3 Boas práticas para o emprego de Scrum

Dean Leffingwell no seu livro *Scaling Software Agility*, descreve sete práticas para escalar o uso de Scrum.

3.1. Definir. Construir. Testar

Para construir código funcionado em um curto período de tempo, as equipes precisam se organizar para conter três capacidades, fundamentais para entregar software.

• Capacidade para agir nas definições e como ponte entre o cliente e o próprio time, tarefa ao qual o *Product Owner* é incumbido.

• Criar código com o mínimo de distração e troca de contexto entre projetos. E para esse caso desenvolvedores capacitados e protegidos por um *Scrum Master*.

• Testes, de forma integral, automatizados além de capacidade em testes manuais.

Todo método ágil divide grandes conjuntos de trabalho em pequenos pedaços, representados como histórias, casos de uso, itens de *Backlog*, enfim, uma lista de "coisas" a serem feitas. O objetivo do time é definir, construir e testar cada uma dessas "coisas" em um tempo pré-estabelecido (*time box*).

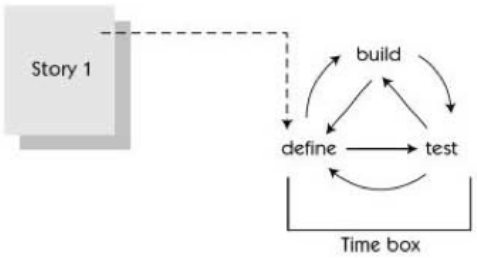

Figura 6 – Definir. Construir. Testar

[Leffingwell, 2007]

3.1.1. Definir

Não importa o quão bem elaborada está uma história, os desenvolvedores irão interagir com o *Product Owner* para entender o que ela quer dizer. Durante esse processo algum design já estará presente na imaginação de cada desenvolvedor, e não estando certamente será criada.

Nesse processo não somente o design de código começa a surgir, os requisitos também podem ser modificados para facilitar o desenvolvimento no período de tempo disponível. É possível inclusive acrescentar novas idéias, enriquecendo o requisito original.

Figura 7 – Retro alimentação de requisitos e design

[Leffingwell, 2007]

3.1.2. Construir

A construção propriamente dita. Mesmo durante o desenvolvimento a equipe se mantém em contato com o *Product Owner* para validar eventuais questões de definição, havendo inclusive a possibilidade de redefinir a própria história ou parte dela.

3.1.3. Testar

A história não estará completa até que esteja testada e aceita. Em geral criam-se testes automatizados para garantir que eventuais desenvolvimentos no entorno da história em questão não gerem quebra. O próprio teste servirá também como documentação técnica do funcionamento de cada componente gerado pela a história em questão.

Se considerarmos a prática de TDD (*Test Driven Development*) Construir e Testar não serão necessariamente sequenciais, uma vez que, para cada parte do código, os testes são criados antes do código em si.

Para que exista esse fluxo de forma constante, é necessário que o time seja composto por membros capazes de desenvolver integralmente a funcionalidade ou história em questão.

"Um time deve ser multifuncional, contendo todo o conhecimento necessário para atingir o objetivo daquela iteração" [Schwaber / Beedle, 2001].

Uma das características comuns às empresas e que dificulta a adoção de métodos ágeis, é a separação de funções em silos que são:

- Otimizados para comunicação vertical através de gerência.
- Gera facilmente fricção entre silos interdependentes
- Seus profissionais são alocados e agrupados por função
- Naturalmente gerando barreiras políticas entre funções

Com métodos ágeis a empresa precisa se reorganizar para que cada time tenha as habilidades para definir, desenvolver, testar e entregar cada história. Nesse caso é fundamental quebrar esses silos e gerar um time, comprometido com a entrega, contendo representantes de cada uma das partes envolvidas [Leffingwell, 2007].

É possível ainda organizar cada time por responsabilidade, componentes, funcionalidades ou serviço.

3.2. Dois níveis de planejamento

Por mais que o desenvolvimento ágil se afaste de planejamento detalhado muito a frente, não descarta a possibilidade de criar um plano a longo prazo, pensando em entregas em pequenas iterações e de uma forma flexível, isso é, o plano deve ser contínuo sendo revisado a cada iteração para, caso necessário ajustá-lo.

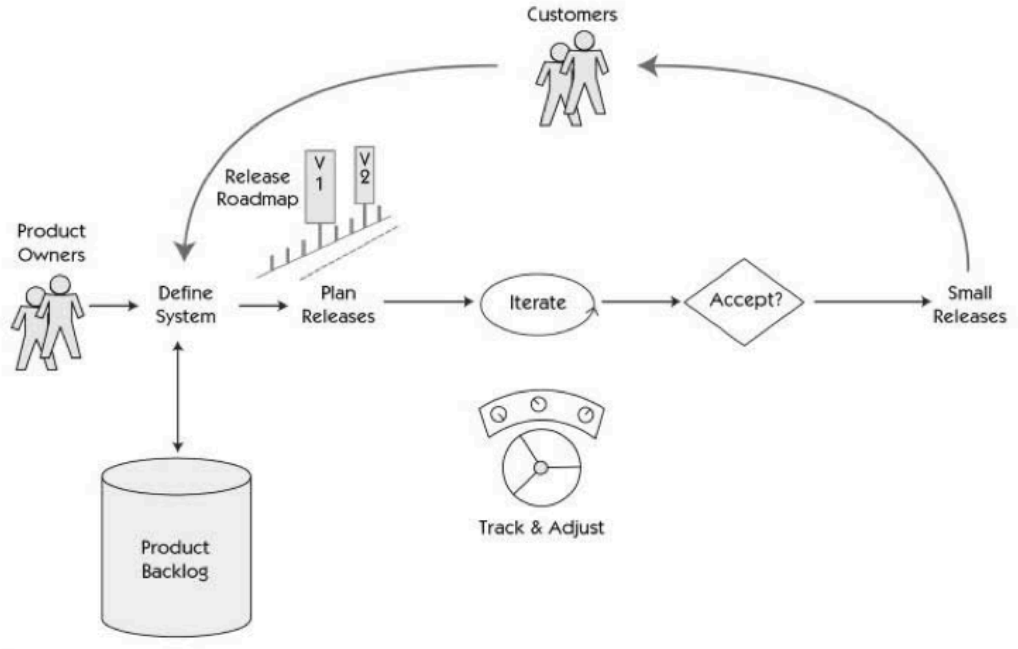

Figura 8 – Planejamento contínuo

[Leffingwell, 2007]

Para garantir esse fluxo de desenvolvimento precisamos dividir o planejamento em duas partes, divididas em Release e Iteração.

A primeira parte é no nível de release, em que se destacam as macro-funcionalidades ou temas, observadas a partir do *Product Backlog*. Ela abrange um grupo de iterações, tendo como entrega macro-funcionalidades.

A figura abaixo representa uma parte do acompanhamento da evolução do *roadmap* trimestral de um dos times de desenvolvimento da globo.com.

Nele podemos observar temas e macro funcionalidades (*features*) e a sua evolução. Por exemplo:

Um dos temas que podemos observar é a "Página de Tópico", para que esse tema seja desenvolvido serão necessárias quatro macro funcionalidades, "Página Básica", "Template Padrão", "Template Pessoa" e "Otimização de montagem de página".

Cada uma das macro funcionalidades pode representar uma ou diversas histórias no contexto do time.

No relatório a evolução é ilustrada por uma barra verde dentro da caixa da macro funcionalidade. A observação dessa evolução dada eventuais dificuldades de desenvolvimento podem levar com que outra macro funcionalidade perca a prioridade visto a necessidade de ajuste do *roadmap*.

Figura 9 – Acompanhamento do roadmap trimestral

A segunda fase é no nível de iteração, que deverá ser quebrado em um período menor. Nesse nível o planejamento é mais detalhado chegando ao nível das tarefas.

A iteração é guiada por um objetivo, logo, as histórias selecionadas para essa iteração deverão somadas serem capazes de atingir a esse objetivo.

Cada iteração tem inicio meio e fim. Organizados inicialmente por uma fase de planejamento, histórias contendo definição, construção e teste. Uma revisão das histórias agrupadas espelhadas no alcance ou não do objetivo e finalmente uma retrospectiva para eventuais ajustes para a próxima iteração.

Ao fim de cada iteração o primeiro planejamento (de releases) deverá ser revisitado e eventualmente reajustado.

3.3. Dominando a iteração

Uma das diferenças cruciais entre as metodologias tradicionais e ágeis é a habilidade de construir código funcional, testado em um pequeno espaço de tempo. Nesse caso cada ajuste ou nova funcionalidade torna-se potencialmente uma entrega.

Mas para que isso aconteça, é necessário que o time tenha a capacidade de dominar a iteração, dividindo suas histórias ou funcionalidades em partes que são capazes de serem produzidas no tamanho de sua iteração.

A primeira questão que os times enfrentam no que diz respeito à iteração é: Qual o tamanho ideal de uma iteração? A maioria dos livros concorda que independente do seu tamanho, iterações devem manter o mesmo formato durante um projeto, por dar melhor visibilidade de progresso ao time e facilitar a manutenção do *Release Plan*. Uma iteração muito curta pode forçar o time a paralelizar histórias, enquanto uma iteração muito longa pode não dar margem para ajustar o planejamento de forma rápida.

Em geral, duas semanas de iteração força quebrar histórias em pequenas partes, possibilita tempo suficiente para gerar código funcionando, oportunidade de ter sucesso ou falha cedo e replanejar mais rápido.

Independente de seu tamanho, toda iteração deverá ter o mesmo padrão, isso é parte da disciplina do desenvolvimento ágil.

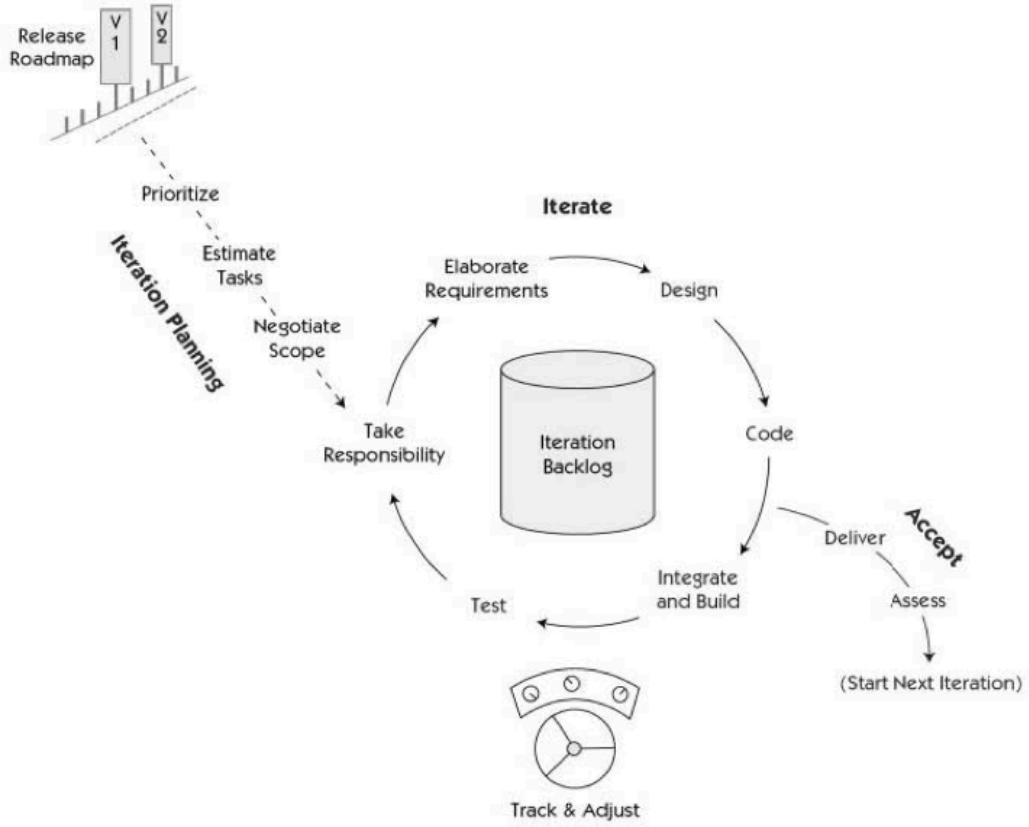

Figura 10 – Ciclo de Roadmap e Iterações

[Leffingwell, 2007]

3.4. Entregas pequenas e frequentes

Outro princípio requerido nos métodos ágeis é a participação constante dos clientes ou usuários. A melhor forma de obter essa interação é entregando software funcionando constantemente. Dessa maneira podemos observar se o caminho que o time está percorrendo precisa ou não de ajustes. Quanto mais rápido forem feitas as entregas, mais rápido absorve-se a resposta do cliente. O risco desenvolver algo não alinhado com a expectativa do cliente é

substancialmente reduzido porque ele passa a ter a habilidade de prover feedback rápido a respeito da evolução do desenvolvimento, se esse está ou não endereçado a resolver suas necessidades.

Além do cliente prover feedback rápido daquela entrega, ele também poderá trazer novos problemas, eventualmente prioritários àqueles que estão previstos nas próximas entregas.

Fazer entregas de forma frequente é uma das bases do desenvolvimento ágil, da mesma forma que as iterações, releases também precisam ser planejadas, acompanhadas e entregues aos usuários finais. Contudo esse planejamento deve ser feito de uma forma mais abstrata, considerando no máximo duas ou três releases com uma quantidade limitada de funcionalidades, exatamente para mantê-la curta e de fácil mudança.

Uma vez sendo considerada uma release baseada em data, dois pontos não deverão ser modificados: data e qualidade. Resta apenas o escopo a ser modificado dentro de uma release: redução da quantidade de funcionalidades ou simplificação de algumas delas. Considerando que os ajustes necessários durante cada interação estejam alinhados a solucionar as necessidades do cliente.

Com podemos notar, o Scrum é direcionado a ciclos datados. No modelo em cascata e modelos baseados em planos, requisitos foram fixados e times fizeram o possível para estimar os recursos necessários e a data para a qual o desenvolvimento de uma funcionalidade pré-determinada fosse atingida com esses recursos. Mas essas datas estimadas não são muito precisas e isso causa grande dificuldade para nós e organizações, que dependem desses compromissos. Na verdade essa incapacidade em prever datas realizáveis é um dos principais motivadores do desenvolvimento ágil. Reconhecemos que, para os fins mais práticos, recursos são fixos e é difícil adicionar recursos a um projeto em andamento sem reduzir sua velocidade [Brooks, 1975]. Em um modelo aonde recursos, data e qualidade são fixas, a lista de funcionalidades deverá ser variável.

O *Release Plan* é uma representação de como e quais *features* deverão ser implementadas, culminando em um *Product Release*. O plano é dividido em iterações e associado a funcionalidades de alto nível, ou histórias. Esse plano

deverá ser criado durante uma reunião a parte das tradicionais do fluxo do Scrum mas deverá ser direcionada a ser executada dentro dos fluxos das iterações.

Esse plano também inclui informações sobre a arquitetura e qualquer dependência ou outras questões que eventualmente poderiam afetar o caminho das iterações.

O resultado de um *release plan* deverá conter:

• *Goal* ou tema

• Lista priorizada de funcionalidades ou histórias

• Uma lista de iterações necessárias para entregar o release em uma visão macro

• Dependências, suposições e preocupações, ouvidas pelos membros do time que precisarão ser endereçadas

• Compromisso de todo o time

• O plano deverá estar exposto de maneira visível em uma área comum ao time.

Quando estamos falando de um projeto grande, com múltiplos times, o *Release Planning* torna-se ainda mais critico, pois existe a necessidade de alinhar todos os times em um objetivo comum. Tendo sido feito esse alinhamento, todos os times passam a trabalhar por um objetivo, suas interdependências podem ser datadas e acompanhadas a partir do progresso do *Release Plan*. Nesse caso puderam ser observados não só o progresso como a necessidade de repriorização ou simplificação de uma necessidade.

3.5. Testes simultâneos

Em se tratando de métodos ágeis, todo código deve ser um código testado. Seja unitariamente ou em forma de aceitação, performance, testes necessários a cada trecho de código devem ser produzidos e automatizados dentro de cada iteração. Em alguns modelos ágeis como XP, os times são encorajados a produzir

os testes antes mesmo do código, além de facilitar o teste, isso ajuda também a entender melhor os requisitos daquela funcionalidade.

Princípios de testes em um ambiente ágil

• Todo código é um código testado.
• Testes são escritos antes em conjunto com o próprio código.
• Teste é um esforço do time. Ambos testadores e desenvolvedores deverão escrever testes.
• Automação é uma regra, não uma exceção.

Existem diversas estratégias de testes, a maioria dos times no ambiente ágil adotam basicamente os testes a seguir:

3.5.1. Teste Unitário

Ou teste de unidade, em que se entende por unidade a menor parte testável de um sistema, é o método pelo qual desenvolvedores testam o código como um módulo único, independente. Teste unitário é um teste de baixo nível, aonde o teste deverá ter acesso ao mais interno do objeto, método ou interface que está sendo testada. É um teste de isolamento, em que não deve ser levada em consideração a sua relação com outros objetos.

Geralmente testa-se em um teste unitário diversos tipos de entrada e saída de um único método, para que possa ser verificado se a resposta do método em questão está em acordo com o esperado, e no caso de entradas inválidas que seja feito um tratamento adequado.

3.5.2. Teste de Aceitação

Refere-se à reprodução do teste feito por um membro do time, uma área responsável por testes de qualidade, *Product Owner* ou o próprio cliente. Deve validar se o novo código está em acordo com os requisitos. Diferente do teste unitário, os testes de aceitação deverão tocar os componentes ao seu entorno, deverá servir como um teste "caixa-preta" em que o que está sendo testado é o comportamento do novo código na sua relação com os demais componentes.

Como o teste unitário deverá ser adicionado a cada nova funcionalidade ou nova ação de uma funcionalidade existente.

3.5.3. Teste de Performance

Testes unitários e de aceitação fazem referencia ao comportamento de cada componente, classe ou método, mas não testam o quanto efetivo esse novo trecho de código é no que diz respeito a performance.

O teste de desempenho mostra a escalabilidade e precisão do novo componente e seu impacto na adição ao sistema.

Poderá nos dar uma visão desde consumo de CPU, memória, disco, uso de sistemas associados, como também tempo de carregamento de uma página e seu tamanho, em se tratando de desenvolvimento web.

3.6. Integração contínua

Essa não é uma pratica nova, mas torna-se mais desafiadora quando falamos em times ou projetos grandes, uma vez que a quantidade de código e mudanças a serem integradas e testadas torna-se maior.

Martin Fowler define integração contínua como:

"Integração Contínua é uma pratica de desenvolvimento de software em que os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um build automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente". [Fowler, M. 2006]

Integração contínua então, é um processo suportado por ferramentas, que resulta em *builds* frequentes do sistema, considerando cada novo código adicionado. Com a integração contínua a inspeção do sistema por compiladores e testes automatizados é frequente. A resposta à adição do novo código é imediata.

Defeitos podem ser identificados e corrigidos pouco tempo depois de terem sido injetados no código ou mesmo no que diz respeito a desconformidade de especificação, havendo teste para ela.

Alguns benefícios da integração contínua podem ser vistos abaixo:

• Menos tempo gasto examinando bugs causados pelo código de uma pessoa causar reflexo no código de outra.

• Descoberta rápida de falha de entendimento entre desenvolvedores trabalhando em um mesmo componente ou em componentes relacionados

• Defeitos são descobertos enquanto o trabalho está fresco na cabeça dos envolvidos

• Todo o código no repositório (GIT, CVS...) está testado e existe uma execução de um estado conhecido

• Todo novo código está compilado e testado

• Segurança em relação a mudanças

• O progresso do sistema como um todo pode ser medido

3.7. Reflexão e adaptação constante

Um dos princípios do Manifesto Ágil é "Em intervalos regulares, o time observa reflete sobre como tornar-se mais efetivo, podendo ajustar e otimizar seu comportamento de acordo com as observações."

Essa é a melhor maneira de dar poder ao time, observar e corrigir tudo o que da mínima forma atrapalha ou impede a melhoria do time. Fazer uso constante dessa prática faz com que a cada iteração o time possa produzir mais e melhor.

A reunião de retrospectiva, executada ao final de cada iteração, provê uma das primeiras oportunidades para que o time possa fazer essa reflexão e implementar melhorias em seu processo, seu produto e a relação desses com o objetivo final.

O ciclo rápido e frequente de reflexão e adaptação faz que não só problemas sejam corrigidos como facilita a experimentação de formas diferentes de corrigir um determinado problema. Ao fim dessa iteração analisa-se inclusive se a forma que determinado problema foi atacado foi a melhor e se está sendo eficaz.

No que diz respeito a essa reflexão, não devemos nos limitar apenas a iteração em si, mas também ao ser atingido o final de um ciclo maior, como o fechamento de um *Release Plan*, ou de uma meta anual, ou até da entrega de um sistema como um todo.

É interessante analisar as métricas da iteração e como o time pode fazer para melhorá-las. Alguns exemplos de métricas que podem ser analisadas estão a seguir:

• Número de histórias adicionadas no início de iteração e completadas a seu fim.

• Número de histórias aceitas

• Número de histórias endereçadas para o próximo *Sprint*

• Histórias adicionadas durante a iteração (deveria sempre ser zero)

• Defeitos conhecidos no início da iteração

• Defeitos conhecidos ao fim da iteração (e sua relação com o item anterior)

• Novos casos de teste (preferencialmente automatizados)

• Percentual de cobertura de testes

Outro ponto primordial a ser observado é a qualidade das histórias escritas pelo *Product Owner* e a relação do entendimento delas com o Time. Se estão escritas de forma a dar pleno entendimento do problema que está sendo atacado e o seu objetivo. Em alguns times o próprio *Product Owner* é responsável por escrever o teste de aceitação antes do desenvolvimento para guiar o time no progresso da história em questão.

Além de levantar quais foram os problemas e de que forma eles podem ser endereçados, é muito importante destacar os sucessos da iteração, não somente para que o time tenha a percepção de suas vitórias mas para que continuem fazendo bem (e cada vez melhor) aquilo que já é positivo.