo (pelo menos 2m de altura x 2m de largura, ou 3x2m para uma equipe grande). Então faça assim:

É claro que você poderia usar um quadro branco. Mas seria um desperdício. Se possível, use os quadros-brancos para rascunhos de projeto e as outras paredes para os quadros de tarefas.

NOTA – se você usar post-its para tarefas, não esqueça de colá-los usando durex ou fita crepe mesmo, ou você irá encontrar todos eles misturados no chão no dia seguinte.

Como funciona o quadro de tarefas

Você pode, claro, adicionar todo tipo de colunas adicionais. “Esperando pelo teste de integração” por exemplo. Ou “Cancelado”. Entretanto antes de complicar as coisas, pense profundamente. Esta adição de coluna é *realmente* necessária?

Eu descobri que simplicidade é extremamente válida para este tipo de coisa, então eu apenas adiciono complicações quando o custo de *não* fazer é muito alto.

Exemplo 1 – depois da primeira reunião diária

Depois da primeira reunião diária, o quadro de tarefas vai parecer como este:

Como se pode ver, três tarefas foram selecionadas e colocadas em produção. A equipe vai trabalhar nestas tarefas hoje.

Às vezes, para equipes grandes, uma tarefa fica presa na etapa de produção pois ninguém lembra quem estava trabalhando nela. Se isto acontecer de forma recorrente em uma equipe, normalmente se define uma política como rotular as tarefas em produção com o nome da pessoa que está trabalhando nela.

Exemplo 2 – poucos dias depois

Poucos dias depois, o quadro de tarefas pode se parecer com algo assim:

Como pode ver, completamos a estória “depósito” (i.e. foi integrada ao repositório com o código fonte, testada, refatorada, etc). A ferramenta de migração está parcialmente completa, a parte do login foi iniciada e a parte de administração de usuários não começou.

Tivemos 3 itens não-planejados, como pode se ver na parte inferior direita. Guardá-los é útil para que sejam lembrados quando você for fazer a retrospectiva do *sprint*.

Eis um exemplo de um *sprint backlog* real perto do fim do *sprint*. Ele de fato fica meio bagunçado com o andamento do *sprint*, mas isso está OK pois sua vida é curta. A cada novo *sprint*, nós criamos um fresco, limpo e novo *sprint backlog*.

Como funciona o gráfico de burndown

Vamos dar um *zoom* no gráfico de *burndown*:

Este gráfico mostra que:

- No primeiro dia do *sprint*, 1 de agosto, a equipe estimou que havia aproximadamente 70 pontos por estória de trabalho a serem feitos. Isso foi na verdade a *velocidade estimada* de todo o *sprint*.
- Em 16 de agosto a equipe estimou que havia aproximadamente 15 pontos por estória de trabalho a serem feitos. A linha de tendência tracejada mostra que eles estão aproximadamente dentro do prazo, isto é, neste ritmo eles completarão tudo até o final do *sprint*

Nós pulamos os finais de semana no eixo-x, uma vez que o trabalho raramente é executado aos finais de semana. Nós costumávamos incluir finais de semana mas isso tornaria o gráfico de *burndown* um pouco confuso, já que o mesmo ficaria “reto ” nos finais de semana e isso poderia parecer um sinal de alarme.

Sinais de alarme do quadro de tarefas

Uma olhada rápida no quadro de tarefas daria a qualquer um uma indicação de quão bem o *sprint* está progredindo. O *scrum master* é responsável por garantir que a equipe haja quando surgirem sinais de alarme como:

Ei, e a rastreabilidade?!

A melhor rastreabilidade que eu posso sugerir neste modelo é tirar uma foto digital do quadro de tarefas todo dia. Eu faço isso às vezes, mas eu nunca encontrei um motivo para vasculhar estas fotos.

Se rastreabilidade é muito importante para você, então o quadro de tarefas talvez não seja uma boa solução para você.

Mas eu sugiro que você realmente tente estimar o valor real de ter uma rastreabilidade detalhada do *sprint*. Uma vez que o *sprint* está pronto, o software funcionando e a documentação entregue, alguém realmente se preocupa com quantas estórias foram completadas no dia 5 de um *sprint*? Alguém realmente se preocupa com qual era o tempo estimado para "escrever um teste de falha para Depósito"?

Estimando dias vs horas

Na maioria dos livros e artigos sobre *Scrum*, você vai encontrar que as tarefas são estimadas em horas, e não em dias. Costumávamos fazer isto. Nossa formula geral era: 1 homem-dia = 6 horas-homem reais.

Atualmente deixamos de fazer assim, pelo menos na maior parte de nosso grupo, pelas seguintes razões:

- As estimativas em homem-hora eram muito granulares, e isto provocava uma tendência a estimar muitas tarefas de 1-2 horas e a conseqüente micro gerencia.
- De qualquer forma, como todos estavam pensando em temos de homens-dias, e simplesmente multiplicavam por 6 antes de escrever homem-hora. "Hmmmm, esta tarefa deve gastar um dia. Bom, tenho que escrever horas, então escrevo 6 horas".
- Duas unidades diferentes podem causar confusão. "Esta estimativa é em homem-hora ou homem-dia?

Assim, atualmente usamos o homem-dia para todas as nossas estimativas (embora continuemos chamando de pontos por estória - *story points*). Nosso menor valor é 0,5, isto é, qualquer tarefa que é menor que 0,5 será eliminada, combinada com outras tarefas ou receberá o valor de 0,5. (Não tem problema superestimar um pouco). Simples e elegante.

7

Como arrumamos a sala da equipe

O canto do design

Eu notei que muitas das mais interessantes e valiosas discussões sobre *design* acontecem espontaneamente na frente do quadro de tarefas.

Por esta razão, nós tentamos arrumar esta área explicitamente como um "canto do *design*".

Isto é realmente bem útil. Não há forma melhor de ter uma visão geral do sistema do que ficar no canto do *design* e dar uma olhadela em ambas as paredes, e então dar uma olhadela no computador e tentar o último *build* do sistema (se você for sortudo o bastante para ter *build* contínuo, veja pg 81 "Como combinamos Scrum e XP").

A "parede do *design*" é só um grande quadro-branco contendo os rabiscos de *design* e esboços (*printouts*) da documentação de *design* mais importante (gráficos de seqüências, protótipos de GUI, modelos de domínio, etc).

Acima: uma reunião diária acontecendo no canto mencionado.

Hmmmm..... aquele *burndown* parece suspeitosamente legal e direto, não. Mas a equipe insiste que é real :o)

Sente a equipe junta!

Com relação aos assentos e a organização das mesas de trabalho há uma coisa que pode não ter sido reforçada o suficiente:

Sente a equipe junta!

Para esclarecer um pouquinho mais, o que eu estou dizendo é

Sente a equipe junta!

As pessoas são relutantes para mudar. Pelo menos nos lugares onde eu trabalhei. Eles não querem ter que juntar todas as suas coisas, desconectar o computador, mover todas as suas tralhas para uma nova mesa, e conectar tudo de novo. Quanto menor a distância, maior a relutância, "Vamos lá chefe, para que mudar somente 5 metros?".

Mas quando estamos construindo equipes de *Scrum* eficientes não há outras alternativas. Somente mantenha a equipe junta. Mesmo que você tenha que pessoalmente ameaçar cada um, carregar todo o equipamento deles, e limpar suas velhas manchas de café. Se não há espaço para a equipe, crie um espaço. Qualquer lugar. Mesmo se você tiver que colocar a equipe no porão. Mova as mesas ao redor, suborne o gerente responsável, faça o que for necessário. Somente mantenha a equipe junta.

Uma vez que você tenha a equipe toda junta, o retorno será imediato. Depois de somente um *sprint* a equipe irá concordar que foi uma boa idéia sentar junto (falando por experiência pessoal, mas não há nada dizendo que a sua equipe não será tão teimosa para não admitir isto).

Agora o que "junto" significa? Como devem ser distribuídas as mesas de trabalho? Bem, eu não tenho nenhuma opinião firme sobre a melhor distribuição das mesas. E mesmo que eu tivesse, eu acredito que a maioria das equipes não tenha o luxo de decidir exatamente como posicionar suas mesas de trabalho. Geralmente existem restrições físicas – a equipe vizinha, a porta do banheiro, uma grande máquina caça-níqueis no meio da sala, o que for.

"Junto" significa:

- **Audibilidade:** Qualquer um da equipe pode conversar com o outro sem gritar ou sair da mesa.

- **Visibilidade:** Todos da equipe podem ver todos os demais. Cada um da equipe consegue ver o quadro de tarefas. Não necessariamente perto o suficiente para lê-lo, mas pelo menos para vê-lo.
- **Isolamento:** Se toda equipe de repente levantar e se envolver em uma espontânea e animada discussão sobre implementação, não haverá ninguém de fora da equipe perto o suficiente para ser perturbado. E vice-versa.

"Isolamento" não significa que a equipe tenha que ficar completamente isolada. Em um ambiente de trabalho aberto e repartido em cubículos pode ser suficiente que a equipe tenha seu próprio cubículo com paredes altas o bastante para filtrar o máximo do barulho das outras equipes.

E se você tiver uma equipe distribuída? Bem, então você está sem sorte. Use o máximo de resursos técnicos que você puder para minimizar o dano – vídeo conferência, *webcams*, ferramentas de compartilhamento da área de trabalho, etc.

Mantenha o product owner por perto

O *product owner* deve estar suficientemente perto para permitir que a equipe possa ir onde esta e perguntar qualquer coisa, e também para que ele possa ir ao *quadro de tarefas*. Mas ele não deve estar junto a equipe. Por que? Porque ele provavelmente não será capaz de se controlar e não se meter em detalhes e a equipe não "fluirá" adequadamente (isto é, alcançar um estado focado, hiperprodutivo e auto-gerenciado)

Para ser honesto, isto é uma especulação. Nunca vi um caso em que o *product owner* esteve junto da equipe, de modo que não tenho nenhum motivo para dizer que não é uma boa idéia. É só uma intuição pessoal e o que comentam outros *scrum masters*.

Mantenha os gerentes e coaches por perto

Isto é um pouco difícil para escrever, pois fui tanto gerente quanto *coach*...

Minha tarefa era trabalhar o mais proximo possivel da equipe. Formei os grupos, me movia entre eles, programava em pares com outras pessoas, ajudava outros *scrum masters*, organizava reuniões de planejamento de *sprint*, etc. Fazendo um retrospecto, a maioria das pessoas considerou isso

como uma Coisa Boa, já que tinha alguma experiência com o desenvolvimento ágil de software.

Mas, então, eu era também (entra a musica tema do Darth Vader) o chefe de desenvolvimento, um perfil funcional gerencial. O que significa que ao entrar em uma equipe ela se tornava automaticamente menos auto-gerenciavel. “caramba, o chefe está aqui, ele provavelmente tem um monte de opiniões sobre o que deveríamos estar fazendo e sobre quem deveria estar fazendo o que. Vamos deixar ele falar!”

Meu ponto de vista é: se você é um *coach Scrum* (e talvez também um gerente), envolva-se o máximo possível. Porem só por uma período limitado de tempo, então saia e deixe o grupo “fluir” e se auto-gerenciar. Verifique a equipe de vez em quando (não muito freqüentemente) participando das apresentações de sprint e observando o quadro de tarefas e escutando os *scrum* matinais. Se voce encontrar uma área onde pode haver melhorias, reúna-se com o *scrum master* e ajude-o. *NUNCA* em frente a equipe. Outra boa idéia é participar das retrospectivas do *sprint* (veja na pagina 77 “Como fazemos as retrospectivas do *sprint*), desde que sua equipe confie suficientemente em você para que sua presença não os iniba.

Para as equipes *Scrum* que funcionam bem, garanta que tenham tudo o que precisam e tire do caminho tudo aquilo que pode atrapalhar (exceto durante as apresentações de *sprint*).

8

Como fazemos as reuniões diárias

Nossas reuniões diárias (daily scrums) são seguidas bem a risca. Elas começam exatamente na hora, todo dia no mesmo lugar. No começo nós íamos para um sala separada para fazer o planejamento do sprint (esses eram os dias que usávamos o sprint backlog eletrônico). No entanto, agora nós fazemos nossas reuniões diárias na sala da equipe na frente do quatro de tarefas. Nada supera isso.

Nós normalmente fazemos as reuniões em pé, uma vez que isso reduz o risco de ultrapassar 15 minutos.

Como atualizamos o quadro de tarefas

Normalmente atualizamos o quadro de tarefas durante a reunião diária. Conforme cada pessoa descreve o que fez ontem e o que fará hoje, vai colocando post-its no quadro de tarefas. Enquanto essa pessoa descreve um item não planejado, também cria um post-it para este item. Quando atualiza uma estimativa de prazo, atualiza a mesma no respectivo post-it escrevendo a nova e riscando a estimativa antiga. Às vezes o scrum master faz o trabalho com os post-its enquanto as outras pessoas conversam.

Algumas equipes têm a política de que cada pessoa deve atualizar o quadro de tarefas *antes* de cada reunião. Isso também funciona bem. Apenas escolha uma política e se atenha a ela.
Independente do formato em que seu *sprint backlog* está, tente fazer com que toda a equipe se envolva na tarefa de manter o *sprint backlog* atualizado. Nós tentamos realizar *sprints* onde o *scrum master* era o único mantenedor do *sprint backlog* e todos os dias tinha que sair e perguntar as pessoas sobre suas estimativas de prazo. As desvantagens disso são:

- O *Scrum master* gasta tempo demais administrando essas coisas, ao invés de apoiando a equipe e removendo impedimentos.
- Os membros da equipe perdem a noção do status do *sprint*, uma vez que o *sprint backlog* não é algo com o que eles precisem se preocupar. Essa falta de *feedback* reduz a agilidade e o foco da equipe.

Se o *sprint backlog* for bem definido, cada membro da equipe poderá atualizá-lo facilmente.

Imediatamente após a reunião diária do *Scrum*, alguém totaliza todas as estimativas de prazo (óbviamente, excluindo aquelas que estiverem na coluna de "prontos") e marca um novo ponto no *sprint burndown*.

Lidando com atrasados

Algumas equipes possuem um pote de moedas e notas. Quando você se atrasa, mesmo que apenas um minuto, você coloca um valor pré-determinado no pote. Sem argumentação. Se você ligar antes da reunião avisando que vai se atrasar, ainda assim tem que pagar.

Você só se livra dessa se tiver uma boa desculpa, como consulta médica, seu próprio casamento ou algo do tipo.

O dinheiro do pote é usado para eventos sociais. Comprar sanduíches quando temos noite de jogo, por exemplo. :o)

Isso funciona bem. Mas só é necessário para equipes onde as pessoas se atrasam freqüentemente. Algumas equipes não precisam desse tipo de método.

Lidando com o "não sei o que fazer hoje"

Não é raro alguem dizer "Ontem eu fiz bla bla bla, mas hoje eu não tenho a mínima ideia do que fazer " (ei, essa ultima parte rimou). E agora?

Vamos dizer que Joe e Lisa são os que não sabem o que fazer hoje.

Se eu sou *scrum master*, eu apenas passo e deixo a próxima pessoa falar, mas anoto a pessoa que não tem nada para fazer. Depois de todos terem falado, eu vou ao quadro de tarefas com toda a equipe, de cima para baixo, e verifico que tudo está em sincronia, que todos sabem o que cada

item significa, etc. Eu convido as pessoas a adicionarem mais post-its. Então eu volto para aquelas pessoas que não sabem o que fazer: "agora que nós passamos pelo quadro de tarefas, vocês tem alguma idéia sobre o que vocês podem fazer hoje"? Tomara que tenham.

Se não, eu avalio se há uma oportunidade de programação em par. Vamos dizer que Niklas está indo implementar a GUI de *back office* do usuário administrador hoje. Nesse caso eu educadamente sugiro que talvez Joe e Lisa possam fazer par com Niklas nisso. Normalmente funciona.

E se aquilo não funcionar, aqui está o próximo truque.

Scrum master: "OK, quem quer mostrar a *release* beta pra gente? (considerando que isso era o objetido do *sprint*)
Equipe: silêncio confuso
Scrum master: "Não estamos prontos?"
Equipe: "hum... não"
Scrum master: "Mas que coisa. Por que não? O que está faltando?"
Equipe: "Bem, não temos o servidor de testes pra rodar, e o *script* de geração de *build* está falhando."
Scrum master: "Aha." (adiciona dois post-its no quadro de tarefas). "Joe e Lisa, como podem nos ajudar hoje?"
Joe: "Hum.... Acho que vou tentar algum servidor de testes em algum lugar".
Lisa: "... e eu vou tentar arrumar o *script* da *build*".

Se tiver sorte, alguém de fato irá mostrar a *release* que pediu. Ótimo! Você atingiu o objetivo do *sprint*. Mas e se estiver no meio do *sprint*? Fácil. Parabenize a equipe pelo trabalho bem feito, pegue uma ou duas estórias da seção "próximos" no seu quadro de tarefas e passe para "a fazer" à esquerda. Então refaça a reunião diária. Notifique o *product owner* que você adicionou novos intens no *sprint*.

Mas o que acontece se a equipe não alcançou o objetivo e Joe e Lisa ainda se recusam a fazer algo útil. Eu normalmente considero uma das seguintes estratégias (nenhuma delas é muito agradável mas é uma última alternativa):

- **Vergonha:** "Bem eu não tenho nenhuma idéia de como vocês podem ajudar a equipe mas acho que talvez vocês possam ir para casa, ler um livro ou alguma outra coisa. Ou apenas se sentar e ver se alguém precisa de ajuda.".
- **Velha-Escola:** Apenas atribua-os uma tarefa.

- **Pressão do par:** Diga “Usem o tempo que for necessário, Joe e Lisa. Nós todos estaremos esperando aqui sem pressa até que vocês possam arrumar algo que nos ajude a alcançar o objetivo.”
- **Servidão:** Diga “Vocês podem ajudar a equipe indiretamente sendo mordomos hoje. Façam café, massageiem as pessoas, limpem o lixo, cozinhem-nos algo para comer, e qualquer outra coisa que nós pedirmos durante o dia.” Vocês se surpreenderão em quão rapidamente Joe e Lisa se apresentarão para as tarefas técnicas :o)

Se uma pessoa freqüentemente te forçar a tomar esse tipo de medidas, você precisa removê-la do grupo e dar um sério treinamento. Se o problema persistir, você deve avaliar o quanto a pessoa é importante para a equipe.

Se *não for*, tente removê-la da equipe.

Se *for*, tente pareá-lo com alguém que possa ser o seu “pastor”. Joe será um ótimo desenvolvedor e arquiteto, apenas ele prefere que outra pessoa lhe fale o que fazer . Perfeito. Dê à Nicklas o dever de ser o pastor permanente de Joe. Ou assuma a responsabilidade você mesmo. Se Joe é importante o suficiente para sua equipe, isso será um esforço válido. Nós tivemos casos como esse e isso funcionou mais ou menos.

9

Como fazemos apresentações de sprint

A apresentação de *sprint* (ou *revisão* de sprint como algumas pessoas chamam), é uma parte importante do *Scrum* que as pessoas tendem a subestimar.

"Oh nós realmente precisamos fazer uma apresentação? Realmente não há muita diversão para mostrar!"
"Nós temos tempo para preparar a &%$# de uma apresentação!"
"Eu não tenho tempo para assistir às apresentações de outras equipes!"

Por que insistimos que todos os sprints terminem com uma apresentação

Uma apresentação de *sprint* bem feita, apesar disso poder parecer dramático, tem um efeito profundo.

- A equipe ganha crédito por suas realizações. *Eles se sentem bem.*
- Outros aprendem o que sua equipe está fazendo.
- A apresentação atrai *feedback* vital dos *stakeholders*.
- Apresentações são (ou deveriam ser) um evento social onde equipes diferentes podem interagir umas com as outras e discutir seu trabalho. Isso tem muito valor.
- Fazer uma apresentação força a equipe a realmente *terminar as coisas* e liberá-las (mesmo que seja apenas em um ambiente de teste). Sem apresentações, nós continuamos recebendo imensas pilhas de coisas 99% prontas. Com apresentações nós podemos ter menos itens prontos, mas estes itens estarão *realmente prontos*, o que é (no nosso caso) muito melhor do que termos uma pilha inteira de coisas que estão apenas parcialmente prontas e que irão poluir o próximo *sprint.*

Se uma equipe é mais ou menos forçada a fazer uma apresentação do *sprint* mesmo não tendo muita coisa funcionando para ser mostrada, a apresentação será constrangedora. A equipe irá gaguejar e tropeçar

durante a apresentação e os aplausos no final serão fingidos. As pessoas se lamentarão pela equipe, alguns podem ficar irritados por terem perdido tempo indo à uma apresentação ruim.

Isso machuca. Mas o efeito é como o de um remédio amargo. No *próximo sprint*, a equipe realmente tentará concluir as coisas! Eles pensarão "talvez nós possamos demonstrar apenas 2 coisas no próximo *sprint* ao invés de 5, mas poxa, dessa vez elas irão funcionar!". A equipe sabe que de qualquer forma eles terão que fazer uma apresentação, o que aumenta significantemente a chance de que haja algo útil para demonstrar. Eu vi isso acontecer muitas vezes.

Checklist para as apresentações de sprint

- Certifique-se de apresentar claramente o objetivo do *sprint*. Se houver pessoas na apresentação que não sabem absolutamente nada sobre seu produto, utilize alguns minutos para descrevê-lo.
- Não gaste muito tempo preparando a apresentação, especialmente evite enfeitar demais as apresentações. Corte tudo que não interessa e foque-se apenas em demonstrar código realmente funcionando.
- Mantenha um bom ritmo, isto é, foque sua preparação em fazer com que a apresentação seja rápida ao invés de bonita.
- Mantenha a apresentação em um nível orientado ao negócio, deixe de fora detalhes técnicos. Foque em "o que nós fizemos" ao invés de "como nós fizemos".
- Se possível, deixe a audiência testar o produto.
- Não demonstre um monte de correções de pequenos bugs e funcionalidades triviais. Mencione-as mas não as demonstre, já que isso geralmente leva muito tempo e desvia o foco das estórias mais importantes.

Lidando com coisas "não demonstráveis"

Membro da equipe: "Eu não vou demonstrar esse item, porque isso não pode ser demonstrado. A estória é 'Melhorar a escalabilidade para que o sistema possa suportar 10.000 usuários simultaneamente'. Eu não posso preparar 10.000 usuários simultaneamente para uma apresentação, posso?"
Scrum master: "Você terminou a estória?"
Membro da equipe. "Sim, claro".
Scrum master: "Como você sabe?"

Membro da equipe: “Eu configurei o sistema em um ambiente de teste de performance, iniciei 8 servidores de carga e disparei requisições simultâneas contra o sistema”.
Scrum master: “Mas você tem alguma indicação de que o sistema iria suportar 10.000 usuários?”.
Membro da equipe: “Sim. As máquinas de teste estavam com o limite insignificante, elas ainda poderiam suportar 50.000 requisições simultâneas durante meu teste”.
Scrum master: “Como você sabe?”
Membro da equipe (frustrado): “Bem, eu tenho esse relatório! Você mesmo pode ver, o relatório mostra como o teste estava configurado e quantas requisições eu enviei!”
Scrum master: “Oh excelente! Então essa é sua “apresentação”. Apenas mostre o relatório e passe-o para a platéia. Melhor do que nada, né?”.
Membro da equipe: “Oh, isto é suficiente? Mas isso é feio, preciso melhorá-lo.”.
Scrum master: “OK, mas não gaste muito tempo. O relatório não tem que estar bonito, apenas informativo.”

10

Como fazemos retrospectivas de Sprints

Por que insistimos que todas as equipes façam retrospectivas

A coisa mais importante sobre retrospectivas é *assegurar-se de que elas aconteçam.*

Por alguma razão, equipes nem sempre parecem propensas a fazer retrospectivas. Sem um estímulo, muitas de nossas equipes freqüentemente não fariam a retrospectiva e seguiriam adiante para o próximo *sprint*. Pode ser algo cultural na Suécia, não tenho certeza.

Mesmo assim, todos concordam que retrospectivas são extremamente úteis. Na verdade, eu diria que a retrospectiva é o segundo evento mais importante no *Scrum* (o primeiro seria a reunião de planejamento do *sprint*), pois essa é a sua *melhor chance de melhorar*!

É claro, você não precisa que de uma reunião de retrospectiva, saiam boas idéias. Você pode fazer isso na sua banheira em casa! Mas a equipe aceitará a idéia? Talvez, mas a probabilidade de obter a aceitação da equipe é muito maior se a idéia vem "da equipe", isto é, ocorre durante a retrospectiva, quando todos têm permissão para contribuir e discutir as idéias.

Sem retrospectivas, você descobrirá que a equipe continua a cometer os mesmos erros repetidamente.

Como organizamos retrospectivas

O formato geral varia um pouco, mas geralmente nós fazemos assim:

- Nós alocamos 1 a 3 horas dependendo de quanto a discussão já estiver adiantada.
- Participantes: o *product owner*, a equipe toda, e eu mesmo (*Scrum master*).
- Nós vamos para uma sala fechada, ou um confortável sofá de canto, um terraço, ou algum outro lugar como estes. Onde nós pudermos não ter interrupções.
- Nós geralmente não fazemos retrospectivas no espaço de trabalho da equipe, pois as pessoas tendem a perder a atenção.
- Alguém é designado como secretário.
- O *Scrum master* mostra o *sprint backlog* e, com a ajuda da equipe, resume o *sprint*. Eventos e decisões importantes, etc.
- Nós fazemos "as rodadas". Cada pessoa tem a chance de dizer, sem ser interrompido, o que eles acharam que foi bom, o que eles acharam que poderia ter sido melhor, e o que eles gostariam de fazer de forma diferente no próximo *sprint*.
- Nós comparamos a velocidade estimada com a velocidade realizada. Se existir uma grande diferença nós tentamos analisar o motivo.
- Quando o tempo já está quase acabado, o *Scrum master* tenta resumir as sugestões concretas sobre o que nós poderemos fazer melhor no próximo *sprint*.

Nossas retrospectivas não são geralmente muito estruturadas. Mas o tema subjacente é sempre o mesmo: "o que nós podemos fazer melhor no próximo *sprint*".

Aqui está um quadro de exemplo de uma retrospectiva recente:

Três colunas:

- **Bom**: se pudéssemos refazer o mesmo *sprint* novamente, faríamos as coisas do mesmo modo;
- **Poderia ter sido melhor**: se pudéssemos refazer o mesmo *sprint* novamente, faríamos as coisas de uma maneira diferente;
- **Melhorias**: idéias concretas sobre como podemos melhorar no futuro;

Assim sendo, as colunas 1 e 2 olham pro passado enquanto que a coluna 3 olha pro futuro.

Depois que a equipe fez o *brainstorming* com todos esses post-its, eles usaram uma votação para determinar quais melhorias devem dar atenção

durante o próximo *sprint*. A cada membro da equipe foi dado três ímãs e foram convidados a votar em quaisquer das melhorias que gostariam que a equipe priorizasse para execução durante o próximo *sprint*. Cada membro da equipe poderia distribuir os ímãs como quisesse, até mesmo colocar todos no mesmo item.

Baseado nisso, selecionarão cinco melhorias de processo para dar atenção e verificarão isso durante a próxima retrospectiva do *sprint*.

É importante não ser ambicioso demais nesse momento. Mantenha o foco somente em algumas melhorias por *sprint*.

Divulgando lições aprendidas entre equipes

A informação que surge durante uma retrospectiva do *sprint* é normalmente muito valiosa. Esta equipe está com dificuldades de foco por que o gerente de vendas está sequestrando programadores para participarem como "especialistas técnicos" em reuniões de vendas? Esta é uma informação importante. Quem sabe outras equipes estejam tendo os mesmos problemas? Deveríamos educar mais o gerente de produtos a respeito de nossos produtos, para que ele possa ser o apoio às vendas?

Uma retrospectiva de *sprint* não diz respeito a apenas como uma única equipe pode fazer um bom trabalho para o próximo *sprint.* Possui implicações muito maiores que isso.

Nossa estratégia para tratar isso é bastante simples. Uma pessoa (neste caso, eu) participa de todas as retrospectivas de *sprint*, e age como uma ponte de conhecimento. Bastante informal.

Uma alternativa seria fazer com que cada equipe publicasse um relatório da reunião de retrospectiva. Nós tentamos, mas descobrimos que poucas pessoas liam estes relatórios, e menos ainda são os que agem através deles. Então fizemos realmente do jeito mais fácil.

Regras importantes para uma pessoa "ponte de conhecimento":

- Deve ser um bom ouvinte
- Se a retrospectiva estiver em muito silêncio, ele deve estar preparado para fazer perguntas simples mas estratégicas para estimular a discussão do grupo. Por exemplo: "se você pudesse retornar no tempo e refazer o mesmo sprint por 1 dia, o que seria feito diferente?"

- Deve estar disposto a investir tempo visitando todas as retrospectivas de todas as equipes.
- Deve estar em algum tipo de posição de autoridade, para que possa agir em sugestões melhorias que estejam fora do controle da equipe.

Isto funcionou perfeitamente bem, mas podem existir outras abordagens que funcionariam muito melhor. Neste caso, por favor me elucide.

Mudar ou não mudar

Digamos que a equipe conclua que "nos comunicamos muito pouco entre nós, então pisamos nos calos dos outros e bagunçamos os *designs* uns dos outros."

O que você deve fazer sobre isso? Introduzir reuniões diárias de *design*? Introduzir novas ferramentas para facilitarem a comunicação? Adicionar mais páginas wiki? Bem, talvez. Pensando novamente, talvez não.

Descobrimos que, em muitos casos, só identificar o problema com clareza suficiente é o bastante para resolvê-lo automaticamente no próximo *sprint*. Especialmente se você cola a retrospectiva do *sprint* na parede (sempre nos esquecemos disso, que vergonha pra gente!). Cada mudança que você introduz, adiciona algum tipo de custo. Então, antes de introduzí-las, considere não fazer nada e esperar o problema desaparecer (ou se tornar menor) automaticamente.

O exemplo acima ("nos comunicamos pouco entre nós…"), é um típico exemplo de algo que é melhor resolvido sem se fazer absolutamente nada.

Se você intorduzir uma nova mudança a cada vez que alguém reclamar de alguma coisa, pessoas podem se tornar relutantes a revelar problemas menores, o que seria terrível.

Exemplos de coisas que podem aparecer durante as retrospectivas

Aqui vão alguns exemplos de situações comuns que podem aparecer durante o planejamento do *sprint*, e algumas ações.

"Nós deveríamos ter gasto mais tempo quebrando as estórias em sub-itens e tarefas"

Isto é bem comum. Todos os dias na reunião diária, membros da equipe se pegam dizendo: "Eu realmente não sei o que fazer hoje". Então após cada reunião diária você gasta mais tempo tentando encontrar as tarefas concretas. Geralmente é mais eficaz adiar isto para um momento mais oportuno.

Ações típicas: Nenhuma. A equipe provavelmente encontrará uma solução por ela mesma durante o próximo planejamento do *sprint*. Se isto acontece frequentemente, aumente o tempo do planejamento do *sprint*.

"Interferências externas demais"

Ações típicas:

• peça à equipe para reduzir o fator de foco no próximo *sprint*, assim terão um plano mais realista

• peça à equipe para recordar melhor as interferências no próxmo sprint. Quem interrompeu, quanto tempo tomou. Será mais fácil resolver o problema depois.

• peça à equipe para concentrar as interferências no *scrum master* ou *product owner*

• peça à equipe para designar uma pessoa para ser o "goleiro". Todas as interrupções serão repassadas à ele, assim o resto da equipe pode se concentrar. Pode ser o *scrum master* ou ser revezado entre o resto.

"Nós nos comprometemos com coisas demais e só metade está pronta"

Ações típicas: nada. A equipe provavelmente não vai se comprometer com coisas demais no próximo *sprint.* Ou no mínimo não errar por tanto.

"Nosso ambiente é barulhento e bagunçado demais"

Ações típicas:

- tente criar um ambiente melhor, ou tire a equipe do local. Alugue um quarto de hotel, Qualquer coisa. Veja pg 59 "Como arrumamos a sala da equipe").
- Se não for possível, diga à equipe para diminuir o fator de foco no próximo sprint, e deixar claro que isso se deve ao ambiente barulhento e bagunçado. Espera-se que que isto faça o *product owner* tomar alguma atitude sobre isto.

Felizmente nunca tive que mudar a equipe de lugar. Mas faria isto se fosse necessário :o)

11

Intervalo entre sprints

Na vida real, nem sempre você pode estar em uma corrida (*sprint*). Você precisa descansar entre as corridas (*sprint*). Se você está sempre correndo, na verdade você está caminhando.

O mesmo acontece no *Scrum* e no desenvolvimento de software em geral. *Sprints* são muito intensos. Como desenvolvedor, você nunca irá querer estar descansando, todos os dias você tem que ir naquela reunião infeliz e dizer diante de todo mundo o que você fez ontem. Poucos irão dizer: “Eu gastei a maior parte do meu dia navegando por blogs e servindo cappuccino”.

Adicionalmente ao descanso propriamente dito, existe uma outra boa razão para ter um intervalo entre os *sprints*. Após a apresentação do *sprint* e a retrospectiva, a equipe e o *product owner* estarão ambos cheios de informações e idéias para digerir. Se eles imediatamente correrem e começarem a planejar o próximo *sprint*, existirá grande possibilidade de ninguém ter tempo de digerir qualquer informação ou lições aprendidas, o *product owner* não terá tido tempo para ajustar suas prioridades após a demonstração do *sprint*, etc.

Ruim:

Nós tentaremos introduzir algum tipo de intervalo após começar um novo *sprint* (mais especificamente, no período *após* a retrospectiva do *sprint* e *antes* da próxima reunião de planejamento do *sprint*). Nem sempre conseguimos.

Por último, nós tentamos ter a certeza que a retrospectiva do *sprint* e suas reuniões de planejamento subseqüentes não ocorram no mesmo dia. Todos devem ter uma boa noite-sem-*sprint* de sono antes de começar um novo *sprint*.

Melhor:

Melhor ainda:

Uma forma de fazer isso é com os "*lab days*" (ou qualquer nome que você quiser dar). São dias que os desenvolvedores podem fazer o que eles quisererm (OK, fui inspirado pelo Google). Por exemplo, eles podem ler sobre as últimas versões de algumas ferramentas ou APIs, estudar para certificação, discutir temas técnicos com os colegas, participar de um projeto pessoal, etc.

Nosso objetivo é ter um "*lab day*" entre cada *sprint*. Dessa forma você ganha um descanso natural entre os *sprints*, e ao mesmo tempo, você permite à sua equipe de desenvolvedores manter os conhecimentos em dia. Além disso, é um benefício bem atraente para contratação.

Melhor de todos?

Atualmente nós temos um *lab day* por mês. Especificamente, a primeira sexta-feira de cada mês. Por que não entre os *sprints*? Bem, porque eu senti que era importante para a empresa toda ter o *lab day* ao mesmo tempo. De outra forma, as pessoas tendem a não levar isso muito a sério. E já que nós não temos os *sprints* de todos os produtos alinhados, eu tive que selecionar um *lab day* independente dos *sprints*.

Um dia nós poderíamos tentar sincronizar os *sprints* de todos os produtos (p.ex., a mesma data de início e fim dos *sprints* para todos os produtos e equipes). Nesse caso, nós colocaríamos o *lab day* entre cada *sprint*.

12

Como fazemos o planejamento de release e contratos com preço fixo

Algumas vezes nós precisamos planejar antecipadamente mais de um *sprint* por vez. Tipicamente em conjunto com um contrato de preço fixo onde nós *temos* que planejar antecipadamente, ou então arriscar assinar algo que não poderemos entregar a tempo.

Tipicamente, o planejamento de *release* é para nós uma tentativa de responder à questão "*quando*, no *pior* caso, nós seremos capazes de entregar a versão 1.0 deste novo sistema".

Se você *realmente* quer aprender sobre planejamento de release eu sugiro que você pule este capítulo e compre o livro "*Agile Estimating and Planning*", de Mike Cohn. Eu realmente gostaria de ter lido esse livro mais cedo (eu o li *após* termos percebido as coisas por conta própria...). Minha versão de planejamento de release é um pouco simplista mas deve servir como um bom ponto de partida.

Defina seus critérios de aceitação

Além do *product backlog*, o *product owner* define uma lista de *critérios de aceitação*, que é uma simples classificação do que os níveis de importância no *product backlog* realmente significam em termos do contrato.

Aqui está um exemplo de regras para critérios de aceitação:

- Todos os itens com importância >= 100 *devem* ser incluídos na versão 1.0, caso contrário seremos mortalmente penalizados.
- Todos os itens com importância entre 50 e 99 *deveriam* ser incluídos na versão 1.0, mas nós *devemos* ser capazes de concluí-los em um release subsequente feito rapidamente.
- Itens com importância entre 25 e 49 são necessários, mas podem ser feitos em release 1.1 subsequente.

- Itens com importância < 25 são especulativos e podem até mesmo nunca vir a ser necessários.

E aqui está um exemplo de um *product backlog*, codificado em cores com base nas regras acima.

Importância	Nome
130	banana
120	maçã
115	laranja
110	goiaba
100	pêra
95	passa
80	amendoim
70	donut
60	cebola
40	framboesa
35	mamão
10	mirtilo
10	pêssego

Vermelho = *deve* ser incluído na versão 1.0 (banana - pêra)
Amarelo = *deveria* ser incluído na versão 1.0 (passa - cebola)
Verde = pode ser feito mais tarde (framboesa - pêssego)

Logo se nós entregarmos tudo entre banana e cebola dentro do prazo, estamos a salvo. Se o tempo encurtar nós podemos deixar de entregar passa, amendoim, donut ou cebola. Tudo abaixo disso é bônus.

Faça estimativas de tempo para os itens mais importantes

Afim de fazer o planejamento de release o *product owner* precisa de estimativas, ao menos para todas as histórias que estão inclusas no contrato. Assim como no planejamento do *sprint*, este é um esforço cooperativo entre o *product owner* e a equipe – a equipe estima, o *product owner* descreve os itens e responde questões.

Uma estimativa de tempo é *valiosa* ao se mostrar perto do correto, menos valiosa ao se distanciar, digamos, por um fator de 30% e completamente sem valor se não possuir qualquer conexão com a realidade.

Aqui está um exemplo do valor de uma estimativa de tempo em função de quem fez os cálculos e de quanto tempo eles gastaram fazendo isso.

Tudo isso foi apenas uma maneira complicada de dizer:

- Deixa a *equipe* fazer as estimativas.
- Não faça com que eles gastem tempo demais.
- Certifique-se de que eles tenham entendido que as estimativas de tempo são estimativas cruas, não compromissos.

Normalmente o *product owner* reune toda a equipe em uma sala, fornece algumas distrações e lhes diz que o objetivo da reunião é fazer uma estimativa de tempo para as 20 estórias mais importantes no *product backlog* (ou algo do tipo). Ele passa por cada estória uma vez e então deixa a equipe trabalhar. O *product owner* permanece na sala para responder perguntas e esclarecer o escopo de cada item conforme necessário. Assim como no planejamento de *sprint*, o campo "como apresentar" é uma maneira bastante útil de diminuir o risco de equívocos.

Esta reunião deve ocorrer dentro de um intervalo de tempo fixo, caso contrário as equipes tendem a gastar tempo demais estimando poucas estórias.
Se o *product owner* desejar que mais tempo seja gasto nessa tarefa, ele simplesmente agenda outra reunião para mais tarde. A equipe deve se certificar de que o impacto destas reuniões nos seus *sprints* seja claramente visível para o *product owner*, para que ele possa compreender que o trabalho de estimativas de tempo possui um custo.

Imp	Nome	Estimativa
130	banana	12
120	maçã	9
115	laranja	20
110	goiaba	8
100	pêra	20
95	passa	12
80	amendoim	10
70	donut	8
60	cebola	10
40	framboesa	14
35	mamão	4
10	mirtilo	
10	pêssego	

Aqui está um exemplo de como estimativas de tempo devem terminar (em pontos por estória):

Estime velocidade

OK, agora nós temos uma estimativa de tempo, bem a grosso modo, para as estórias mais importantes.
Próximo passo é estimar nossa velocidade média por *sprint*.

Isso significa que nós precisamos decidir nosso fator de foco. Veja a pg 24 "Como a equipe decide quais estórias incluir no *sprint*".

O fator de foco é basicamente “quanto do tempo da equipe é gasto focando na execução da estória em questão”. Nunca é de 100%, visto que equipes perdem tempo fazendo itens não planejados, trocando o contexto que estão fazendo, ajudando outras equipes, verificando seus e-mails, consertando seus computadores com defeito, discutindo políticas na cozinha, etc.

Vamos dizer que nós determinemos o fator de foco da equipe em 50% (um pouco baixo, nós normalmente paramos em torno de 70%). E vamos dizer que o tamanho do nosso *sprint* será de 3 semanas (15 dias) e nossa equipe é composta por 6 membros.

Cada *sprint* tem 90 dias-homem de duração, mas podemos esperar uma produção completa de 45 dias-homem no valor das estórias (devido o fator de foco ser de 50%).

Então nossa velocidade estimada é de 45 pontos.

Se cada estória tinha uma estimativa de 5 dias (que eles não) então essa equipe poderia desenvolver aproximadamente 9 histórias por *sprint*.

Junte tudo num plano de release

Agora que temos estimativas de tempo e uma velocidade (45) podemos facilmente fatiar o *product backlog* em *sprints*:

Imp	Name	Estimativa
		Sprint 1
130	banana	12
120	maçã	9
115	laranja	20
		Sprint 2
110	goiaba	8
100	pêra	20
95	passa	12
		Sprint 3
80	amendoim	10
70	donut	8
60	cebola	10
40	framboesa	14
		Sprint 4
35	mamão	4
10	mirtilo	
10	pêssego	

Cada *sprint* inclue tantas estórias quanto possível sem exceder a velocidade estimada de 45.

Agora podemos ver que precisaremos de provavelmente 3 *sprints* para terminar todos os “deve ter” e os “deveria ter”.

3 *sprints* = 9 semanas = 2 meses. É essa a *deadline* que prometemos ao cliente? Depende completamente da natureza do contrato; quão fixo é o escopo, etc. Muitas vezes adicionamos um *buffer* significante para nos proteger de más estimativas, problemas inesperados, funcionalidades inesperadas, etc. Neste caso devemos acordar em uma entrega para daqui a 3 meses, nos dando um mês de “reserva”.

O lado bom é que podemos demonstrar algo usável para o cliente a cada 3 semanas e convidá-lo a mudar os requisitos conforme vamos avançando (claro que dependendo de como é o contrato).

Adaptação do plano de release

A realidade não se adaptará ao plano, então o plano deve adaptar-se à realidade.

Após cada *sprint*, observamos a velocidade para aquele *sprint*. Se a velocidade atual foi muito diferente da velocidade estimada, revisamos a velocidade estimada para os *sprints* futuros e atualizamos o plano de *release*. Se isso nos deixar em apuros, o *product owner* pode então negociar com o cliente ou verificar como reduzir o escopo sem descumprir o contrato. Também pode ser que ele e a equipe descubram alguma maneira de aumentar a velocidade ou o fator de foco, através da remoção de alguns impedimentos graves que foram identificados durante o *sprint*.

O *product owner* pode ligar para o cliente e dizer "Olá, nós estamos um pouco atrasados, mas acredito que podemos entregar no prazo simplesmente removendo a funcionalidade "Pacman embutido" que leva um grande tempo de desenvolvimento. Se você quiser, poderemos adicioná-la em um *release* posterior, 3 semanas após o primeiro *release*."

Talvez não seja uma boa notícia para o cliente, mas pelo menos estaremos sendo honestos e daremos ao cliente a opção de adiantar a entrega – devemos entregar as coisas mais importantes no prazo ou entregar tudo atrasado. Geralmente, esta não é uma decisão difícil. :o)

13

Como combinamos Scrum e XP

Dizer que Scrum e XP (*eXtreme Programming*) podem ser combinados de forma vantajosa não seria uma afirmação polêmica. A maioria do material que vejo na internet apoia essa hipótese, então não vou me demorar justificando o porquê.

Bem, terei que dizer uma coisa. O *Scrum* é focado nas práticas de gerenciamento e organização, enquanto o XP dá mais atenção às tarefas de programação mesmo. Aí está o porquê de elas trabalharem bem juntas – elas abrangem áreas diferentes e uma complementa a outra.

Eu deixo aqui minha opinião de que o *Scrum* e o XP podem ser combinados de forma a dar resultados muito bons!

Eu vou destacar algumas das melhores práticas do XP e como elas se aplicam ao nosso dia-a-dia. Nem todas nossas equipes decidiram adotar todas elas, mas no total nós experimentamos a maioria das combinações de XP/*Scrum*. Algumas dessas práticas são diretamente tratadas pelo *Scrum* e podem ser vistas como sobrepostas, por exemplo, "Toda a Equipe", "Sentar Juntos", "Estórias" e "*Planning game*". Em todos esses casos, nós simplesmente usamos o *Scrum* ao invés do XP.

Programação em par

Nós começamos com esta prática em uma de nossas equipes. Atualmente tem funcionado bem. A maioria de nossas outras equipes não usam programação em par freqüentemente, mas como tentamos em uma equipe por alguns *sprints*, eu me sinto encorajado a incentivar mais equipes a colocarem em prática.

Algumas conclusões sobre programação em par:

- Programação em par aumenta a qualidade do código

- Programação em par aumenta o foco da equipe (por exemplo: quando o par lhe diz: “Ei, estas coisas realmente são necessárias para este *sprint*?)
- Surpreendentemente muitos desenvolvedores que são fortemente contra programação em par nunca tentaram antes e rapidamente aprendem a gostar uma vez que experimentam.
- Programação em par é cansativa e não deve ser feito todos os dias
- Trocar os pares freqüentemente é bom.
- Programação em par nivela o conhecimento da equipe. Surpreendentemente de forma rápida também.
- Algumas pessoas simplesmente não se sentem confortáveis com programação em par. Não jogue fora um excelente programador simplesmente porque ele não se sente confortável com programação em par
- Revisão de código é uma alternativa válida para a programação em par.
- O “navegador” (o cara que não está digitando) deve ter um computador para si próprio. Não para ser usado no desenvolvimento, mas para pequenas atividades quando necessário, navegar na documentação enquanto o “piloto” (o cara nos teclados) o espera, etc.
- Não force as pessoas a praticarem a programação em par. Encoraje as pessoas e forneça as ferramentas corretas, mas deixe-as experimentar por si mesmas.

Desenvolvimento Orientado a Testes (TDD)

Amém! Isso é para mim mais importante que *Scrum* e XP. Você pode tirar minha casa, minha TV e meu cachorro, mas não tente me fazer parar de usar TDD! Se você não gosta de TDD então não me deixe no recinto, porque eu vou tentar fazer isso passar de um jeito ou de outro :o)

Aqui vai um sumário de 10 segundos de TDD:

> *Desenvolvimento orientado a testes significa que você escreve um teste automatizado, então escreve apenas código suficiente para fazê-lo passar, então você refatora o código primeiramente para melhorar a legibilidade e remover duplicação. Enxague e repita.*

Algumas reflexões sobre o desenvolvimento orientado a testes.

- TDD é difícil. Leva um tempo para o programador pegar o jeito. Na verdade, em muitos casos não importa quanto você ensina, treina e

demonstra - em muitos casos a única maneira de um programador "pegar o jeito" é fazendo ele programar em par com alguém que é bom em TDD. Contudo, uma vez que pega o jeito, normalmente é severamente infectado e nunca mais irá querer trabalhar de outra forma.

- TDD tem um efeito profundamente positivo no *design* do sistema.
- Leva tempo para deixar o TDD efetivamente pronto e funcionando em um novo produto, especialmente em testes de integração de caixa-preta, mas o retorno de investimento é rápido.
- Tenha certeza que você investe o tempo necessário para tornar mais fácil escrever os testes. Isso significa obter as ferramentas certas, educar as pessoas, prover o utilitário de classes ou base de classes certo, etc.

Nós usamos as seguintes ferramentes para desenvolvimento orientado a testes.

- JUnit / httpUnit / jWebUnit. Estamos considerando TestNG e Selenium.
- HSQLDB como BD embutido na memória para fins de teste.
- Jetty como *container web* embutido na memória para fins de testes.
- Cobertura para métricas de cobertura de testes.
- *Spring framework* para ligação dos diferentes tipos de *fixtures* de teste (com *mocks*, sem *mocks*, com banco de dados externo, com banco de dados na memória, etc).

Em nossos produtos mais sofisticados (da perspectiva do TDD) nós automatizamos os testes e aceitação fechados. Esses testes carregam o sistema todo na memória, incluindo bases de dados e servidores *web*, e acessam o sistema usando apenas as interfaces públicas (por exemplo HTTP).

Dessa forma nós aceleramos muito os ciclos de desenvolvimento-*build*-teste. Esse ambiente também funciona como uma rede de segurança, dando aos desenvolvedores a confiança suficiente para refatorar com freqüência, o que significa que o projeto mantem-se limpo e simples à medida que o sistema cresce.

TDD em códigos novos

Nós usamos o TDD para todo novo desenvolvimento, mesmo que o início do projeto demore mais (já que precisamos de mais ferramentas e suporte para os testes. É uma dose de no-*brainer*, os benefícios são tão grandes que não há nenhuma desculpa para não praticarmos o TDD.

TDD em códigos já existentes

TDD é difícil, mas tentar aplicá-lo numa base de código que não foi construída visando o TDD desde o princípio... é realmente muito difícil! Por que? Bem, na verdade, eu poderia escrever muito sobre isso, mas acho que vou parar aqui. Vou tratar disso num próximo artigo meu "TDD from the Trehches" :o)

Nós gastamos bastante tempo tentando automatizar os testes de integração num dos nossos sistemas mais complexos. Os códigos foram bastante mexidos e estavam bastante bagunçados e completamente desprovidos de teste.

Para cada versão do sistema nós temos uma equipe dedicada de testadores que fazem um conjunto completo de testes de regressão e performance. Os testes de regressão são normalmente trabalho manual. Isso atrasa bastante o ciclo de desenvolvimento e liberação da versão. Nosso objetivo é automatizar esses testes. Depois de termo batido nossas cabeças contra o muro alguns meses, não chegamos muito perto do que queríamos.

Depois disso, mudamos nossa abordagem. Nós aceitamos o fato de que estávamos estagnados nos testes manuais de regressão e, ao invés disso, deveríamos estar nos perguntando "Como posso fazer esse processo de testes manuais ficar mais rápido?" Isso era um jogo, e nós constatamos que muito trabalho de teste da equipe foi gasto fazendo tarefas triviais de instalação como navegar no *back office* para configurar *tournments* para os propósitos dos testes, ou esperando pelo agendamento de um *tournament* para começar. Então nós criamos utilidades para isso. Pequenas, de fácil acesso como atalhos e *scripts* que dispensavam o trabalho duro e deixava o foco nos testes efetivamente.

Esse esforço realmente se pagou. Na verdade, isso é provavelmente o que deveríamos ter feito desde o início. Nós estávamos tão ansiosos em automatizar os testes que esquecemos de fazer o passo a passo, onde o primeiro passo seria fazer os testes *manuais* mais eficientes.

Lição Aprendida: se você estiver estagnado com os testes manuais de regressão, e quiser automatizar desse jeito, não o faça (a menos que isso seja realmente muito fácil). Ao invés disso, construa coisas que façam os testes de regressão mais fáceis. *Aí sim* considere automatizar o teste efetivo.

Design incremental

Significa manter o design simples desde o início e melhorá-lo continuamente, ao invés de tentar deixá-lo perfeito desde o início e então congelá-lo.

Nós estamos indo muito bem assim, isto é, nós gastamos uma quantidade considerável de tempo melhorando o design existente e nós raramente perdemos tempo projetando de forma precipitada. Às vezes nós estragamos tudo, é claro, por exemplo quando deixamos que um *design* instável crie muitas raízes, de forma que refatorar se torna um grande projeto. Mas no geral estamos muito satisfeitos.

Melhoria contínua do *design* é praticamente um efeito coleteral do Desenvolvimento Orientado a Testes.

Integração contínua

A maioria dos nossos produtos possui uma configuração de integração contínua razoalvelmente sofisticada, baseada em *Maven* e *QuickBuild*. Isto é extremamente válido e poupa tempo. Essa é a solução definitiva para um velho problema "Ei, mas isso funciona na minha máquina". Nosso servidor de construção contínua age como o "juiz" ou ponto de referência a partir do qual se determina a qualidade de todas as nossas bases de código.

Toda vez que alguém atualiza alguma coisa no sistema de controle de versão o servidor de construção contínua irá acordar, construir tudo desde o início num servidor compartilhado e realizar todos os testes. Se alguma coisa der errado, ele enviará um email avisando a equipe inteira que a construção falhou, incluindo informações sobre exatamente qual mudança de código quebrou a construção, *links* para relatórios de teste, etc.

Todas as noites o servidor de construção contínua irá reconstruir o produto desde o início e publicar binários (ears, wars, etc), documentações, relatórios de teste, relatórios de cobertura de testes, relatórios de dependências, etc, no nosso portal de documentações internas. Alguns produtos serão também automaticamente implementados em um ambiente de teste.

Configurar isso deu *muito* trabalho, mas valeu cada minuto.

Propriedade coletiva do código

Nós encorajamos a propriedade coletiva do código mas ainda nem todas as equipes têm adotado esse método de trabalho. Descobrimos que a programação em pares (*pair programming*) com a mudança freqüente dos pares leva automaticamente a um maior nível de propriedade coletiva de código. As equipes com essa característica têm provado ser muito robustas, o *sprint* delas, por exemplo, não morre apenas porque alguém importante ficou doente.

Ambiente de trabalho informativo

Todas as equipes têm acesso aos quadros brancos e a uma parede vazia e fazem um bom uso desses recursos. Na maioria das salas você encontrará as paredes preenchidas com todo tipo de informação sobre o produto e o projeto. Assim, o maior problema passa a ser o lixo antigo acumulado nas paredes. Por isso deveríamos introduzir o papel do "faxineiro" em cada equipe.

Encorajamos o uso do quadro de tarefas, mas nem todas as equipes ainda o adotaram. Veja na página 59 "Como nós organizamos a sala da equipe".

Padrão de codificação

Recentemente nós começamos a definir o nosso padrão de codificação. Teria sido muito útil se tivéssemos feito isso antes. Não leva quase tempo nenhum. Comece com ele simples e vá crescendo conforme a necessidade. Escreva apenas regras que não sejam óbvias para todo mundo e correlacione com algum material já existente, sempre que possível.

A maioria dos programadores tem seu estilo próprio de codificação. São pequenos detalhes como, por exemplo, a forma de tratar exceções, como comentar o código, quando retornar valor nulo, etc. Em alguns casos as diferenças não causam problemas, mas em outros casos pode levar ao projeto de um sistema extremamente inconsistente e um código fonte difícil de entender. Um padrão de codificação aqui é muito útil, já que você foca em coisas que realmente importam.

Aqui estão alguns exemplos do nosso padrão de codificação:

- Você pode quebrar qualquer uma dessas regras, mas tenha a certeza que é por uma boa razão e documente isso.

- Use a convenção de codificação da Sun como padrão: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
- Nunca, jamais, nem pense em capturar exceções sem logar o *stack trace* ou *rethowing*. log.debug() é uma boa, mas não perca o stack trace.
- Use a dependência baseada em métodos setter para desacoplar as classes umas das outras. (exceto quando um acoplamento forte for desejável).
- Evite abreviações. Abreviações bem conhecidas como DAO são bem-vindas.
- Os métodos que retornam *collections* ou *arrays* não devem retornar nulo. Retorne coleções ou *arrays* vazios ao invés de nulo.

Rítimo Sustentável / Trabalho energizado

Muitos livros sobre desenvolvimento ágil de software dizem que horas-extras são anti-produtivas no desenvolvimento de software.

Após algumas experiências relutantes com isso eu só posso concordar plenamente com eles!

Há aproximadamente um ano atrás uma de nossas equipes (a maior delas) estava trabalhando horas extras excessivas.
A qualidade do software legado era horrível e eles tinham que gastar boa parte do tempo apagando incêndios. A equipe de teste (que também estava fazendo horas extras) não tinha chance de fazer nenhuma garantia de segurança. Nossos usuários estavam furiosos e os relatórios internos queriam comer-nos vivos.

Depois de alguns meses nós tinhamos negociado cargas horárias menores e mais decentes. Pessoas trabalhando cargas horárias normais (exceto durante alguns picos eventuais no projeto). E, surpresa, produtividade e qualidade notáveis.

Claro, reduzir a carga horária não foi o *único* aspecto que nos levou à essa melhoria, mas nós todos estávamos convencidos que foi um dos grandes responsáveis.

14

Como fazemos testes

Essa é a parte mais difícil. Não tenho certeza se é a parte mais difícil do *Scrum* ou do desenvolvimento de software em geral.

A etapa de testes é, provavelmente, a que mais varia de uma empresa para outra. Ela depende de quantos testadores você tem, de quão automatizados são os testes, de que tipo de sistema você tem (apenas servidor+aplicação web? ou na verdade você entrega pacotes de software?), do tamanho dos ciclos de *release*, quão crítico é o sistema (servidor de *blog* versus controle de tráfego aéreo), etc.

Nós temos feito muitos experimentos em como fazer testes com *Scrum*. Tentarei descrever o que temos feito e o que aprendemos até agora.

Você provavelmente não vai conseguir eliminar a fase dos testes de aceitação

No mundo *Scrum* ideal, um *sprint* resulta em um versão potencialmente instalável do seu sistema. Então é só instalar, certo?

Errado.

Nossa experiência é que isso na verdade não funciona. Haverá *bugs* desagradáveis. Se a qualidade tem qualquer tipo de valor para você, é necessário algum tipo de fase de testes de aceitação manuais. É nessa hora que testadores dedicados que *não* fazem parte da sua equipe martelam o sistema com aqueles tipos de testes que a equipe *Scrum* não imaginou, ou não teve tempo para fazer, ou não tiveram o *hardware* necessário para fazer. Estes testadores acessam o sistema exatamente da mesma forma que os usuários finais, o que significa que eles devem ser realizados manualmente (assumindo que seu sistema seja feito para usuários humanos).

A equipe de testes encontrará *bugs*, a equipe *Scrum* terá que produzir releases para correção de *bugs* e mais cedo ou mais tarde (espera-se que mais cedo) você estará apto para liberar uma versão 1.0.1 sem *bugs* aos usuários finais, ao invés de entregar uma versão 1.0.0 instável.

Quando eu digo "fase de testes de aceitação" eu estou me referindo a todo o período de teste, *debug* e re-*release* até que exista uma versão boa o bastante para ser colocada em produção.

Minimize a fase de testes de aceitação

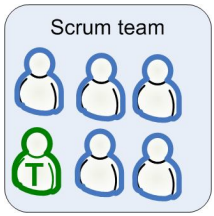
A fase de testes de aceitação machuca. Ela é principalmente não-ágil. Apesar de não podermos eliminá-la, nós podemos (e devemos) minimizá-la. Mais especificamente, minimizar a quantidade de tempo necessária para a fase de testes de aceitação. Isso pode ser afeito:

- Maximizando a qualidade do código entregue pela equipe *Scrum*
- Maximizando a eficiência do trabalho de teste manual (isto é, encontrar os melhores testadores, dar-lhes as melhores ferramentas, certificar-se de que eles lhe avisam quando houver tarefas que consomem muito tempo e que podem ser automatizadas

Então, como nós maximizamos a qualidade do código entregue pela equipe *Scrum*? Bem, existem diversas maneiras. Aqui estão duas que nós achamos que funcionam muito bem:

- Coloque os testadores na equipe *Scrum*
- Faça menos a cada *sprint*

Aumente a qualidade colocando os testadores na equipe Scrum



Sim, eu costumo escutar as duas objeções:

- *"Mas isso é óbvio! Equipes Scrum devem ser multi-funcionais!"*
- *"Equipes Scrum não devem possuir cargos! Nós não podemos ter um cara que é apenas*

testador!"

Deixe-me explicar. O que eu quero dizer com "testador" neste caso é ter "uma pessoa cuja habilidade primária é testar", ao invés de "uma pessoa cuja função é apenas testar"

Desenvolvedores são quase sempre maus testadores. Especialmente desenvolvedores que testam o seu próprio código.

▪ O testador é "o cara que aprova"

Além de ser "apenas" mais um membro da equipe, o testador tem uma importânte função. Ele é o cara que aprova. Nada é considerado 'pronto" em um *sprint* até que *ele diga* que está. Eu descobri que desenvolvedores frequentemente dizem que algo está pronto quando na verdade não está. Mesmo que você tenha uma definição de "pronto" muito clara (o que realmente deveria ser o caso, veja página 34 "Definição de pronto"), desenvolvedores freqüentemente irão se esquecer disso. Nós programadores somos pessoas impacientes e queremos sempre passar para a próxima etapa o mais rápido possível.

Então como o Sr. T (nosso testador) sabe que algo está pronto? Bem, antes de mais nada, ele deve (surpresa!) testar! Em muitos casos algo que o desenvolvedor considerou como pronto não era nem mesmo possível de ser testado! Seja porque o código não foi adicionado ao sistema de controle de versões, ou porque não foi instalado no servidor de testes, ou não pôde ser inicializado, ou qualquer outra causa. Assim que o Sr. T tiver testado a funcionalidade, ele deve repassar a lista de "pronto" com o desenvolvedor (caso você tenha uma). Por exemplo, se a definição de "pronto" diz que deve existir uma nota de release, então o Sr. T verifica se existe uma nota de release. Se houver algum tipo de especificação mais formal para esta funcionalidade (o que é raro no nosso caso) então o Sr. T também verifica isso, etc.

Um interessante efeito colateral disso é que agora a equipe tem uma pessoa que é perfeitamente apta para organizar a apresentação do *sprint*.

O que o testador faz quando não há nada para testar?

Essa pergunta sempre vem à tona. Sr. T: "*Scrum master*, não tenho nada para testar agora, o que mais eu posso fazer?". Ainda vai demorar uma semana para que a equipe termine a primeira estória. Então, o que o testador poderia fazer *nesse* período?

Bem, antes de tudo, ele deveria estar *se preparando para os testes*. Ou seja, escrevendo as especificações de teste, preparando um ambiente de testes, etc. Assim, quando um desenvolvedor liberar alguma funcionalidade para ser testada, não haveria nenhuma espera. O sr. T poderia começar o teste imediatamente.

Se a equipe está usando TDD, as pessoas ocupam tempo escrevendo códigos de teste desde o primeiro dia. O testador deveria fazer par com os desenvolvedores que estão escrevendo esses códigos de teste. Se o testador não sabe programar, ele ainda assim deveria fazer par com os desenvolvedores, a menos que ele apenas navegue e deixe o desenvolvedor digitar. Um bom testador normalmente adiciona tipos diferentes de testes aos de um bom desenvolvedor. Assim, um complementa o outro.

Se a equipe não está usando TDD, ou se não há casos de teste suficientes para ocupar todo o tempo do testador, ele deveria simplesmente fazer o possível para ajudar a equipe alcançar o objetivo do *sprint*. Igual a qualquer outro membro da equipe. Se o testador sabe programar então é ótimo. Se não, sua equipe terá que identificar as tarefas extra programação que devem ser feitas no *sprint*.

Quando você quebrar as estórias em tarefas, durante a reunião de planejamento do *sprint*, a equipe tende a focar nas *tarefas de programação*. Entretanto, normalmente existem muita outras que não são programação e que também precisam ser feitas no *sprint*. Se você investir tempo tentando *identificar as tarefas que não são programação* durante a fase de planejamento do *sprint*, as chances de o sr. T contribuir são muito grandes, mesmo que ele não programe e que não haja nenhum teste para fazer agora.

Exemplos de tarefas necessárias ao *sprint*, mas que não são programação:

- Preparar um ambiente de teste.
- Esclarecer requisitos.
- Discutir detalhes de *deploy* com a operação.
- Escrever documentos de *deploy* (*release notes*, RFC, ou qualquer outro que sua empresa use).
- Contatar recursos externos (projetistas de interface com usuário, por exemplo).
- Melhorar os *scripts* de *build*.
- Mais quebras de estórias em tarefas.
- Identificar perguntas-chave dos desenvolvedores e conseguir a resposta para elas.

Por outro lado, o que fazer se o sr. T tornar-se um gargalo? Vamos supor que estejamos no último dia do *sprint* e de repente muita coisa está pronta e o sr. T não teve a oportunidade de testar tudo. O que fazer? Bem, vamos transformar todo o resto da equipe em ajudantes do sr. T. Ele decide qual funcionalidade ele próprio precisa fazer e delega as outras para o restante da equipe. Isso é o que significa ter uma equipe multi-disciplinar!

Resumindo, sim, o sr. T tem um papel especial na equipe, mas ele também pode fazer outras coisas. E os outros membros da equipe também podem fazer o trabalho dele.

Aumentar a qualidade fazendo menos por Sprint

Voltando à reunião de planejamento do *sprint*. Colocando de forma simples, não coloque muitas estórias no *sprint*! Se você tem problemas de qualidade, ou longos ciclos de aceitação, faça menos por *sprint*! Isso irá meio que automaticamente elevar a qualidade. Menores ciclos de teste para aceitação, menos bugs afetando o usuário, e maior produtividade ao longo da caminhada. Isso pois a equipe poderá focar nas novas coisas ao invés de ficar consertando coisas velhas que vivem quebrando.

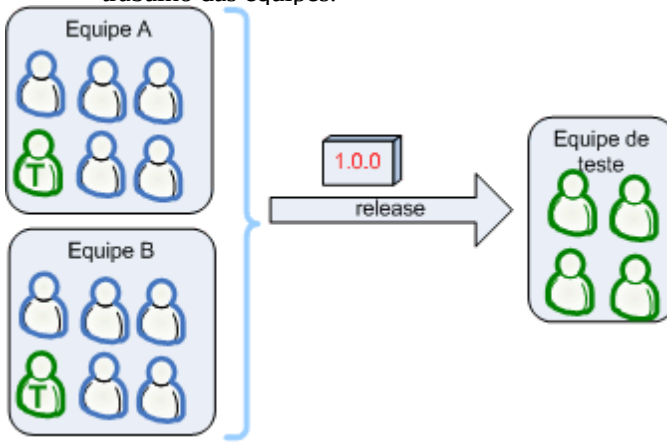
Quase sempre é mais barato construir menos, mas construir algo estável. Melhor do que construir montes de coisas que te deixam em pânico com consertos de urgência.

O teste de aceitação deveria fazer parte do sprint?

Nós variamos muito aqui. Algumas de nossas equipes incluem o teste de aceitação no *sprint*. Porém, a maioria delas, não. Por duas razões:

- Um *sprint* tem um limite de tempo. Testes de aceitação (Usando minha definição que inclui *debugar* e re-instalar), são muito difíceis de se mensurar o tempo. O que fazer se o tempo se esgotar e você ainda tiver um *bug* crítico? Você vai lançar uma *release* com um *bug* crítico? Você vai esperar até o próximo *sprint*? Na maioria dos casos ambos são inaceitáveis. Então deixe o teste manual de aceitação de lado.
- Se você tem várias equipes trabalhando no mesmo produto, o teste manual de aceitação deve ser seguido combinado com o trabalho de ambas equipes. Se ambas fizeram o teste manual de aceitação dentro do tempo do *sprint*, você ainda precisa de uma

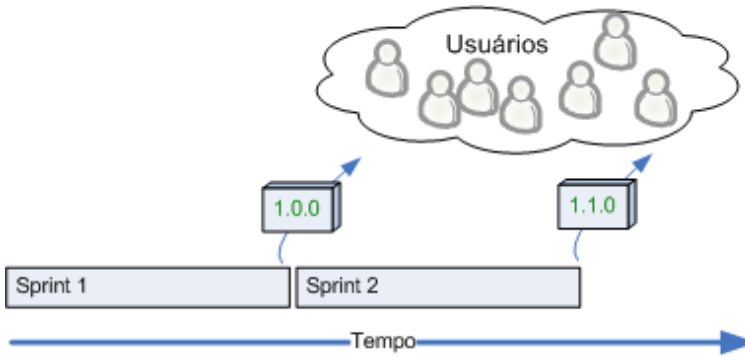
equipe para fazer o teste final de release, que é a integração do trabalho das equipes.



Isso pode não significar a solução perfeita mas é boa o suficiente pra gente na maioria dos casos.

Ciclos de sprint vs. ciclos de testes de aceitação

Idealmente, no mundo perfeito do *Scrum*, você não precisa de testes de aceitação, uma vez que as releases geradas ao final de cada *sprint* já estão prontas para a produção.

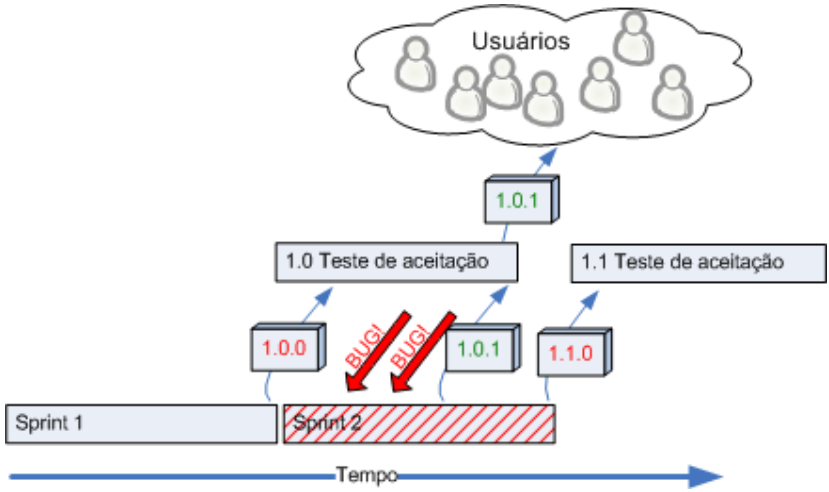


Bem, não é bem assim. Segue um esquema mais realista:

Após o *sprint* 1, uma release bugada 1.0.0 é lançada. Daí, durante o *sprint* 2, os *bugs* da última *release* começam a ser notificados, a equipe gasta grande parte do tempo corrigindo-os e é forçada a lançar uma versão intermediária 1.0.1 no meio do sprint 2. Então, ao final do *sprint* 2, uma release 1.1.0 com novas funcionalidades é disponibilizada. Provavelmente esta versão 1.1.0 estará ainda mais bugada que a release 1.0.0, já que o tempo nela investido foi prejudicado devido à necessidade urgente de correção da primeira release 1.0.0, e isso sempre se repetirá.

As linhas diagonais em vermelho da figura simbolizam o caos.

Não fica muito bonito, né? A má notícia, entretanto, é que este problema permanecerá mesmo se você tiver uma equipe só para testes de aceitação de releases. A única diferença será que a maioria dos relatos de *bugs* virão da equipe de teste, e não de usuários raivosos. Sob a perspectiva de negócios, isso já faz uma grande diferença. Para desenvolvedores, contudo, dá na mesma. Talvez a única exceção seja o fato de testadores serem menos agressivos que usuários. Geralmente.



Não encontramos ainda nenhuma solução simples para este problema, apesar de já termos experimentado diferentes modelos.

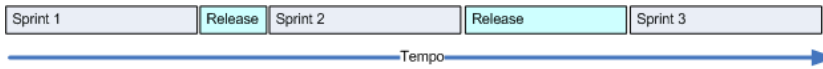
Primeiro de tudo, novamente, maximize a qualidade do código que a equipe gera. O custo de encontrar e corrigir *bugs* cedo, durante o *sprint*, é extremamente menor se comparado ao mesmo custo após o término da iteração.

De qualquer forma, permanence o mesmo fato. Ainda que minizemos a quantidade de *bugs*, haverá registros de *bugs* após o *sprint* ter sido finalizado. Como lidar com isso?

Abordagem 1: "Não inicie a construção de coisas novas antes das coisas antigas estarem em produção"

Parece bom não? Você também sentiu aquele sentimento aconchegante de incerteza?

Nós estivemos próximos de adotar esta abordagem diversas vezes, e desenhamos modelos requintados sobre como poderíamos fazer isto. Entretanto sempre mudamos de idéia quando visualizamos o lado negativo disto. Nós teríamos que adicionar um período de liberação sem um espaço de tempo fixo entre cada *sprint*, onde faríamos apenas teste e depuração até que pudéssemos realizar um release em produção.



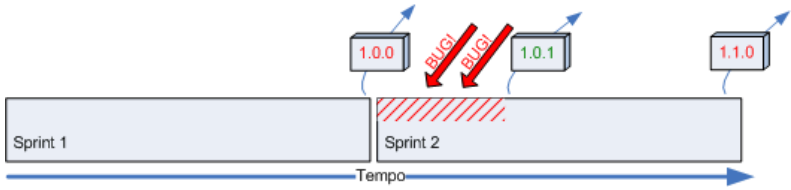
Nós não gostamos da noção de ter períodos sem espaço de tempo fixo entre os *sprints*, principalmente porque isto iria quebrar o ritmo que a equipe possui nos *sprints*. Nós não poderíamos mais dizer que "a cada 3 semanas nós iniciamos um novo *sprint*". Além do que, isto não resolve completamente o problema. Mesmo que tivéssemos um período de *release*, vão aparecer relatos de defeitos urgentes de tempos em tempos, e precisamos estar preparados para tratar deles.

1. Abordagem 2: "OK para começar a implementar novas funcionalidades, mas priorizar colocar as coisas antigas em produção"

Esta é nossa abordagem preferida. Pelo menos até agora.

Basicamente quando terminamos um *sprint*, nós vamos para o próximo. Mas esperamos gastar algum tempo deste próximo *sprint* consertando coisas do último *sprint*. Se os próximos *sprints* se tornarem severamente prejudicados por causa do gasto excessivo de tempo na correção de *bugs* do último *sprint*, nós avaliamos por que isto aconteceu e como podemos melhorar a qualidade. Nós procuramos ter a certeza de que *sprints* são duradouros o suficiente para não serem impactados por um punhado de tempo gasto concertando *bugs* do *sprint* anterior.

Gradualmente, após um período de muitos meses, a quantidade de tempo gasto consertando *bugs* do *sprint* anterior diminuiu. Assim nos tornamos hábeis em escalar menos pessoas envolvidas quando estes *bugs* acontecem, e nem *toda* equipe precisa ser incomodada todas as vezes. Agora estamos em um nível mais aceitável.



Durante as reuniões de planejamento do *sprint*, nós ajustamos o fato de foco baixo o suficiente para considerar o tempo que gastamos concertando os *bugs* do último *sprint*. Com o tempo, as equipes se tornaram bastante eficazes em fazer esta estimativa. A métrica de velocidade ajuda muito (veja página 25 "Como a equipe decide quais estórias são incluidas no sprint?")

Abordagem ruim – "foco em construir novas coisas"

Este efeito significa "foco em construir novas coisas *ao invés de colocar as coisas antigas em produção*". Quem gostaria de fazer isto? Mesmo assim cometemos este erro de vez em quando no começo, e eu tenho a certeza de que muitas empresas fazem o mesmo. É uma doença relacionada a pressão do projeto. Muitos gerentes realmente não compreendem que quando toda a codificação terminar, você geralmente estará longe do *release* de produção. Pelo menos para sistemas complexos. Então o gerente (ou *product owner*) pede para a equipe continuar adicionando novas funcionalidades enquanto a mochila de código das coisas "quase prontas" se torna cada vez mais e mais pesada, diminuindo a velocidade de tudo.

Não exponha o elo mais fraco de sua corrente

Digamos que o teste de aceitação seja o seu elo mais fraco. Você tem pouquíssimos testadores ou o período de testes demora mais devido à baixa qualidade do código.

Digamos que a equipe de testes de aceitação possa testar no máximo 3 funcionalidades por semana (não, nós não usamos "funcionalidades por semana" como métrica; só estou dando um exemplo). E digamos que seus desenvolvedores produzam 6 novas funcionalidades por semana.

É tentador para os gerentes ou *product owners* (ou talvez mesmo para a equipe) prever o desenvolvimento de 6 novas funcionalidades por semana.

Não faça isso! A realidade irá pegá-lo de um modo ou de outro. E vai machucá-lo.

Ao invés disso, planeje 3 novas funcionalidades por semana e invista o restante do tempo para aliviar o gargalo de testes. Exemplo:

- Aloque alguns desenvolvedores como testadores (Ah, eles irão amá-lo por isso…).
- Implemente ferramentas e *scripts* que facilitem os testes.
- Adicione mais código que automatize os testes.
- Aumente o tamanho do *sprint* e tenha os testes de aceitação incluídos nele.
- Defina alguns *sprints* como "*sprints* para teste" onde a equipe inteira atuará como uma equipe de testes de aceitação.
- Contrate mais testadores (mesmo que isso signifique dispensar alguns desenvolvedores).

Nós temos usado todas essas alternativas (exceto a última). As melhores soluções a longo prazo são obviamente a 2ª e 3ª, isto é, melhores ferramentas e *scripts* para automação dos testes

Análises retrospectivas são um bom fórum para identificar o elo mais fraco de sua corrente.

De volta à realidade

Provavelmente eu tenho passado a impressão de que nós temos testadores em todas as equipes, que temos uma enorme equipe de testes de aceite para cada produto, que entregamos depois de cada *sprint*, etc, etc.

Bem, nós não temos.

Nós *algumas vezes* gerenciamos projetos com testes, e nós temos visto efeitos positivos. Mas ainda estamos longe de um aceitável processo de garantia de qualidade, e ainda temos muito a aprender.

15

Como lidamos com várias equipes

Muitas coisas se complicam quando se tem várias equipes trabalhando no mesmo produto. Esse problema é universal e não tem nada a ver com o *Scrum*. Mais desenvolvedores = mais complicações.

Nós temos (como sempre) experiência com isso. No máximo, nós temos aproximadamente 40 pessoas trabalhando no mesmo produto.

A questões chaves são:

- Quantas equipes criar
- Como alocar pessoas nas equipes

Quantas equipes criar

Se lidar com várias equipes é tão dificil, por que vamos nos incomodar? Por que simplesmente não colocar todos na mesma equipe?

A maior equipe de scrum que nós tivemos tinha aproximadamente 11 pessoas. Ela funcionava, mas não muito bem. Reuniões diárias tendiam a durar mais de 15 minutos. Os membros da equipe não sabiam o que os membros das outras equipes estavam fazendo, e isso gerava confusão. Era muito dificil para o *scrum master* manter todos alinhados ao objetivo, e dificil de encontrar tempo de endereçar todos os obstáculos que eram reportados.

A alternativa era dividir em duas equipes. Mas isso é melhor? Não necessariamente.

Se a equipe tem experiência e é confortável com o *Scrum*, e há um jeito lógico de dividir o *roadmap* em duas pistas diferentes, e essas duas pistas não involvem o mesmo código fonte, então eu deveria dizer que é uma boa idéia dividir a equipe. De outra forma eu optaria por manter a equipe única, a despeito das desvantagens de uma grande equipe.

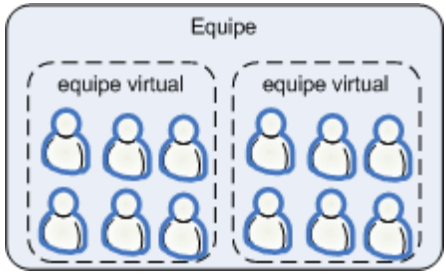
Minha experiência diz que é melhor ter menos equipes, embora grandes, do que ter muitas equipes pequenas que interfiram umas nas outras. Faça pequenas equipes apenas se não forem interferir umas nas outras!

▪ Equipes virtuais

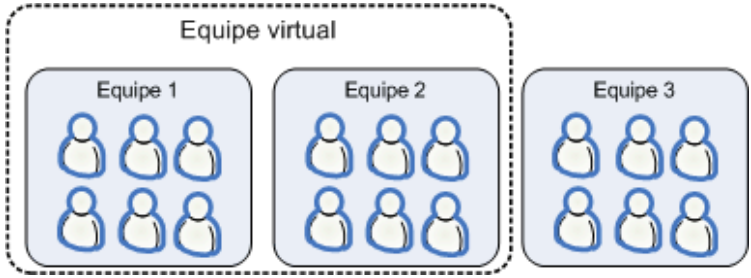
Como você sabe se tomou a decisão certa ou errada em relação aos prós e contras de ter uma "equipe grande" ou uma "equipe pequena"?

Se você manteve seus olhos e ouvidos abertos você deve ter notado o termo "equipes virtuais".

Exemplo 1: Você escolhe ter uma equipe grande. Mas quando você começa a observar quem fala com quem durante o *sprint*, percebe que a equipe acabou se divindindo em duas sub-equipes.



Exemplo 2*:* Você escolhe ter três equipes menores. Mas quando você começa a observar quem fala com quem durante o *sprint*, percebe que as equipes 1 e 2 estão conversando entre si o tempo todo, enquanto que a equipe 3 está trabalhando isoladamente.



O que isso significa? Que sua estratégia de divisão de equipes estava errada? Sim, se as equipes virtuais parecerem permanentes. Não, se as equipes virtuais parecerem temporarias.

Olhe novamente para o exemplo 1. Se as duas sub-equipes virtuais tendem a mudar de vez em quando (por exemplo, as pessoas vão de uma sub-equipe para outra) então provavelmente você tomou a decisão certa ao mantê-las como uma única equipe. Se as duas sub-equipes virtuais mantêm-se as mesmas durante todo o *sprint*, talvez você queira separá-las em duas equipes reais no próximo *sprint*.

Agora olhe novamente para o exemplo 2. Se as equipes 1 e 2 estão conversando entre si (e não com a equipe 3) durante todo o *sprint*, talvez você queira combinar as equipes 1 e 2 em uma única equipe no próximo *sprint*. Se as equipes 1 e 2 estão conversando muito entre si durante a primeira metade do *sprint* e então as equipes 1 e 3 começam a conversar durante a segunda metade do *sprint*, então você deve considerar a possibilidade de combinar as três equipes em uma, ou então deixá-las como três equipes. Cite esta questão durante a retrospectiva do *sprint* e deixe que as equipes decidam por si.

Divisão de equipes é das partes realmente difíceis do *Scrum*. Não pense ou se esforce demais em otimização. Experimente, observe as equipes virtuais e certifique-se que você tem tempo suficiente para discutir esse tipo de coisa durante as retrospectivas. Mais cedo ou mais tarde você encontrará a solução correta para o seu caso. O importante é que as equipes estejam confortáveis e não fiquem trombando umas na outras com freqüência.

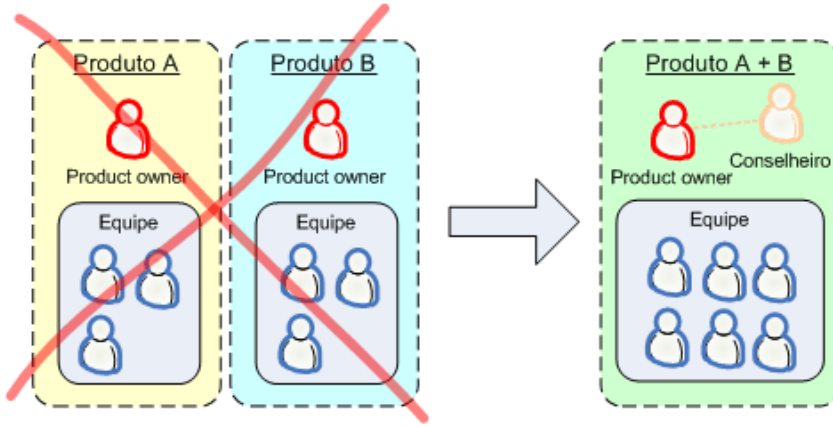
▪ Tamanho de equipe ideal

A maioria dos livros que eu li dizem que o tamanho de equipe "ideal" está em algum ponto entre 5 e 9 pessoas.

Do que eu vi até agora a única coisa que eu posso fazer é concordar. Apesar de que eu diria de 3 a 8 pessoas. Na verdade, eu acredito que vale à pena pagar o preço para chegar em uma equipe deste tamanho.

Digamos que você tem uma única equipe com 10 pessoas. Considere a possibilidade de retirar da equipe os dois membros mais fracos. Oops, eu disse isso?

Digamos que você tenha dois produtos diferentes, com uma equipe de três pessoas por produto, e as duas estão lentas. Seria uma boa idéia combiná-las em uma única equipe de 6 pessoas responsável pelos dois produtos. Neste caso retire um dos dois *product owners* da função (ou lhe dê outra função de consultoria ou algo do tipo).

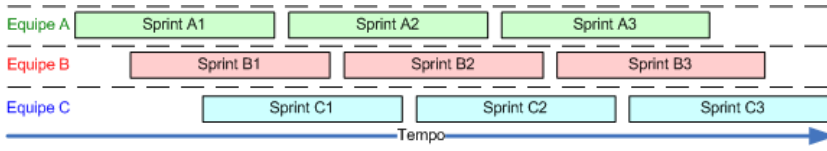


Digamos agora que você possui uma única equipe com 12 pessoas, porque a base de código está tão ruim que não há como duas equipes diferentes trabalharem independentemente. Não poupe esforços em consertar a base de código (ao invés de criar novas funcionalidades) até que você chegue a um ponto onde é possível dividir a equipe. Rapidamente este investimento irá compensar.

Sincronizar sprints – ou não?

Digamos que você tenha três equipes trabalhando no mesmo produto. Os *sprints* delas deveriam ser sincronizados, p.ex. começar e terminar juntos? Ou eles deveriam se sobrepor?

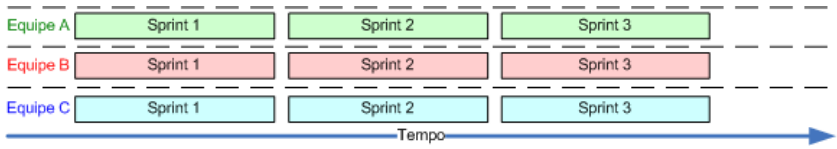
Nossa primeira escolha foi ter sprints sobrepostos (com respeito ao tempo).



Essa situação parece boa. A qualquer momento haveria um *sprint* perto de terminar e um outro quase começando. O trabalho do *product owner* estaria diluído durante todo o tempo. Haveria releases jorrando continuamente do sistema. Apresentações toda semana. Aleluia!

Esse cenário realmente me pareceu convincente!

Nós começamos assim, até que um dia eu tive a oportunidade de conversar com Ken Schwaber (na época de minha certificação em *Scrum*). Ele me mostrou que essa era uma *péssima* idéia, que seria muito melhor sincronizar os *sprints*. Eu não lembro direito os motivos, mas depois de debatermos um pouco, ele me convenceu.

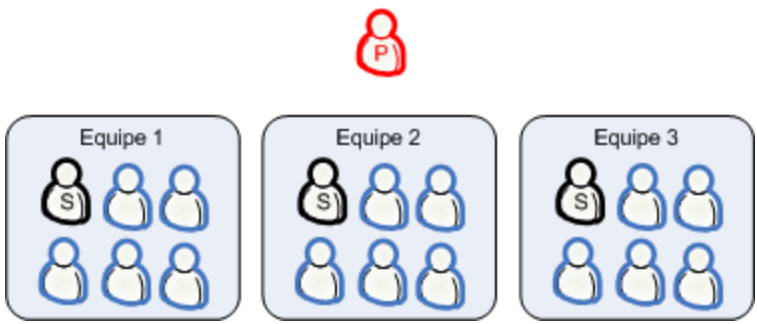


Essa é a solução que temos usando desde então, e eu nunca me arrependi disso. Eu nunca saberei se os *sprints* sobrepostos funcionariam, mas acho que não. As vantagens dos *sprints* sincronizados são:

- Existe naturalmente um tempo para rearranjar as equipes – entre os *sprints*! Com a sobreposição, não há como rearranjar uma equipe sem atrapalhar pelo menos uma outra, no meio do *sprint*.
- Todas as equipes podem trabalhar com o mesmo objetivo num *sprint* e fazer as reuniões de planejamento juntas, o que promove melhor colaboração entre as equipes.
- Menos sobrecarga administrativa, p.ex., menos reuniões de planejamento, *sprint* demos e releases.

Por que criamos o papel de "Líder de Equipe"

Digamos que temos um único produto e três equipes.



O sujeito de vermelho (P) é o *product owner*. Os caras de preto (S) são os *Scrum masters*. Os outros são ~~peões~~... quer dizer... respeitáveis membros da equipe.

Com tantas estrelas no projeto, a quem cabe decidir quem fica em qual equipe? Ao *product owner*? Aos três *Scrum masters* juntos? Todo mundo escolhe em que equipe quer ficar? Mas e se todo mundo quiser ficar na equipe 1(porque a *Scrum master* 1 é *tão bonitinha*...)?

E se mais tarde chega-se a conclusão que é impossível ter mais do que duas equipes trabalhando em paralelo no mesmo código e precisarmos transformar as três equipes de seis pessoas em duas de nove. Ou seja, apenas dois *Scrum masters*. Qual dos três será devidamente exonerado?

Em muitas empresas essas são questões delicadas.

É extremamente tentador entregar a alocação e realocação dos recursos nas mãos do *product owner*. Mas isso não é coisa de *product owner*, certo? O *product owner* é o especialista no domínio que dá o norte a equipe. Ele não deve se envolver nos pequenos detalhes. Especialmente porque é um frango (se você não conhece a metáfora dos porcos e frangos digite "*chickens and pigs*" no Google).

Nós resolvemos essa questão criando um papel de "Líder de Equipe". Esse papel corresponde ao que chamamos de "*Scrum master* dos *Scrums masters*" ou "o chefão" ou "*Scrum master senior*" etc. Ele não lidera nenhuma equipe específica mas é responsável por resolver problemas entre as equipes, como quem será o *Scrum master* da equipe, como as pessoas devem ser divididas entre as equipes, etc.

Não foi fácil achar um nome para esse papel, "Líder de Equipe" foi o menos pior.

Essa solução funcionou para nós e podemos recomendá-la (independente de como você resolve chamar esse papel).

Como alocamos pessoas às equipes

Existem duas estratégias gerais para alocar pessoas às equipes, quando você tem várias equipes trabalhando no mesmo produto.

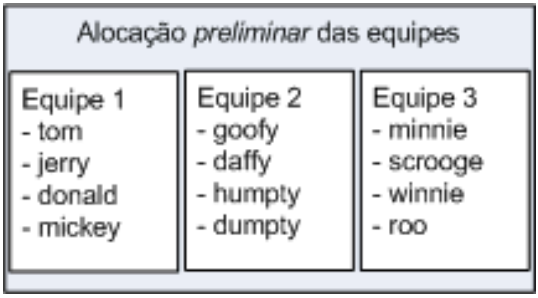
- Deixe uma pessoa designada para fazer a alocação. Por exemplo o "líder da equipe" citado acima, o *product owner*, ou o gerente funcional (se ele estiver envolvido o suficiente para tomar boas decisões a esse respeito)
- Deixe as equipes fazerem a escolha por elas próprias.

Nós temos experimentado todas as três opções. Três?! Sim. Estratégia 1, estratégia 2 e as duas combinadas.

Descobrimos que a combinação das duas tem melhor resultado.

Antes da reunião de planejamento do *sprint*, o líder da equipe convoca uma reunião de alocação das equipes junto ao *product owner* e todos os *Scrum masters*. Nós conversamos sobre o último *sprint* e decidimos se as realocações da equipes estão garantidas. Entretanto, podemos decidir combinar duas equipes, ou transferir alguém de uma equipe para outra. Nós tomamos uma decisão e anotamos como uma *proposta de alocação das equipes*, que levamos à reunião de planejamento do *sprint*.

A primeira coisa que nós fazemos na reunião de planejamento do *sprint* é atacar o *product backlog* na ordem das prioridades. O líder da equipe então diz algo como:
"Olá pessoal. Nós sugerimos a seguinte alocação das equipes para o próximo *sprint*."



"Como podem ver, isso significaria uma redução de 4 para 3 equipes. Nós relacionamos os membros de cada equipe. Por favor, agora juntem-se e arrumem uma seção na parede."

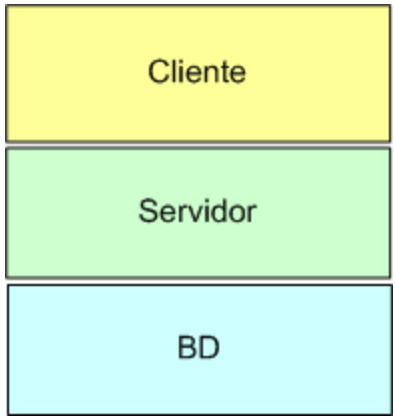
(o líder da equipe aguarda enquanto as pessoas ajeitam-se na sala. Depois de algum tempo haverá 3 grupos de pessoas, cada um deles perto de uma parte vazia na parede).

"Agora essa divisão das equipes é *uma prévia*! É só um ponta-pé inicial para ganharmos tempo. Durante o avanço dessa reunião vocês são livres para mudar de equipe, dividir sua equipe em duas, juntar com outra, o que vocês quiserem. Usem o bom senso baseado nas prioridades do projeto."

Isto é o que temos visto que melhor funciona. Um certo nível de controle centralizado no início, seguido de um certo nível de melhoria descentralizada, depois.

Equipes especializadas – ou não?

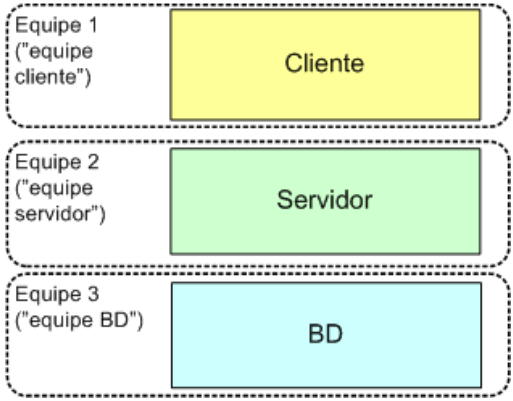
Digamos que sua tecnologia seja composta de 3 partes principais:



E digamos que você tenha 15 pessoas trabalhando neste produto, e você não gostaria de executá-lo como uma única equipe. Quais equipes você criaria?

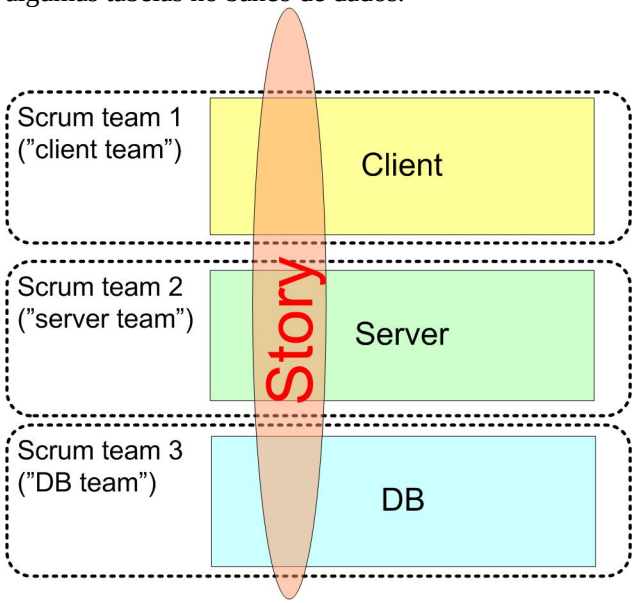
Abordagem 1: Equipes especializadas em componentes

Uma solução é criar equipes especializadas em componentes, como “equipe cliente”, “equipe servidor” e o “equipe BD”.



Foi assim que começamos a princípio. Não funciona muito bem, pelo menos não quando a maioria das estórias envolve múltiplas camadas.

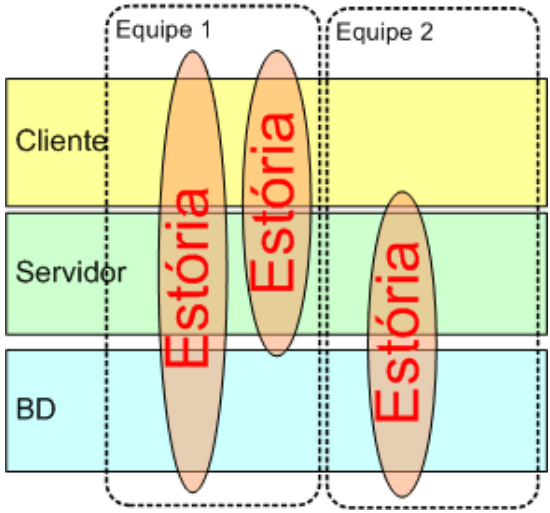
Por exemplo, digamos que nós tenhamos uma estória denominada "quadro de avisos onde pessoas possam colocar recados/mensagens para outras pessoas". Este quadro de avisos implicaria atualizar a interface gráfica no cliente, adicionar processamento lógico no servidor e adicionar algumas tabelas no banco de dados.



Ou seja, as três equipes – cliente, servidor e banco de dados – devem interagir e colaborar para finalizar a estória. E isso não é muito bom.

Abordagem 2: Equipes multifuncionais

Uma segunda abordagem é criar equipes multifuncionais, isto é, equipes que não estão amarradas a nenhum componente específico.



Se a maioria de nossas estórias envolvem múltiplos componentes, esta estratégia de divisão de equipes irá funcionar melhor. Cada equipe pode implementar uma estória completa, incluindo as partes do cliente, do servidor e do BD. Cada equipe pode desse modo trabalhar mais independente de cada outra, o que é uma Boa Coisa.

Uma das primeiras coisas que fizemos quando introduzimos o *Scrum* foi dissolver as equipes existentes especializadas em componentes (abordagem 1) e criar equipes multifuncionais em seu lugar (abordagem 2). Isto diminuiu o número de casos de "não pudemos completar este item porque estávamos esperando pelos caras do servidor fazer o trabalho deles."

Contudo, fazemos de vez em quando a criação de equipes temporárias especializadas em componentes, sempre que surge uma forte necessidade.

Reorganizar as equipes entre os sprints – ou não?

Cada *sprint* geralmente é bem diferente do outro., dependendo de quais tipos de estórias possuem maior prioridade naquele momento específico. Como resultado, a definição da equipe ideal pode ser diferente para cada *sprint*.

Na verdade, praticamente em todos os *sprints* nos vemos falando coisas como "este *sprint* realmente não é um sprint *normal* porque (blablabla) ...". Após um tempo nós desistimos da definição de *sprints* "normais". Não existem *sprints* normais. Assim como não há famílias "normais" ou pessoas "normais".

Em um *sprint* pode ser uma boa idéia ter uma equipe apenas de clientes, consistindo apenas de pessoas que conhecem bem a base de código do cliente. No próximo *sprint* pode ser uma boa idéia ter duas equipes e dividir o pessoal cliente entre elas.

Um dos aspectos chave do *Scrum* é o da "cola da equipe", ou seja, se uma equipe acaba trabalhando junta através de múltiplos *sprints* eles se tornarão *muito unidos*. Eles aprenderão a alcançar *fluência com o grupo* e atingirão um nível de produtividade incrível. Mas demora alguns *sprints* para que chegue neste nível. Se você ficar mudando as equipes você nunca alcançará uma forte cola da equipe.

Logo, se você pretende reorganizar as equipes, certifique-se de ter levando em consideração todas as consequências. Esta é uma mudança de curto ou longo prazo? Se esta for uma mudança de curto prazo, ignore-a. Se for uma mudança de longo prazo, faça isso.

Uma exceção é quando você começa a praticar *Scrum* com uma equipe grande pela primeira vez. Neste caso, provavelmente vale à pena experimentar um pouco com a subdivisão da equipe até que você encontre algo com que todos se sintam confortáveis. Tenha certeza de que todos entendão de não há problema algum se tudo der errado nas primeiras vezes, desde que você se mantenha sempre melhorando.

Membros de equipe em tempo parcial

Eu tenho que confirmar o que os livros de *Scrum* dizem – ter membros em tempo parcial em uma equipe *Scrum* geralmente não é uma boa idéia.

Digamos que você está pretendendo incluir o Joe como um membro em tempo parcial na sua equipe *Scrum*. Primeiro pense nisso com cuidado. Você realmente precisa do Joe na sua equipe? Você tem certeza de que não pode incluir o Joe em tempo integral? Quais são seus outros compromissos? Será que algúém pode assumir estes outros compromissos do Joe e fazer com que ele assuma uma função um pouco mais passiva ou de apoio em relação a este compromisso? Será que o Joe pode passar a

integrar sua equipe em tempo integral no *próximo sprint*, e neste intervalo transferir suas outras responsabilidades para outra pessoa?

Algumas vezes simplesmente não há solução. Você precisa desesperadamente do Joe porque ele é o único DBA no prédio, mas as outras equipes também precisam dele tanto quanto, logo ele nunca será alocado em tempo integral na sua equipe e a empresa não pode contratar mais DBAs. Tudo bem. Este é um caso válido para tê-lo por tempo parcial (o que a propósito é exatamente o que nos aconteceu). Mas tenha certeza que você está sempre fazendo essa avaliação.

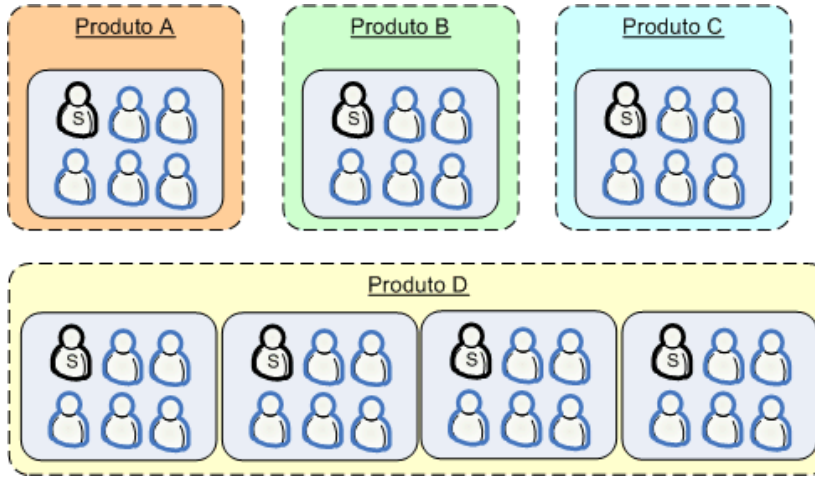
De modo geral, eu prefiro ter uma equipe com 3 membros em tempo integral do um equipe com 8 membros em tempo parcial.

Se você tem uma pessoa que irá dividir seu tempo entre multiplas equipes, como o DBA citado anteriormente, ainda assim é uma boa idéia tê-lo como membro primário em uma única equipe. Descubra qual a equipe que mais irá precisar dele e faça esta sua "equipe casa". Quando mais ninguém estiver precisando dele, ele estará presente nas reuniões diárias, reuniões de planejamento de *sprint*, retrospectivas e etc desta equipe.

Como fazemos Scrum-de-Scrums

Scrum-de-*Scrums* são basicamente reuniões regulares onde todos *Scrum masters* se encontram para conversar.

Em um ponto no tempo nós tínhamos quatro produtos, onde três dos produtos tinham apenas uma equipe cada, e o último produto tinha 25 pessoas divididas em várias equipes. Algo assim:



Isso significa que tínhamos dois níveis de *Scrum*-de-*Scrums*. Nós tínhamos um *Scrum*-de-Scrums de "nível de produto" consistindo de todas as equipes dentro do Produto D, e um Scrum-de-Scrums de "nível corporativo" consistindo de todos produtos.

Scrum-de-Scrums de nível de produto

Esta reunião foi muito importante. Nós a fazíamos uma vez por semana, algumas vezes mais do que isso. Discutíamos problemas de integração, de balanceamento de equipes, preparações para a próxima reunião de planejamento de *sprint*, etc. Nós alocávamos 30 minutos, mas freqüentemente passávamos disso. Uma alternativa seria termos tido *Scrum*-de-*Scrums* todos os dias, mas nunca chegamos a tentar isso.

Nosso planejamento de *Scrum*-de-*Scrums* era
1) Ao redor da mesa, todos descrevem o que sua equipe realizou na última semana, o que planeja alcançar nessa semana, e que impedimentos eles têm.
2) Quaisquer outras preocupações inter-equipes que precisem ser levantadas, como problemas de integração, por exemplo.

O planejamento de *Scrum*-de-*Scrums* não é realmente importante para mim, o importante é que você *faça* reuniões *Scrum*-de-*Scrums* regularmente.

Nível corporativo Scrum-de-Scrums

Nós chamamos esta reunião de "O Pulso". Nós temos feito essa reunião numa variedade de formatos, com uma variedade de participantes.

Ultimamente temos suprimido todo o conceito e, ao invés disso, substituído por uma reunião com todas as pessoas (bem, todas as pessoas envolvidas no desenvolvimento). 15 minutos

O quê? 15 minutos? Com todos? Todos os membros de todas as equipes de todos os produtos? Isto funciona?

Sim, isto funciona se você (ou quem conduz a reunião) for rigoroso quanto a mantê-la curta.

O formato da reunião é:

1) Notícias e atualizações do chefe de desenvolvimento. Informações sobre próximos eventos, por exemplo.
2) *Round-robin*. Uma pessoa de cada grupo de produtos relata o que eles fizeram na semana passada, o que eles planejam fazer essa semana e qualquer problema. Algumas outras pessoas também reportam (líder do gerenciameto de configuração, líder do controle de qualidade, etc)
3) Todos os outros são livres para adicionar qualquer informação ou fazer perguntas

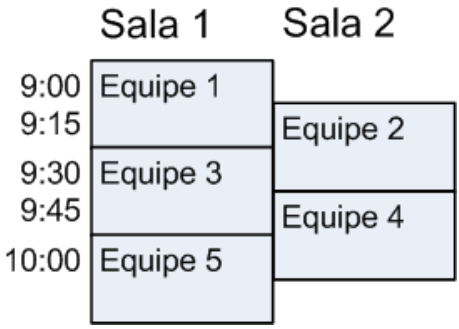
Esse é um fórum para informações rápidas, não discussões ou reflexões. Deixando-a assim, 15 minutos normalmente funciona. Algumas vezes nós estrapolamos, mas muito raramente mais que 30 minutos. Se discussões interessantes começam, eu faço uma pausa e convido todos os interessados para ficar depois do reunião e continuá-la.

Por que nós fazendo uma reunião de pulso "à-todas-mãos"? Porque nós notamos que o nivel corporativo *Scrum*-de-*Scrums* é mais para comunicação. Nós raramente temos discussões nesse grupo. Além disso, muitas outras pessoas fora do grupo estão famintos por esse tipo de informação. Basicamente, equipes querem saber o que outras equipes estão fazendo. Então nós notamos que, se estamos indo à reunião e gastando tempo informando cada um sobre o que cada equipe está fazendo, por que não simplesmente deixar todos freqüentarem?

Intercalando as reuniões diárias

Se você tem muitas equipes *Scrum* para um mesmo produto, e todos fazem a *reunião diária* ao mesmo tempo, você tem um problema. O *product owner* (e pessoas intrometidas como eu) só podem atender a *reunião diária* de uma equipe por dia.

Por isso pedimos que às equipes evitem ter reuniões diárias simultâneas.



A agenda de exemplo acima é de um período em que nós tínhamos *reuniões diárias* em salas separadas, ao invés de ser na sala da própria equipe. As reuniões duravam 15 minutos normalmente, mas cada equipe tem um período reservado de 30 minutos caso eles precisem estourar o tempo um pouquinho.

Isto é *extremamente* importante por dois motivos:

1. Pessoas como eu e o *product owner* podem ir a todas as *reuniões* diárias de uma manhã. Não há maneira melhor de se ter uma foto precisa de como a *sprint* está se saindo e quais são as potenciais ameaças.
2. As equipes podem visitar as *reuniões diárias* umas das outras. Não acontece com muita freqüência, mas de vez em quando as equipes estarão trabalhando em áreas semelhantes, então alguns membros da equipe atendem às reuniões umas das outras para que haja sincronia.

A parte ruim é que isso reduz a liberdade da equipe – eles não podem escolher o horário que quiserem para fazer suas *reuniões* diárias. Mas isto não tem sido realmente um problema para nós.

Equipes de bombeiros

Tivemos uma situação onde não era possível adotar *Scrum* em um grande produto porque gastávamos muito tempo apagando incêndios, ex: corrigindo exaustivamente falhas de uma versão prematura do sistema. Este era um verdadeiro ciclo vicioso. Ficávamos tão ocupados corrigindo falhas que não tínhamos tempo hábil para trabalhar proativamente na *prevenção* de novas falhas (exemplo: melhorias no desenvolvimento, automação de testes, criação de ferramentas de monitoramento, ferramentas de suporte, etc).

Nós resolvemos este problema criando uma equipe de bombeiros e outra equipe *Scrum* dedicada.

A função da equipe *Scrum* era (com as bençãos do *product owner*) tentar estabilizar o sistema e efetivamente prevenir os *"incêndios"*.
Já a equipe de bombeiros (chamávamos de "suporte", na verdade) tinha duas funções:

1) Apagar incêndios;
2) Proteger a equipe *Scrum* de todos os tipos de perturbação, incluindo coisas de como requisições ad-hoc vindas de lugar algum.

A equipe de bombeiros ficava disposta próxima à porta; a equipe *Scrum* ficava ao fundo da sala. Desta forma, os bombeiros poderiam proteger "*fisicamente*" a equipe *Scrum* de interferências tais como vendedores vorazes ou clientes raivosos.

Desenvolvedores experientes eram distribuídos nas duas equipes, desta forma uma equipe não seria tão dependente da outra.

Basicamente, esta foi uma tentativa de resolver este impasse. Como poderíamos começar a usar *Scrum* se a equipe não tinha sequer a chance de planejar mais que um dia de trabalho por vez? Nossa estratégia foi, como mencionado, dividir o grupo.

Esta solução funcionou muito bem. Quando a equipe *Scrum* teve uma sala para trabalhar proativamente, eles finalmente estabilizaram o sistema. Enquanto isso, a equipe de suporte desistiu completamente da ideia de planejar atividades futuras. Trabalharando reativamente, eles simplesmente consertavam qualquer defeito que aparecia.

Claro que a equipe *Scrum* não era totalmente *intocável*. Freqüentemente a equipe de suporte tinha que envolver pessoas-chave da equipe *Scrum*, ou pior, a equipe toda.

De qualquer forma, após alguns meses, o sistema estava suficientemente estável e conseguimos desfazer a equipe de suporte e criar equipes adicionais. Os membros da equipe de suporte (bombeiros) ficaram felizes em aposentar seus capacetes de batalha e juntar-se às equipes *Scrum*.

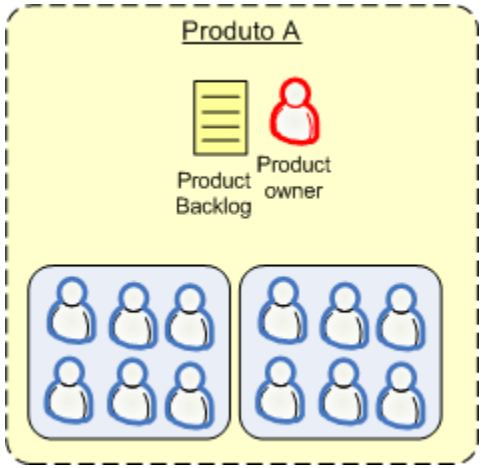
Dividir o product backlog – ou não?

Digamos que você tem um produto e duas equipes *Scrum*. Quantos *product backlogs* você deve ter? Quantos *product owners*? Nós analizamos três modelos para esse tipo de situação. A escolha tem um efeito realmente grande em como as reuniões de planejamento de *sprint* são conduzidas.

▪ Estratégia 1: Um product owner, um backlog

Este é o modelo "Só Pode Haver Um". Nosso modelo predileto.

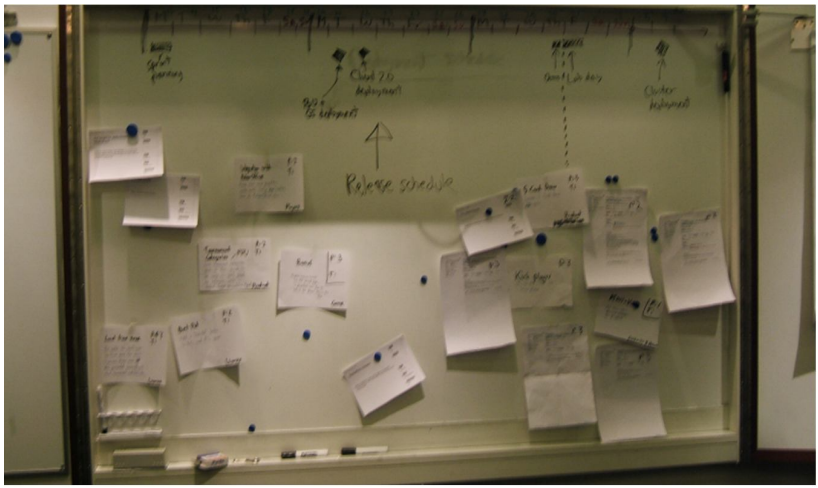
O bom deste modelo é que você pode deixar as equipes praticamente por conta própria, baseado nas prioridades do *product owner*. O *product owner* pode focar *no que ele precisa* e deixar as equipes decidirem como dividir o trabalho.



Para ser mais concreto, aqui está o modo como a reunião de planejamento de *sprint* funciona para esta equipe:

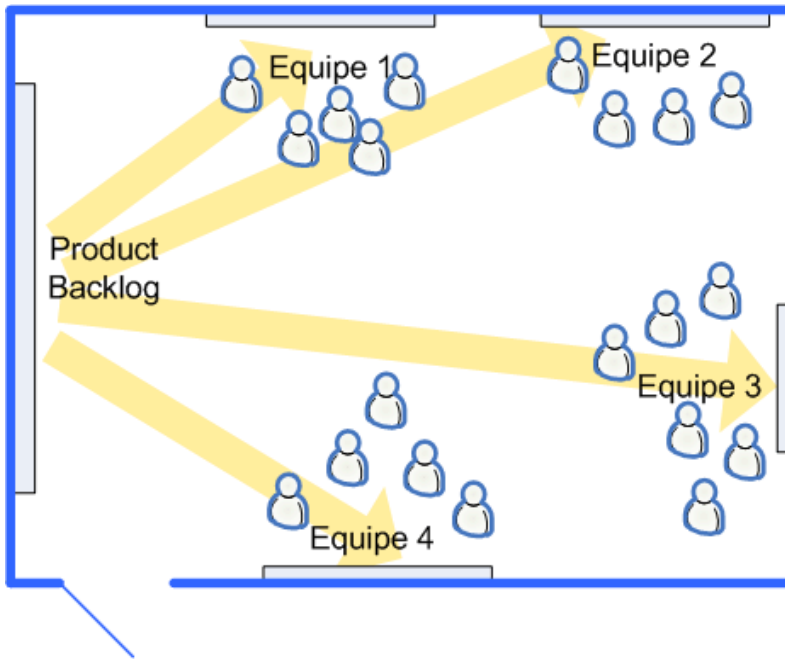
A reunião de planejamento de *sprint* ocorre em um centro de conferências externo.

Logo antes da reunião, o *product* owner define um quadro para ser o "quadro do *product backlog*" e coloca estórias ali (cartões), ordenadas por prioridade relativa. Ele continua colocando-as até que o quadro esteja cheio. Normalmente estes são itens mais do que suficientes para um *sprint*.



Cada equipe seleciona uma parte vazia do quadro e coloca seu nome ali. Este é o seu "quadro de equipe". Cada equipe então pega algumas estórias do quadro do *product backlog*, começando pelas estórias com maior prioridade e coloca os cartões em seu próprio quadro de equipe.

Isso está ilustrado na imagem a seguir, com setas amarelas simbolizando o fluxo dos cartões do quadro de *product backlog* para os quadros de equipe.



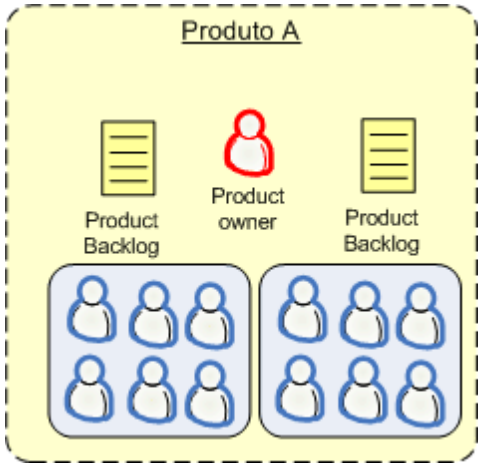
Conforme a reunião progride, o *product owner* e a equipe debatem sobre os cartões, movendo-os para cima e para baixo para alterar sua prioridade, quebrando-os em itens menores, etc. Após mais ou menos uma hora, cada equipe tem uma versão candidata para *sprint backlog* em seu quadro de equipe. Após isso as equipes trabalham independentemente, realizando estimativas de prazo e dividindo as tarefas.

É bagunçado, caótico e exaustivo, mas também efetivo, divertido e social. Quando o tempo acaba, normalmente todas as equipes têm informação suficiente para iniciar seu *sprint*.

Estratégia 2: Um product owner, múltiplos backlogs

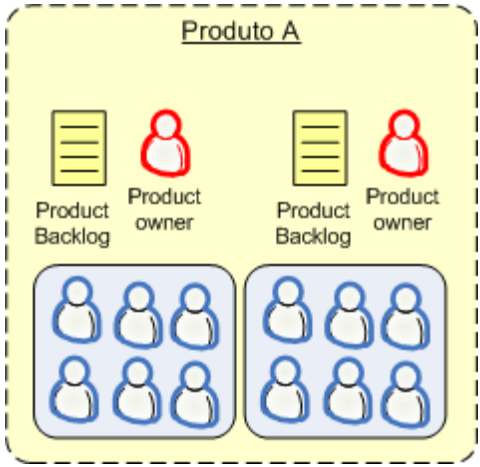
Nesta abordagem, o *product owner* mantém múltiplos *product backlogs*, um por equipe. Nós não tentamos esta abordagem, mas chegamos perto. Este é basicamente nosso plano B no caso da primeira abordagem falhar.

A fraqueza desta estratégia é que é o *product owner* quem aloca estórias para as equipes, tarefa esta que provavelmente seria melhor realizada pelas próprias equipes.



Estratégia 3: Múltiplos product owners, um backlog por dono.

Esta estratégia é semelhante a segunda estratégia, um *product backlog* por equipe, mas com um *product owner* por equipe também!



Nós não chegamos a executá-la, e provalmente nunca iremos.

Se dois *product backlogs* compartilham o mesmo código-fonte, isso irá provavelmente causar sérios conflitos de interesse entre os dois *product owners*.

Se dois *product backlogs* implicarem na construção de codigo-fonte separados, essencialmente seria o mesmo que separar o produto como um todo em sub-produtos e executá-los independentemente. O que significa que estaríamos de volta a situação uma-equipe-por-produto, a qual é a melhor e mais fácil.

Fazendo Branching de código

Com múltiplos times trabalhando no mesmo código-fonte, nós inevitavelmente tivemos que trabalhar com "*branches*" (ramificações) no sistema de gerenciamento de configuração de software (ou *SCM – software configuration management*). Existem inúmeros livros e artigos que explicam como lidar com várias pessoas trabalhando no mesmo código-fonte, então eu não vou entrar em detalhes aqui. Eu não tenho nada novo ou revolucionário a adicionar. Irei, entretanto, resumir algumas das mais importantes lições aprendidas até então por nossas equipes.

- Seja rigoroso ao manter a linha principal (*trunk*) num estado consistente. Isso significa, que em todos os casos, tudo deve estar compilando com todos os testes unitários passando. Deve ser possível gerar uma entrega a qualquer momento. Preferencialmente o sistema de compilação contínuo deve compilar e auto executar testes durante a noite.

- Gere uma *tag* (rótulo) para cada liberação ou entrega do código. Não importa se é uma entrega para testes de aceitação ou para produção, tenha certeza que existe uma *tag* no seu ramo principal para identificar exatamente cada entrega feita. Desta forma você poderá em qualquer momento no futuro voltar e criar *branches* de manutenção a partir daquele ponto específico.

- Crie novos *branches* somente quando necessário. Uma boa prática é fazer o *branch* de uma nova versão do código somente quando não é possível usar uma versão de código já existente sem quebrar sua política. Se estiver em dúvidas, não crie o *branch*. Por que? Porque cada *branch* ativo gera um custo de administração e complexidade.

- Use *branches* pricipalmente para separar ciclos de vida diferentes. Você pode optar ou não por ter o código de cada equipe Scrum no seu próprio ramo de código. Porém se você misturar pequenas correções com grandes alterações na mesma linha de código, você encontrará dificuldades para entregar as pequenas correções!

- Faça a sincronização freqüentemente. Se você estiver trabalhando numa *branch*, sincronize-se ao ramo principal (*trunk*) sempre que tiver algo que esteja compilando. Todos os dias quando chegar para trabalhar, faça a sincronia do ramo principal para o seu *branch*, desta forma você estará garantindo que seu *branch* esteja atualizado respeitando também as alterações das demais equipes. Se isso te levar a problemas com o *merge*, simplemente aceite o fato que seria muito pior esperar para fazer a atualização.

Retrospectivas com várias equipes

Como fazemos retrospectivas de *sprint* quando há múltiplas equipes trabalhando no mesmo produto?

Imediatamente após a apresentação do *sprint*, depois do aplauso e da mistura, cada equipe volta pra sua sala própria, ou para alguma localização confortável fora do escritório. Elas fazem retrospectivas mais ou menos como descrito na pág. 71 “Como fazemos retrospectivas de *sprint*”.

Durante a reunião de planejamento de *sprint* (à qual todas as equipes vão, já que fazemos *sprints* sincronizados a cada produto), a primeira coisa que fazemos é deixar um porta-voz de cada equipe resumir pontos chaves de sua retrospectiva. Leva em torno de 5 minutos por equipe. Então temos uma discussão aberta em torno de 10 a 20 minutos. Depois fazemos uma pausa. Finalmente, começamos o planejamento de *sprint* mesmo.

Nós nunca tentamos outra maneira para múltiplas equipes, isso funciona bem o suficiente. A maior desvantagem é que não há intervalo entre a retrospectiva e o planejamento do *sprint*. (Veja pág. 78 “Intervalo entre sprints”).

Para produtos de uma só equipe, nós não fazemos resumo retrospectivo durante a reunião de planejamento do *sprint*. Não há necessidade, já que todos estavam presentes na verdadeira reunião de retrospectiva.

16

Como lidamos com equipes distribuídas geograficamente

O que acontece quando as equipes estão em locais geográficos diferentes? A maior parte da "mágica" do *Scrum* e do XP é baseada em equipes localizadas juntas e membros de equipe fortemente colaborativos que programam em pares e encontram-se cara-a-cara todos os dias.

Nós temos algumas equipes geograficamente separadas e também alguns membros trabalhando em casa de vez em quando.

Nossa estratégia para essa situação é bem simples. Nós usamos todos os truques possíveis para aumentar a largura de banda de comunicação entre os membros separados fisicamente. Eu não falo simplesmente sobre a largura de banda em Mbits/segundo (apesar disso também ser importante). Eu me refiro à largura de banda de comunicação em um sentido mais amplo:

- A habilidade de programar juntos em pares.
- A habilidade de participar cara-a-cara na reunião diária.
- A habilidade de haver discussões cara-a-cara a qualquer momento.
- A habilidade de encontrar-se fisicamente para uma social.
- A habilidade de ter reuniões espontâneas com a equipe inteira.
- A habilidade de ter a mesma visão do *sprint backlog*, *sprint burndown*, *product backlog* e outros termômetros de informação.

Algumas métricas que nós implementamos (ou estamos implementando. Nós não fizemos tudo ainda) são:

- *Webcam* e fone de ouvido em cada estação de trabalho.
- Salas de conferência habilitadas para encontros remotos com *webcams*, microfones p/ conferência, computadores sempre prontos, *software* de compartilhamento de *desktop*, etc.
- "Janelas remotas". Telas grandes em cada local, mostrando o que está acontecendo nos outros locais. É uma espécie de janela

virtual entre dois departamentos. Você pode ficar lá e navegar. Você pode ver quem está nas mesas e quem está conversando com quem. Isso é para criar um sentimento de "ei, nós estamos juntos nisso".

- Programas de intercâmbio, quando pessoas de um lugar viajam e visitam outras regularmente.

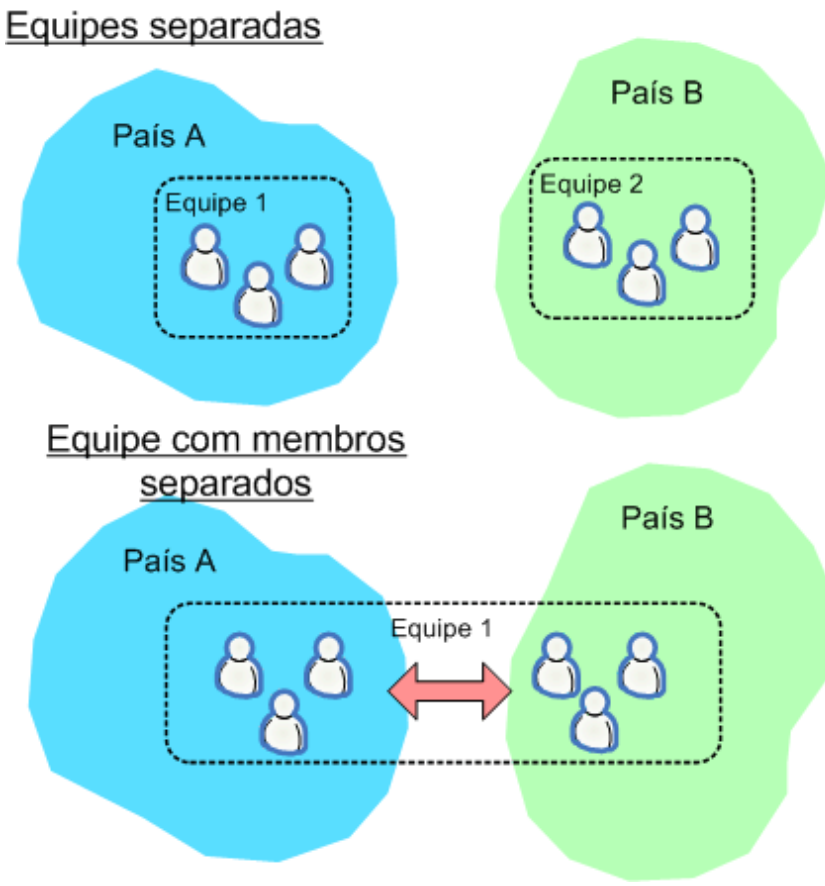
Usando essas técnicas e mais algumas, nós estamos lentamente mas certamente começando a pegar o jeito de como fazer reuniões de planejamento, demonstrações, retrospectivas, reuniões diarias, etc. com membros da equipe distribuídos geograficamente.

Como é normal, isso tudo é um teste. Testar => adaptar => testar => adaptar => testar => adaptar => testar => adaptar => testar => adaptar

Equipes em outro país

Nós temos várias equipes em outros países e temos experimentado como lidar eficientemente com elas usando *Scrum*.

Existem duas estratégias principais aqui: equipes separadas ou membros separados das equipes.



A primeira estratégia, equipes separadas, é uma escolha instigante. Entretanto, nós começamos com a segunda estratégia, membros separados das equipes. Existem várias razões para isso.

1. Nós queremos que os membros das equipes conheçam-se bem.
2. Nós queremos uma excelente infra-estrutura de comunicação entre os dois locais, e também queremos dar um forte incentivo às equipes para mantê-las.
3. No princípio, a equipe em outro país é muito pequena para formar uma equipe autônoma.
4. Nós queremos um período de intenso compartilhamento de conhecimento antes que as equipes em outro país independentes sejam uma alternativa viável.

Ao longo do tempo, nós poderemos andar na direção da estratégia de "equipes separadas"

Membros da equipe trabalhando em casa.

Trabalhar em casa pode ser muito bom algumas vezes. As vezes você consegue programar mais em um dia em casa do que conseguiria a semana toda na empresa. Isso se não tiver filhos :o)

Mas um dos fundamentos do Scrum diz que a equipe toda deve estar fisicamente no mesmo lugar. Então o que fazemos?

Basicamente deixamos a equipe decidir quando e com que frequencia é adequado trabalhar de casa. Alguns membros da equipe trabalham regularmente de casa devido a viagens. Entretanto, nós encorajamos as equipes a estarem no mesmo local a maior parte do tempo.

Quando os membros da equipe trabalham de casa eles participam das *reuniões* diárias através da chamada de voz do *skype* (as vezes por vídeo). Eles ficam disponíveis em mensageiros on-line o dia todo. Não é tão bom quanto estarem na mesma sala, mas o suficiente.

Nós tentamos uma vez o conceito de ter as quartas-feiras como o *dia do foco*. Isto significava que basicamente "se você quisesse trabalhar em casa, tudo bem, mas faça isso nas quartas-feiras. E cheque antes com sua equipe". Isto funcionou muito bem com a equipe na qual tentamos. Geralmente toda a equipe ficava em casa na quarta-feira e conseguiam render bem, sem deixar de colaborarem uns com os outros satisfatoriamente. O fato de que era apenas um dia na semana não deixa que a equipe ficasse muito fora-de-sincronia. Mas por algum motivo a idéia não pegou com as outras equipes.

Em geral, pessoal trabalhando em casa não foi realmente um problema para nós.

17

Checklist do scrum master

No final deste capítulo eu irei mostrar a vocês o "*checklist*" do nosso *scrum master*, listando suas rotinas administrativas mais comuns. Coisa fácil de se esquecer. Nós cortamos as coisas óbvias como "remova os impedimentos da equipe".

Começando o sprint

- Depois da reunião de planejamento do *sprint*, Crie uma página de informações.
  - Adicione um link para sua página do *dashboard* no *wiki*.
  - Imprima a página e coloque num muro onde as pessoas passam.
- Mande um email para todo mundo avisando que um novo *sprint* começou. Inclua o objetivo do *sprint* e um link para a página de informações do *sprint*.
- Atualize o documento de estatísticas do *sprint*. Adicione a velocidade real, o tamanho da equipe, o comprimento do *sprint*, etc.

Todo dia

- Garanta que a reunião diária comece e termine pontualmente.
- Garanta que estórias sejam adicionadas e removidas do *sprint backlog* conforme a necessidade de manter o *sprint* dentro do planejamento.

    - Garanta que o *product owner* seja notificado dessas mudanças.
- Garanta que o *sprint backlog* e *burndown* sejam mantidos atualizados pela equipe.
- Garanta que problemas e impedimentos foram resolvidos ou reportados ao *product owner* e/ou chefe de desenvolvimento.

Fim do Sprint

- Faça uma demonstração aberta do *sprint*.
- Todos devem ser avisados sobre a demonstração dois dias antes.
- Faça uma retrospectiva do *sprint* com a equipe inteira e o *product owner*. Convide o chefe de desenvolviemnto também, daí ele pode ajudar a espalhar os conhecimentos que aprendeu.
- Atualize o documento de estatísticas do *sprint*. Adicione a velocidade real e os pontos chave da retrospectiva.

18

Últimas palavras

Ufa! Nunca pensei que seria tão demorado.

Espero que este livro tenha lhe fornecido algumas idéias úteis, seja você um iniciante em *Scrum* ou um veterano.

Como o *Scrum* deve ser adaptado às necessidades específicas de cada ambiente, é difícil discutir de forma construtiva sobre as melhores práticas de forma genérica. Apesar disso estou interessado em receber seu *feedback*. Diga-me como a sua abordagem difere da minha. Envie suas idéias sobre como podemos melhorar!

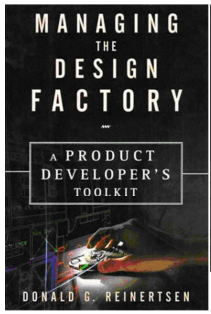
Sinta-se a vontade para me contatar em henrik.kniberg@crisp.se
Ou então em **scrumdevelopment@yahoogroups.com**.

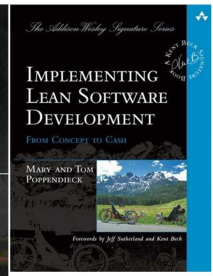
Se você gostou deste livro pode dar uma olhada também em meu blog. Espero ir adicionando alguns posts sobre Java e desenvolvimento Ágil.
http://blog.crisp.se/henrikkniberg/

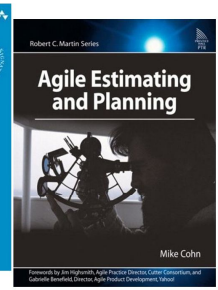
Ah, e não esqueça…
Isto é só um trabalho, certo?

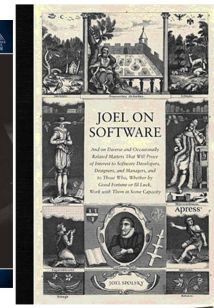
Leituras recomendadas

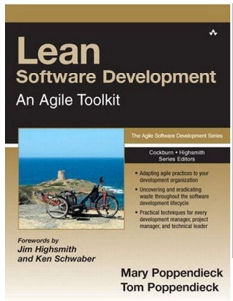
Aqui estão alguns livros que me deram muita inspiração e idéias. São altamente recomendados!

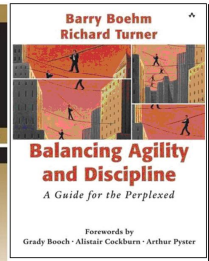


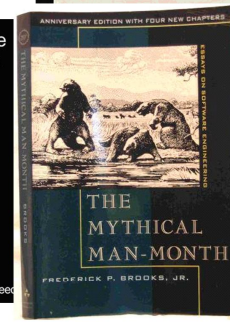


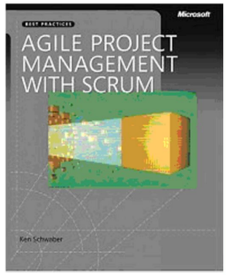


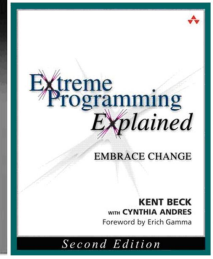


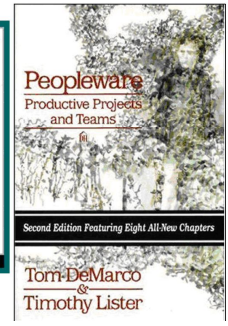












Sobre o autor

Henrik Kniberg (henrik.kniberg@crisp.se) é consultor na Crisp em Estocolmo (www.crisp.se), especializando em Java e desenvolvimento ágil de software.

Desde que apareceram os primeiros livros de XP e o manifesto ágil, Henrik abraçou os princípios ágeis e tentou aprender como aplica-los eficientemente em diferentes tipos de organizações. Como co-fundador a CTO (*chief technology officer*) de Goyada 1998-2003, ele teve ampla oportunidade de experimentar o TDD (*test-driven development*) e outras práticas ágeis bem como montar e gerenciar uma plataforma técnica e uma equipe de desenvolvimento com 30 pessoas.

No final de 2005 Henrik foi contratado como chefe de desenvolvimento em uma companhia Sueca da industria de jogos eletrônicos. A companhia estava em crise com problemas técnicos e organizacionais urgentes. Usando *Scrum* e XP como uma ferramenta, Henrik ajudou a companhia sair da crise através da implementação de princípios ágeis e leves em todos os níveis da empresa.

Em uma Sexta-feira de Novembro de 2006, Henrik estava em casa na cama, com febre e decidiu tomar algumas notas para ele mesmo do que ele aprendeu através do último ano. Uma vez que começou a escrever, não conseguiu parar após três dias frenéticos; escrevendo e desenhando as notas iniciais se tornaram um artigo de 80 páginas intitulado “Scrum e XP direto das Trincheiras”, que veio se tornar este livro.

Henrik usa uma abordagem holística e divertida na adoção de diferentes perfis como gerente, desenvolvedor, *Scrum master*, professor e *coach*. Ele é apaixonado em ajudar as companhias a construir excelentes *softwares*, excelentes equipes, participando em qualquer perfil que seja necessário para isto.

Henrik cresceu em Tokyo e agora vive em Estocolmo com sua esposa Sophia e dois filhos. Ele é um músico ativo em seu tempo livre, compondo e tocando baixo e teclado com sua banda local.

Para mais informações, acesse: http://www.crisp.se/henrik.kniberg

Sobre a Tradução

Este livro foi traduzido voluntariamente por brasileiros, nas mais diversas funções, com os mais diversos conhecimentos das línguas cujo único interesse foi o de disseminar e popularizar as metodologias ágeis

Coordenação

Bruno Pedroso	brunopedroso@gmail.com
Renato Willi	renato.willi@gmail.com

Revisão

Versão 1.0	
Renato Willi	renato.willi@gmail.com
Versão 1.1	
Marcos Vinícius Guimarães	
Ian Gallina	
Versão 1.2	
Leonardo Siqueira Rodrigues	
Versão 1.3	
André Delorme	

Ilustração

Eduardo Bobsin Machado	eduardo.bobsin@gmail.com

Tradução

Vinícius Assef	viniciusban@gmail.com
Cássio Marques	cassiommc@gmail.com
Álvaro Maia	
José Inácio Serafini	serafini@sr.cefetes.br
Rafael Benevides	rafabene@gmail.com
Rafael Recalde Caceres	rafael.r.caceres@gmail.com
Fernanda Stringassi de Oliveira	fernandasoliveira@yahoo.com
Renato Willi	renato.willi@gmail.com
Rodrigo Palhano	
Daniel Wildt	dwildt@gmail.com
Francisco M. Soares Nt.	xfrancisco.soares@gmail.com
Eduardo Bobsin Machado	eduardo.bobsin@gmail.com
Juliana Berossa Steffen	julianaberossa@gmail.com
Mauricio Vieira	mauricio@mauriciovieira.net
Rodrigo Russo	
Adam Victor Brandizzi	brandizzi@gmail.com
Alexandre Gomes	alegomes@gmail.com

Bruno Pedroso	brunopedroso@gmail.com
Marcos Machado	
Victor Hugo Germano	victorhg@gmail.com
Acyr José Tedeschi	
Emanoel Tadeu da Silva Freitas	emanoeltadeu@gmail.com
Gustavo Grillo	gustavogrillo@gmail.com
Israel Rodrigo Faria	israel.faria@gmail.com
Marcos Vinícius Guimarães	
Pablo Nunes Alves	pablotj@gmail.com
Taiza Sousa	

Apoio

SEA Tecnologia www.seatecnologia.com.br