Capítulo 4

Modelos Ágeis

Este capítulo apresenta modelos que têm menos ênfase nas definições de atividades e processos e mais ênfase na pragmática e nos fatores humanos e ambientais do desenvolvimento. Os *métodos ágeis*, antigamente conhecidos também como *processos leves*, estão em franco desenvolvimento e várias abordagens podem ser encontradas na literatura e na indústria. Neste capítulo são apresentados alguns métodos representativos, iniciando com *Scrum* (Seção 4.1), que é de longe o modelo ágil mais utilizado hoje no mundo. O capítulo segue com a apresentação de *Lean* (Seção 4.2) e Kanban (Seção 4.3), modelos ágeis surgidos na indústria automobilística e adaptados para o desenvolvimento de software. Em seguida é apresentado XP ou *eXtreme Programming* (Seção 4.4) que, embora seja pouco usado em sua forma pura, é bastante popular como um complemento para equipes que utilizam Scrum, já que suas práticas usualmente são compatíveis e complementares. O capítulo encerra com a apresentação de FDD – *Feature Driven Development* (Seção 4.5) e *Crystal* Clear (Seção 4.6) que embora sejam importantes do ponto de vista histórico e filosófico, são relativamente menos utilizados na prática.

Os modelos ágeis de desenvolvimento de software seguem uma filosofia diferente da filosofia dos modelos prescritivos. Em vez de apresentar uma "receita de bolo", com fases ou tarefas a serem executadas, eles focam valores humanos, ambientais e sociais.

Apesar de os métodos ágeis serem usualmente mais leves, ou seja, menos burocráticos, é errado entendê-los como modelos de processo simplistas. Não se trata apenas de simplicidade, mas de focar mais nos resultados do que no processo. Pode-se pensar, como metáfora, em como organizar uma padaria: a *padaria prescritiva* tem receitas detalhadas para fazer diversos tipos de pães e os padeiros devem segui-las. Não precisam ser ótimos padeiros, desde que sigam a receita. Quando um pão não dá certo, tenta-se descobrir o porquê, e a forma de evitar o fracasso passa a ser incorporada no processo como uma nova atividade ou regra. Já a *padaria ágil* não tem receitas predefinidas, mas tem excelentes padeiros, equipamentos e ingredientes, e estes padeiros seguem regras de trabalho que foram aprendidas com a experiência. Com esta estrutura espera-se que os padeiros sejam capazes de produzir os melhores pães.

Os princípios dos modelos ágeis foram claramente colocados no *Manifesto ágil* (acessível pelo *QR code* ao lado) [QRC 4.1] e assinados por 17 pesquisadores da área, entre os quais Kent Beck, Jim Highsmith, Ken Schwaber, Martin Fowler, Alistair Cockburn e Robert Martin. O manifesto estabelece o seguinte:

- Indivíduos e interações estão acima de processos e ferramentas.
- Software funcionando está acima de documentação abrangente.
- Colaboração do cliente está acima de negociação de contrato.
- Responder à mudança está acima de seguir um plano.

Isso não significa que os modelos ágeis não valorizem processos, ferramentas, documentação, contratos e planos. Quer dizer apenas que esses elementos terão *mais sentido* e *mais valor* depois que indivíduos, interações, software funcionando, colaboração do cliente e resposta às mudanças *também* forem considerados importantes. É importante grifar estes pontos porque ainda há pessoas que acreditam que ser ágil é abandonar planejamento, modelagem, documentação e

ferramentas. Não é nada disso: ser ágil é valorizar as coisas que realmente importam para que todo o resto faça mais sentido.

Processos bem estruturados de nada adiantam se as pessoas não os seguem; software bem documentado também não adianta se não satisfaz os requisitos ou não funciona, e assim por diante. Mas se as pessoas seguem os processos definidos, eles podem ajudar muito; se o software satisfaz os requisitos, sua documentação também é importante, e assim por diante.

O manifesto ágil é complementado pelos seguintes doze princípios:

- Nossa maior prioridade é satisfazer o cliente através da entrega rápida e contínua de software com valor.
- Mudanças nos requisitos são bem-vindas, mesmo nas etapas finais do projeto. Processos ágeis usam a mudança como um diferencial competitivo para o cliente.
- Entregar software frequentemente, com intervalos que variam de duas semanas a dois meses, preferindo o intervalo mais curto.
- O pessoal de negócios (*business people*) e desenvolvedores devem trabalhar juntos diariamente durante o desenvolvimento do projeto.
- Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte e confie que eles farão o trabalho.
- O meio mais eficiente e efetivo de tratar a comunicação entre/para a equipe de desenvolvimento é a conversa "cara a cara".
- Software funcionando é a medida primordial de progresso.
- Modelos ágeis promovem desenvolvimento sustentado. Os financiadores, usuários e desenvolvedores devem ser capazes de manter o ritmo indefinidamente.
- Atenção contínua à excelência técnica e bom *design* melhoram a agilidade.
- Simplicidade – a arte de maximizar a quantidade de trabalho não feito – é essencial.
- As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizadas.
- Em intervalos regulares, a equipe reflete sobre como se tornar mais efetiva e então ajusta seu comportamento de acordo com essa meta.

Um conjunto significativo de modelos atuais é considerado ágil. Alguns são até muito diferentes entre si, mas praticamente todos consideram os princípios mencionados como pontos fundamentais em seu funcionamento.

Quando se trata de popularidade, porém, há um modelo dominante no mundo: *Scrum*. Um estudo realizado continuamente denominado *Annual State of Agile Survey* (ver *QR code*) [QRC 4.2] indicou em 2018 quais são os modelos ágeis mais usados pelos respondentes da pesquisa. Dentre estes, *Scrum* aparece com 56%, a combinação entre *Scrum* e XP com mais 6%, combinações híbridas de vários modelos aparecem com 14%, *Scrumban*, uma combinação de *Scrum* com *Kanban*, com 8%, *Kanban* puro com 5% e outras, incluindo *Iterative Development*, *Spotify*, *Lean Startup* e *XP* puro com valores entre 1 e 3%. Isso coloca *Scrum* e suas combinações como líder absoluto entre os modelos ágeis, com cerca de 70% de adoção somando a forma pura e combinada, de acordo com a pesquisa. XP e *Kanban* aparecem também como importantes modelos em suas versões puras, mas especialmente quando combinados com *Scrum*. Assim, nas próximas seções apresentaremos com detalhe esses modelos bem como discorreremos sobre suas possíveis combinações.

4.1 Scrum

Scrum é um modelo ágil para a gestão de projetos que procura fazer com que as equipes de desenvolvimento se tornem hiperprodutivas. A concepção inicial de *Scrum* deu-se a partir do artigo de Takeuchi e Nonaka (1986), que apresentava como exemplo o modelo de produção de automóveis da Honda. O modelo *Scrum*, porém, pode ser adaptado a várias outras áreas, incluindo produção de software. No artigo original havia vários conceitos compatíveis com a filosofia *Lean* (Seção 4.2), mas os autores nunca assumiram explicitamente alguma influência direta entre estes trabalhos.

Na área de desenvolvimento de software, o *Scrum* deve sua popularidade inicialmente ao trabalho de Schwaber. Uma boa referência para quem deseja adotar o método é o livro de Schwaber e Beedle (2001), que apresenta o método de forma completa e sistemática. Existe também um documento mantido atualizado na Internet (ver *QR code*) que é o Guia do Scrum (Schwaber & Sutherland, 2016). [QRC 4.3] Segundo os autores, trata-se de um documento definitivo, embora em constante evolução, sobre o que é e o que não é Scrum.

Segundo Schwaber e Sutherland, Scrum não é um processo, mas um *framework* com o qual pessoas podem abordar problemas complexos e adaptativos, como o desenvolvimento de software, enquanto de forma produtiva e criativa entregam produtos com o maior valor possível.

Assim, o Scrum não é composto por descrições de processos, como no caso dos modelos prescritivos. O Scrum é composto pelos seus papeis, artefatos, reuniões e regras, como veremos nas subseções seguintes.

Scrum como modelo para desenvolvimento de software foi imaginado especificamente para lidar com os seguintes problemas recorrentes:

- *Lei de Ziv*: especificações nunca serão totalmente completas.
- *Lei de Humphrey*: o usuário nunca saberá o que ele quer até que o sistema esteja em uso (e talvez nem então).
- *Lema de Wegner*: um sistema interativo nunca será totalmente especificado e nem poderá ser totalmente testado (analogia ao Teorema de Goedel).
- *Lema de Langdon*: software evolui mais rapidamente à medida que ele se aproxima de regiões caóticas, desde que não se perca o controle para o caos.

Os pilares que sustentam o Modelo *Scrum* são transparência, inspeção e adaptação. *Transparência* indica que os aspectos importantes do projeto devem ser visíveis para os interessados. Assim, clientes devem ser capazes de acompanhar o estágio do desenvolvimento da forma mais dinâmica possível. Ao mesmo tempo, os desenvolvedores devem ser capazes de perceber as necessidades do cliente bem como o grau em que a equipe o está satisfazendo.

Inspeção deve ser sempre possível no sentido de que qualquer trabalho feito por alguém possa ser inspecionado por outro. O objetivo da inspeção não é punitivo, mas sim de certificar que o trabalho foi bem feito ou de corrigir eventuais desvios.

Adaptação é necessária porque o processo de desenvolvimento é um amadurecimento durante o qual requisitos vão sendo naturalmente descobertos ou atualizados. Assim, as regras de Scrum procuram justamente facilitar essa adaptação sempre que ela for necessária.

4.1.1 Papéis

A equipe que desenvolve um projeto usando *Scrum* é denominada *Scrum team* (equipe *Scrum*), e é formada por três papeis apenas:

- O *Scrum master*[1]

[1]Frequentemente, nomes de papeis e artefatos em *Scrum* e outros modelos ágeis são grafados em maiúsculas, como *Scrum Master*. Porém, entendemos que pelas regras gramaticais isso deve ser evitado. Apenas o nome do modelo, por ser nome próprio, será então grafado com maiúscula neste texto. Por outro lado, mantivemos a maioria dos termos em inglês, pois eles são assim conhecidos no Brasil.

  (mestre de *Scrum*), que deve ser uma pessoa com profundo conhecimento sobre a forma de aplicar as regras de *Scrum* na prática. Esse papel não é equivalente ao de um gerente que pode existir em outros modelos, mesmo ágeis. *Scrum* considera que não deve haver um gerente, já que as equipes são auto-organizadas. Assim, o *Scrum master* é uma espécie de servidor-líder, facilitador e solucionador de conflitos. Mesmo o aspecto de liderança dele não se dá no sentido de decidir quais requisitos devem ser abordados, nem de como as atividades de desenvolvimento devem ser feitas; sua liderança se refere tão somente a mostrar para a equipe como *Scrum* deve ser aplicado e indicar possíveis desvios em relação aos pilares de transparência, inspeção e adaptação do modelo. Ele é o guardião das regras de *Scrum*.
- O *product owner* (PO – dono do produto), ou seja, a pessoa responsável pelo projeto em si. Tem, entre outras atribuições, a de indicar quais são os requisitos mais importantes a serem tratados em cada *sprint*. O PO (pronuncia-se "pi-ou") é o responsável pelo ROI (*return of investment* – retorno do investimento) e por conhecer e avaliar as necessidades do cliente. Em sua atividade ele pode e deve sempre ouvir a equipe de desenvolvimento e quaisquer outros interessados e, no final, deverá identificar quais atividades trarão mais valor para o cliente com o menor custo possível. Seu papel é fundamental especialmente nas reuniões de planejamento, já que decisões ruins de planejamento podem levar o projeto a amargar prejuízos.
- O *development team*, que é a equipe de desenvolvimento. Segundo o guia de *Scrum*, essa equipe não deve ser dividida em papéis como analista, *designer*, programador ou testador, pois todos interagem para desenvolver o produto em conjunto.

Tanto o *Scrum master* quanto o *product owner* podem ser ou não membros do *development team*. Recomenda-se que o tamanho de um *development team* varie de três a nove pessoas. Nesse limite, só se conta o *Scrum master* e o *product owner* se eles também fazem parte do *development team*, ou seja, se eles, além das suas responsabilidades específicas também contribuem com atividades de desenvolvimento.

Dentro do *development team* o único papel reconhecido pelo modelo é o de *desenvolvedor*. O modelo considera que a equipe deve ser formada por pessoas com habilidades suficientes para realizar em conjunto tudo o que for necessário para a produção do produto, mas o modelo não separa atividades e papeis específicos como outros modelos fazem. O *Scrum* também não reconhece subequipes dentro da equipe de desenvolvimento, sem exceções.

Considera-se que a equipe precisa ter no mínimo três pessoas pois senão haverá pouca oportunidade de interação e não mais do que nove pessoas porque neste caso, medidas de organização mais complexas precisarão ser tomadas e a aplicação do modelo se torna mais difícil na prática. Projetos que exijam equipes maiores podem se valer de recursos de escalabilidade que serão explicados mais adiante.

4.1.2 Artefatos do *Scrum*

Os principais artefatos do *Scrum* são o *product backlog*, o *sprint backlog* e o *incremento*, que serão explicados nas subseções seguintes. O objetivo destes artefatos é principalmente promover a comunicação e a transparência em relação ao estado do produto sendo desenvolvido.

4.1.2.1 *Product Backlog*

As funcionalidades a serem implementadas em cada projeto são mantidas em uma lista chamada de *product backlog*. Os itens dessa lista são usualmente identificados como *histórias de usuário*, ou seja, descrições feitas pelo próprio usuário de como ele se vê usando o produto. Essas descrições correspondem a uma primeira abordagem aos requisitos de um projeto. Opcionalmente, também podem ser usados outros elementos mais elaborados como casos de uso que correspondem a uma forma mais sistematizada de apresentar histórias de usuário e que usualmente são produzidos por analistas a partir das descrições do usuário.

De qualquer forma, o *product backlog* deve ser considerado como um artefato dinâmico que está sempre em atualização. Inclusive após a entrega final do produto ele pode continuar evoluindo, pois quaisquer requisições de manutenção ou evolução do produto, como novas funcionalidades, poderão ser organizadas neste artefato.

Um dos princípios do manifesto ágil é usado aqui: *adaptação* em vez de planejamento. Então o *product backlog* não precisa ser detalhado no início do projeto, pois haverá atividades de refinamento dele ao longo do projeto. Pode-se iniciar apenas com as funcionalidades mais evidentes, para depois, à medida que o projeto avançar, tratar novas histórias de usuário que forem sendo descobertas. Isso, porém, não significa fazer um levantamento inicial excessivamente superficial. Deve-se tentar obter com o cliente o maior número possível de informações sobre suas necessidades antes de iniciar o desenvolvimento destas. Mas espera-se que aquelas que efetivamente surgirem no início dessa interação tenham maior relevância do que outras que forem descobertas mais adiante.

O *product backlog* consiste em uma lista com campos cuja composição varia muito conforme a referência bibliográfica. Mas segundo o guia oficial de *Scrum* de Schwaber e Sutherland, os campos obrigatórios são: *descrição*, *ordem*, *estimativa* e *valor*. A Tabela 4.1 apresenta um pequeno exemplo de como poderia ser este artefato para uma livraria online.

Tabela 4.1 Exemplo de *product backlog*

Id	Descrição	Ordem	Estimativa	Valor
3	Como usuário anônimo eu gostaria de poder visualizar a lista de livros à venda	1	2	alto
8	Como usuário identificado eu gostaria de poder comprar livros online	2	5	alto
9	Como usuário identificado eu gostaria de poder cadastrar um novo endereço de entrega	3	1	médio
17	Como responsável pelo estoque eu gostaria de poder visualizar a quantidade de cada livro em estoque.	4	1	médio
13	Como gerente de vendas eu gostaria de poder visualizar relatórios semanais ou mensais de vendas por livro.	5	3	médio
11	Como usuário identificado eu gostaria de poder cancelar um pedido recém realizado.	6	2	baixo

A *descrição* deve ser uma frase simples e possivelmente curta que explique qual a necessidade do cliente que o item representa. Usualmente trata-se de histórias de usuário, mas não apenas referentes à criação de novas funcionalidades como também de sua alteração ou eliminação. Como a descrição dos itens em um *product backlog* pode mudar com o tempo, recomenda-se adicionar também um campo com um identificador numérico para o item (campo *Id*), para que se possa rastreá-lo mesmo se sua descrição mudar.

Usualmente, uma história de usuário é descrita do ponto de vista de um papel de usuário no sistema. Assim, elas, via de regra, são descritas da forma "como um usuário *de tal tipo*, eu gostaria de...". Por exemplo, "como um administrador eu gostaria de visualizar a lista de todos os usuários do sistema" ou "como um usuário sem cadastro eu gostaria de poder me cadastrar no sistema".

A *ordem* seria a prioridade do item, sendo que os primeiros da lista usualmente são aqueles cujo desenvolvimento é considerado mais urgente. Aqui há autores que recomendam que em vez de usar números de prioridade como 1, 2, 3 etc., sejam atribuídos valores de importância, como 10, 20, 30 etc., sendo que os itens mais importantes são aqueles com valor mais alto. Usar intervalos maiores entre os números (10 em 10, como no exemplo) permite inserir itens com importância intermediária no futuro, como 15, 17, 25 etc. Além disso, se um item com prioridade maior do que o primeiro da lista for descoberto, no caso de números de prioridade ele terá prioridade zero? Com valores de importância bastará escolher um número mais alto do que o mais alto usado até então. Porém, se o *Product Backlog* for registrado em uma ferramenta computacional, essa questão se torna irrelevante pois a própria ferramenta pode reorganizar as prioridades quando houver alguma inserção na lista.

A *estimativa* quantifica o esforço necessário para transformar a história em produto acabado segundo a *definição de feito* (DoD, ou *definition of done*). O valor é dado em *pontos de histórias* (*PH*). Veja mais detalhes na Seção 7.2.

Já o *valor* indica o quanto a história gera de retorno para o cliente. Esse valor tanto pode ser numérico (financeiro, por exemplo), como estimado em escala-camiseta com valores como pequeno, médio e grande. Usualmente espera-se que itens com grande valor para o cliente tenham prioridade mais alta, especialmente se sua estimativa de esforço for baixa. Mas esse nem sempre é o caso. Muitas vezes, itens de menor valor precisam ser priorizados pois eles geram conhecimento que a equipe precisa ter para então sim produzir os itens de maior valor. Isso é particularmente importante no caso de itens que implicam em mitigação de riscos de projeto.

4.1.2.2 *Sprint Backlog*

Além do *product backlog*, *Scrum* propõe o uso de um *sprint backlog*. Pode-se dizer que os dois *backlogs* têm naturezas diferentes:

- O *product backlog* apresenta requisitos de alto nível, normalmente representados como histórias de usuário e, portanto, bastante voltados às necessidades diretas do cliente.
- Já o *sprint backlog* apresenta um detalhamento desses requisitos de forma mais voltada à maneira como a equipe vai desenvolvê-los.

O *product backlog* é uma tabela usualmente única para um projeto que vai evoluindo ao longo do desenvolvimento e mesmo da manutenção do software. Já o *sprint backlog* é uma tabela que é reinicializada a cada iteração (*sprint*).

O *sprint backlog* deve conter o plano da iteração. Nele, a equipe vai fazer constar as atividades que devem ser desenvolvidas para transformar em produto feito um ou mais itens do *product backlog* escolhidos como objetivos da iteração.

Há uma reunião específica chamada *sprint planning meeting* para a inicialização do *sprint backlog* a cada início de *sprint*. Mas, ao longo da iteração, novas atividades podem ser adicionadas pela equipe de desenvolvimento, *desde que* dentro do escopo. Eventualmente atividades também podem ser removidas, mas se essa remoção reduzir o escopo deve haver concordância do *product owner*.

Há diferentes recomendações sobre o que constar no *sprint backlog*. Via de regra o usual é que ele contenha na primeira coluna as histórias de usuário que foram selecionadas, seguida de uma segunda coluna na qual constam as tarefas necessárias para transformar cada história em produto feito.

A terceira coluna poderá conter então as tarefas em andamento, que são transferidas da segunda coluna e a quarta coluna terá as tarefas feitas, de acordo com a *definição de feito* (DoD). Algumas equipes, porém, poderão adicionar outras colunas, como por exemplo, uma coluna intermediária "a revisar" entre a coluna das tarefas em andamento e as tarefas feitas. Isso será particularmente útil para manter a equipe dentro do princípio de que tudo o que é feito deve ser revisado. Mais adiante no texto será explicado como as equipes trabalham com o *sprint backlog*.

A Tabela 4.2 mostra um exemplo de *sprint backlog* com dois objetivos retirados do *product backlog* da Tabela 4.1.

Tabela 4.2 Exemplo de *sprint backlog*

Objetivo	A fazer	Em andamento	A revisar	Feito
3. Como usuário anônimo eu gostaria de poder visualizar a lista de livros à venda	Realizar teste exploratório Programar teste automatizado	Programar consulta ao BD	Programar Interface	Revisar requisitos Desenhar interface
8. Como usuário identificado eu gostaria de poder comprar livros online	Desenhar interface Programar interface com operadora de cartão de crédito Programar funções de consulta e compra Programar Interface Realizar teste exploratório Programar teste automatizado	Revisar requisitos		

Embora existam várias ferramentas, usualmente gratuitas para elaboração de *sprint backlog*, observa-se (a partir da literatura e experiência prática) que a maioria das equipes ainda prefere usar quadros brancos e *post-its* coloridos de papel para gerenciar seu *sprint backlog*. Um exemplo é mostrado na Figura 4.1. Nestes casos, registros históricos dos quadros são mantidos a partir de fotografias tiradas deles diariamente.

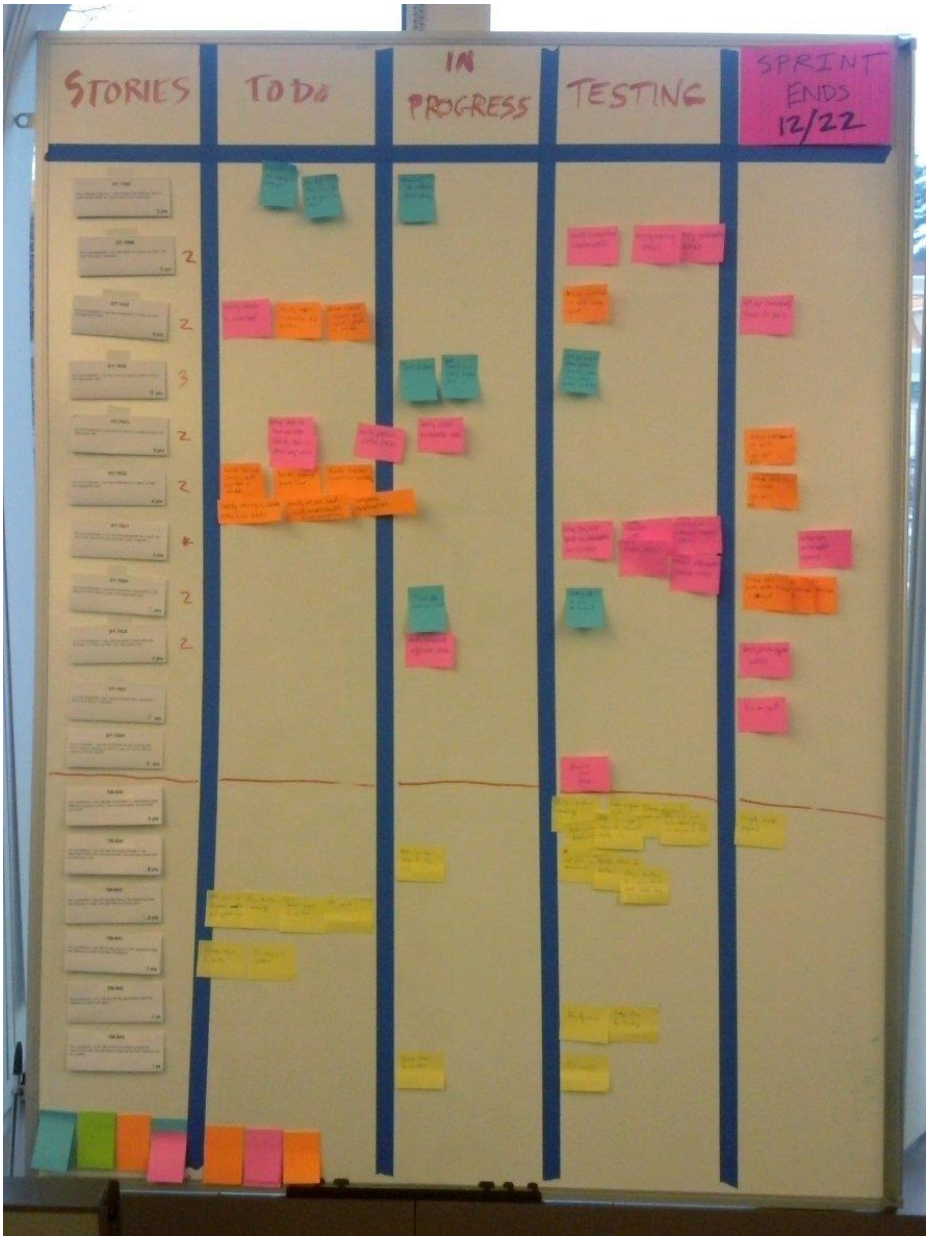

Figura 4.1 Exemplo concreto de *sprint backlog* (Fonte: Logan Ingalls - Task board, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=19838361)

4.1.2.3 Incremento e DoD

O *incremento* consiste no resultado de todo o trabalho realizado durante uma *sprint*. Usualmente trata-se de um novo estado para o produto no qual os objetivos da *sprint* foram incorporados.

É fundamental que a equipe, juntamente com os demais interessados, especialmente o *product owner*, tenham um entendimento comum e objetivo sobre o que significa estar *feito* ou *pronto*. Em inglês, isso é denominado de DoD ou *definition of done*. Por exemplo, algumas equipes poderão considerar o produto feito quando ele passar em todos os testes exploratórios. Outras equipes poderão considerar feito apenas quando o incremento também incluir testes automatizados. Outros ainda poderão exigir que o produto seja aprovado em uma inspeção de código, de forma a eliminar problemas de estilo e padronização que usualmente não são capturados no teste.

Enfim, existem muitas possibilidades em relação ao entendimento do que significa que um produto está feito, mas o importante é que a equipe e os demais interessados tenham um entendimento comum sobre isso.

4.1.3 Eventos do *Scrum*

Um dos pilares que sustentam o *Scrum* são seus eventos. O modelo prevê quatro tipos de reunião e apenas um tipo de iteração de desenvolvimento chamada de *sprint*. As reuniões são as de planejamento da *sprint*, avaliação do produto, avaliação da equipe e a reunião diária em pé.

Todos esses cinco eventos seguem a técnica de *timeboxing*, ou seja, sua duração é estabelecida previamente e deve ser rigorosamente respeitada pela equipe.

4.1.3.1 *Sprint*

A *sprint* é o ciclo de desenvolvimento de poucas semanas de duração sobre o qual se estrutura o *Scrum*. Durante a *sprint*, cabe ao product owner manter o *sprint backlog* atualizado, indicando as tarefas já concluídas e aquelas ainda por concluir, preferencialmente mostradas em um gráfico atualizado diariamente e à vista de todos.

A cada dia pode-se descobrir que tarefas que não foram inicialmente previstas eram necessárias para implementar as histórias de usuário do *sprint*. Essas novas tarefas devem ser colocadas na coluna de tarefas por fazer. Porém, novas histórias de usuário não podem ser adicionadas durante o *sprint* pelo *product owner*; apenas tarefas contidas no escopo já definido, mas não previstas. Por exemplo, se a equipe tem como objetivo desenvolver a história de usuário "fazer pedido", o *product owner* não pode adicionar outras histórias de usuário como "veicular campanha de marketing", mas a equipe pode adicionar atividades incluídas em "fazer pedido" que ela própria não tenha previsto na reunião de planejamento, como, por exemplo, "projetar banco de dados de pedidos" ou "prototipar interface de pedidos".

A cada dia, também, pode-se avaliar o andamento das atividades, contando a quantidade de atividades por fazer e a quantidade de atividades terminadas, o que vai produzir o diagrama *sprint burdown* (Seção 4.1.4).

4.1.3.2 *Sprint Planning Meeting*

No início de cada *sprint* é feito um *sprint planning meeting*, reunião na qual a equipe prioriza os elementos do *product backlog* a serem implementados e transfere esses elementos do *product backlog* para o *sprint backlog*. Esses elementos correspondem à lista de funcionalidades a serem implementadas no ciclo que se inicia, ou sejam o escopo da iteração.

A equipe se compromete a desenvolver as funcionalidades, e o *product owner* se compromete a não trazer novas funcionalidades durante o mesmo *sprint*. Se novas funcionalidades fora do escopo forem descobertas, serão abordadas em *sprints* posteriores.

O *sprint backlog* é inicializado com as histórias de usuário (primeira coluna), retiradas do *product backlog*. A equipe, então, se reúne para determinar quais são as atividades de desenvolvimento necessárias para implementar cada uma das histórias de usuário. Essa lista de atividades é usada para inicializar a segunda coluna "tarefas a fazer" do *sprint backlog*. À medida que as atividades vão sendo feitas, verificadas e terminadas, os *postits* correspondentes vão sendo movidos para a coluna da direita.

4.1.3.3 *Sprint Review* e *Sprint Retrospective*

Ao final de cada *sprint*, a equipe deve realizar uma *sprint review meeting* (ou *sprint demo*) para verificar o que foi feito e, então, partir para uma nova *sprint*. A *sprint review meeting* é a demonstração e a avaliação do produto da *sprint*. Nesta reunião é fundamental que o cliente ou seus representantes que tenham autoridade para homologar requisitos estejam presentes, pois será o momento de verificar se o produto efetivamente satisfaz não só a definição de feito, mas também os requisitos inicialmente estabelecidos. Além disso, será o momento de validar através da observação do produto se os requisitos inicialmente estabelecidos de fato representam as reais necessidades dos interessados.

Outra reunião que pode ser feita ao final de uma *sprint* ou no início da próxima é a *sprint retrospective*, cujo objetivo é avaliar a equipe e os processos (impedimentos, problemas, dificuldades, ideias novas etc.). Este é o momento em que a equipe vai refletir sobre seus métodos de trabalho, enfatizando práticas que deram certo e devem ser mantidas e refletindo como melhorar práticas que demonstraram ser insuficientes ou inadequadas.

A rigor, uma reunião de reflexão pode também ser convocada a qualquer momento, mesmo durante uma *sprint*, caso a equipe verifique que o processo de trabalho está prejudicando a obtenção dos objetivos da *sprint* e precisa ser melhorado.

4.1.3.4 *Daily Scrum*

O *Scrum* sugere que a equipe realize uma reunião diária, chamada *daily scrum*, na qual o objetivo é fazer que cada membro da equipe fale sobre o que fez no dia anterior, o que vai fazer no dia seguinte e, se for o caso, o que o impede de prosseguir.

Essas reuniões devem ser rápidas. Por isso, sugere-se que sejam feitas com as pessoas em pé em frente ao *sprint backlog* ou a um quadro de anotações. Além disso, recomenda-se que sejam feitas sempre no mesmo horário e no mesmo local. A reunião deve ser limitada a no máximo 15 minutos e deve iniciar pontualmente no horário previsto, mesmo que alguns participantes ainda não tenham chegado.

É desse formato de reunião em pé (*scrum*, em inglês), semelhante ao que jogadores de alguns esportes fazem nos intervalos, que vem o nome do método.

4.1.4 *Sprint Burndown Chart*

Um artefato que não é considerado parte do núcleo do *Scrum*, mas que mesmo assim é bastante usado é o *sprint burndown chart*. O diagrama consiste basicamente de uma linha que indica a quantidade de trabalho por fazer. A cada dia, a quantidade de tarefas por fazer ou o somatório do seu número de pontos de história é somado e o valor correspondente é desenhado no gráfico. Um exemplo de *sprint burndown chart* é mostrado na Figura 4.2.

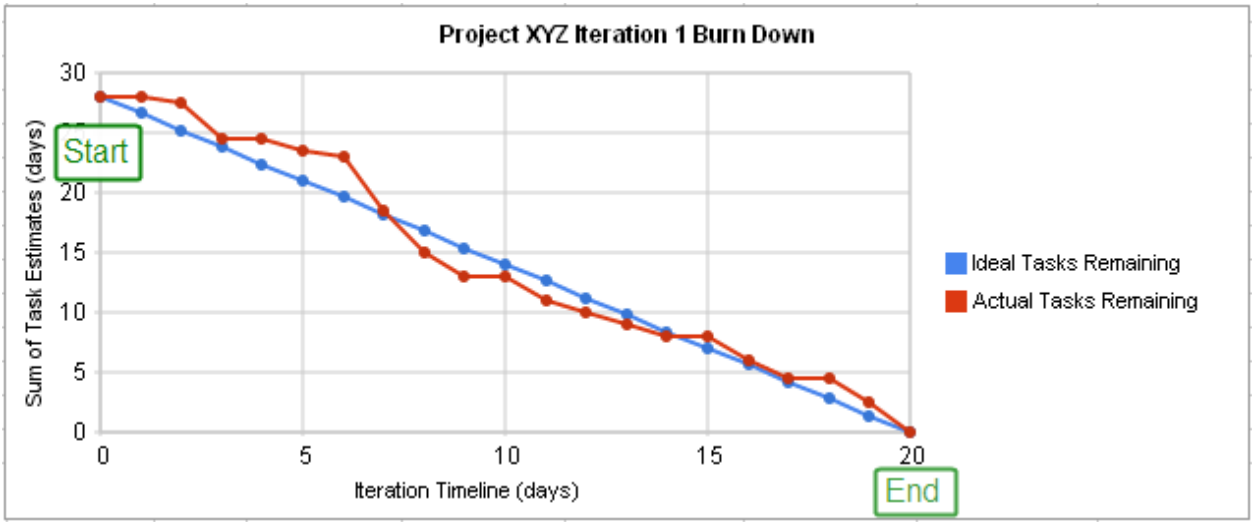

Figura 4.2 Exemplo de *sprint burndown chart* (Fonte: I8abug - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15511814)

TRADUÇÃO:
Sum of task estimates (days)//Soma da estimação das tarefas (dias)
Project XYZ Iteration 1 Burn Down//Projeto XYZ Iteração 1 Burn Down
Start//Início
End//Fim
Iteration timeline (days)//Linha de tempo da iteração (dias)
Ideal tasks remaining//Atividades restantes (ideal)
Actural tasks remaining//Atividades restantes (real)

O gráfico é atualizado todo dia, do início até o final da *sprint*, e espera-se, numa situação ideal, que a quantidade de tarefas por fazer gradualmente diminua até ficar completamente zerada ao final da *sprint*.

Porém, quando as histórias de usuário não foram bem compreendidas e/ou as tarefas não foram bem planejadas, ou ainda, quando a equipe atribui a si mesma tarefas extras, além das inicialmente previstas, a quantidade de tarefas por fazer pode aumentar ao invés de diminuir ao longo do tempo.

Mar (2006) analisa as linhas de tarefas por fazer, identificando sete tipos de comportamentos de equipes conforme seus *sprint burndown chart*:

- *Fakey-fakey*: caracteriza-se por uma linha reta e regular que indica que provavelmente a equipe não está sendo muito honesta, porque o mundo real é bem complexo e dificilmente projetos se comportam com tanta regularidade (Figura 4.3).
- *Late-learner*: indica um acúmulo de tarefas até perto do final da *sprint* (Figura 4.4). É típico de equipes iniciantes que ainda estão tentando aprender o funcionamento do *Scrum*. Essas equipes, só no final da *sprint* percebem que tarefas que deviam ter sido planejadas no escopo não o foram. É muito provável que essa equipe não consiga atingir os objetivos da *sprint* pois, por não ser capaz de prever tarefas, podem ter sido muito otimistas em relação ao tempo necessário para produzir o incremento.
- *Middle-learner*: indica que a equipe pode estar amadurecendo e começando mais cedo as atividades de descoberta e, especialmente, os testes necessários (Figura 4.5). Essa equipe

ainda poderá ter problemas no final da *sprint*, mas possivelmente bem menos do que os *late-learner*.

- *Early-learner*: indica uma equipe que procura, logo no início da *sprint*, descobrir todas as necessidades e depois desenvolvê-las gradualmente até o final do ciclo (Figura 4.6). Eles não são necessariamente capazes de identificar todas as tarefas na *sprint planning meeting*, mas em poucos dias terão uma visão bem mais completa das tarefas necessárias e terão tempo fazer os ajustes para que a *sprint* termine no prazo certo e com seus objetivos atingidos.
- *Plateau*: indica uma equipe que vai descobrindo novas tarefas ao longo da *sprint*, o que acaba levando o ritmo de desenvolvimento a um platô (Figura 4.7). Para cada tarefa feita uma nova tarefa é descoberta e assim, embora a equipe esteja trabalhando, existe a impressão de que o trabalho não está saindo do lugar.
- *Never-never*: indica uma equipe que acaba tendo surpresas desagradáveis no final de uma *sprint* (Figura 4.8). Subitamente, uma quantidade grande de tarefas necessárias é descoberta quando se achava que os objetivos já estavam praticamente atingidos.
- *Scope increase*: indica uma equipe que no início da *sprint* percebe um súbito aumento na carga de trabalho por fazer (Figura 4.9). Usualmente, a solução nesses casos é tentar renegociar o escopo da *sprint* com o *product owner*, mas não se descarta também uma finalização ou cancelamento da *sprint* para que seja feito um replanejamento do *product backlog*.

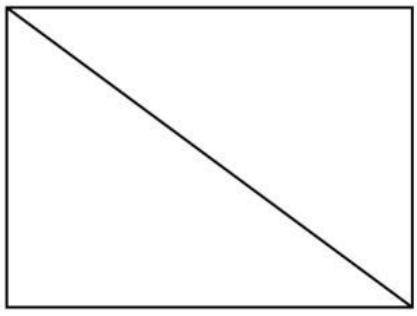

Figura 4.3 *Fakey-fakey*

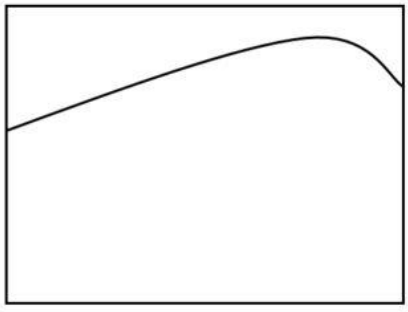

Figura 4.4 *Late-learner*

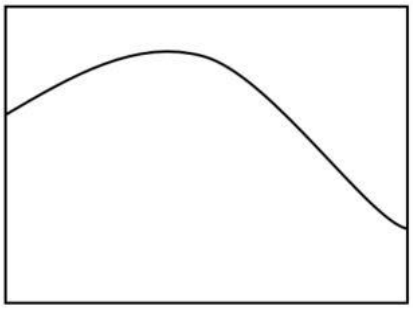

Figura 4.5 *Middle-learner*

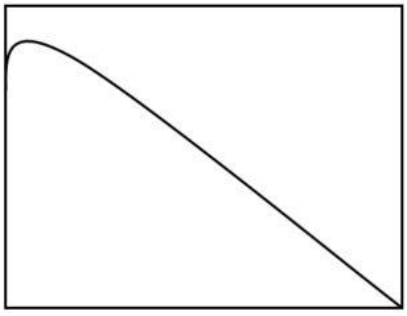

Figura 4.6 *Early-learner*

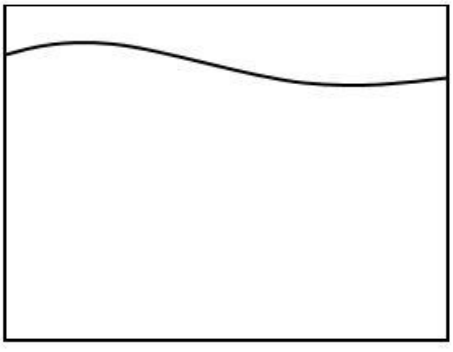

Figura 4.7 *Plateau*

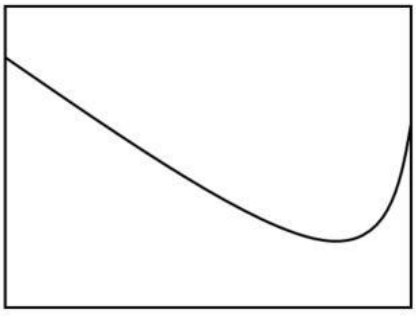

Figura 4.8 *Never-never*

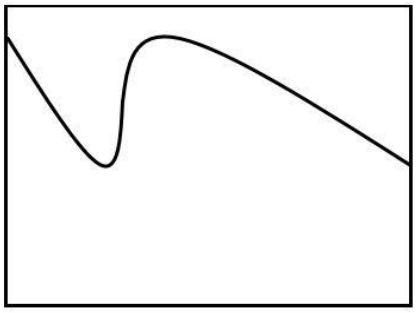

Figura 4.9 Scope increase

A diferença entre o *scope increase* e o *early-learner* é que no segundo caso a equipe ainda terá condições de terminar a *sprint* com sucesso ao fazer ajustes ao longo da iteração, mas no caso do *scope increase* a equipe não será capaz de cumprir com estes compromissos, sendo então necessário renegociar escopo e eventualmente até cancelar a *sprint*, o que não é um evento muito agradável, mas eventualmente pode ser necessário para que a equipe se reorganize e replaneje as *sprints*.

Algumas equipes também poderão adicionar ao gráfico o *burn up*, ou seja, o índice de atividades efetivamente finalizadas, de acordo com a DoD.

4.1.5 Funcionamento Geral do *Scrum*

O funcionamento geral do modelo *Scrum* pode ser entendido a partir do resumo apresentado na Figura 4.10. Basicamente, o lado esquerdo da figura mostra o *product backlog*, com as histórias de usuário, que devem ser priorizadas e ter sua complexidade estimada. As histórias mais importantes são, assim, selecionadas durante a *sprint planning meeting*, até que o número de pontos de história se aproxime da capacidade de produção da equipe durante o *sprint*. Cada ponto de história implica um dia de trabalho por pessoa. Assim, um *sprint* de duas semanas (dez dias de trabalho) com três pessoas teria a capacidade de acomodar trinta pontos de história.

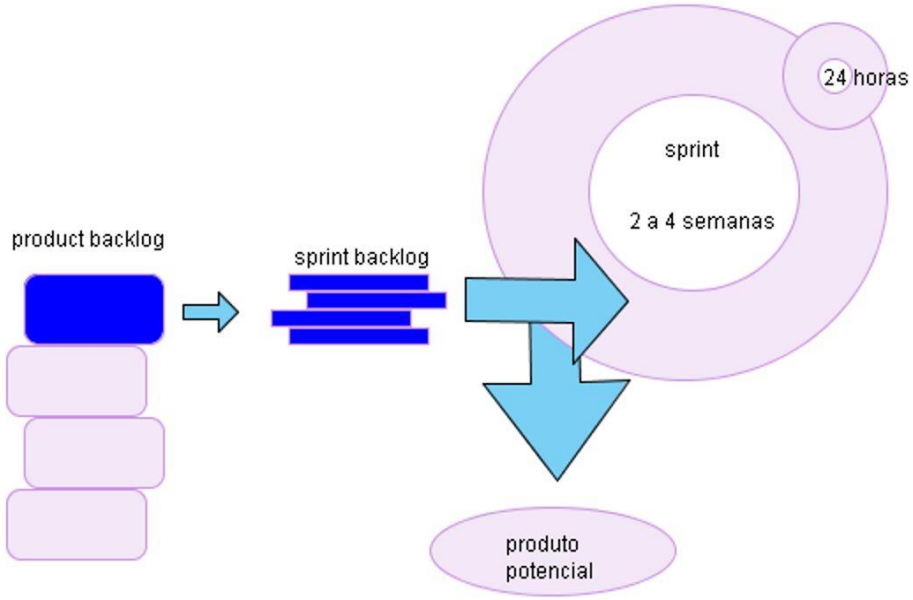

Figura 4.10 Funcionamento geral do *Scrum*

Ainda durante a *sprint planning meeting*, as histórias de usuário selecionadas devem ser detalhadas em atividades de desenvolvimento, ou seja, as tarefas do *sprint backlog* devem ser identificadas.

A *sprint* se inicia e, a cada 24 horas, deve acontecer uma *Scrum daily meeting*, conforme indicado no círculo menor da figura.

Ao final da *sprint* deverá haver a *sprint review meeting*, para avaliar o produto do trabalho (incremento), e, eventualmente, a *sprint retrospective*, para avaliar os processos de trabalho. Assim, se aprovado, o produto (parcial ou final) poderá ser entregue ao cliente. Não sendo essa a *sprint* final, o ciclo é reiniciado.

4.1.6 LeSS – *Large Scale Scrum*

No caso de projetos muito grandes, é possível aplicar o conceito de *Scrum of Scrums* (Cohn, 2007), em que vários *Scrum teams* trabalham em paralelo e cada um contribui com uma pessoa para a formação do *Scrum of Scrums*, quando então as várias equipes são sincronizadas. Essa pessoa não precisa necessariamente ser o *product owner* ou *Scrum master*. Pode ser qualquer membro da equipe.

Mais recentemente, o *Scrum* em grande escala foi sistematizado em um framework conhecido como LeSS, ou *Large Scale Scrum* (Larman & Vodde, 2016). O modelo é baseado em *Scrum* e especialmente no *Scrum of Scrums* como forma de organizar o trabalho com várias equipes.

LeSS parte do princípio de que usar um modelo complexo e aplicar apenas partes dele em projetos mais simples é um grande erro estratégico. Assim, a ideia é iniciar com um modelo simples e enxuto como o *Scrum* e adicionar apenas os elementos necessários para que o trabalho possa ser realizado com equipes maiores e projetos mais complexos.

A ideia, entretanto, não é aumentar o tamanho do *development team*, que deve continuar a ser limitado a nove pessoas, mas sim, criar vários *development teams* que trabalharão em paralelo durante as *sprints*.

O modelo prevê a existência de um único *product backlog* e um único *product owner* que vai selecionar objetivos para as *sprints* de cada equipe. As equipes então elaboram seus *sprint backlogs* específicos e passam a trabalhar. Já um *Scrum master* pode atender de uma a três equipes de desenvolvimento.

Há também um *product owner* para todo o projeto. Toda a priorização passa pelo *product owner*, mas a clarificação ocorre tanto quanto possível diretamente entre as equipes ou entre elas e outros interessados.

Deve-se fazer o refinamento do *product backlog* (PBR – *product backlog refinement*) em uma reunião geral para aumentar o conhecimento compartilhado e explorar oportunidades de coordenação quando há itens fortemente correlacionados ou a necessidade de uma aprendizagem mais ampla. O *product owner* não deve trabalhar sozinho no refinamento do *product backlog*; ele deve ser sempre apoiado pelas equipes e interessados.

Existe uma única *sprint* em nível de produto na qual todas as equipes trabalham em paralelo. Mas cada equipe tem e gerencia o seu próprio *sprint backlog*.

O *sprint planning meeting* consiste de duas partes: o *sprint planning one* é uma reunião geral de todas as equipes; já o *sprint planning two* é feito separadamente para cada equipe, embora se recomende que as reuniões sejam simultâneas e em ambientes próximos, caso interações ainda sejam necessárias.

O *sprint planning one* é feito com representantes de cada equipe ou, se possível, com as equipes completas. Juntos eles tentam selecionar os itens do *product backlog* que cada equipe vai trabalhar na próxima *sprint*.

A iteração entre as equipes ocorre com as reuniões de *Scrum of Scrums*, realizadas com um representante de cada equipe, na qual questões de alinhamento entre as equipes são identificadas da mesma forma que as questões de alinhamento entre os membros de uma equipe são identificadas no *daily meeting*.

Deve-se preferir coordenação decentralizada e informal em vez de coordenação centralizada. A coordenação de esforços entre equipes é decidida pelas próprias equipes.

É prevista uma reunião de todas as equipes apenas no final de uma *sprint* para a revisão do produto (*sprint review*) e reuniões separadas de retrospectiva para cada equipe. Há, assim, uma única *sprint review* envolvendo todas as equipes.

Em relação a *sprint retrospective*, cada equipe fará sua própria reunião de retrospectiva após a *sprint* e, em seguida, haverá uma reunião de retrospectiva geral, com um representante de cada equipe para tratar de questões envolvendo as relações entre as equipes e também para trocas de experiências entre as equipes. A meta de perfeição para as equipes é melhorar a DoD de forma que isso resulte em produtos entregáveis ao fim de cada *sprint* ou até mais frequentemente.

O modelo aponta como fundamental que as equipes sejam divididas em função das histórias de usuário, ou funcionalidades, que cada uma vai desenvolver. Não devem ser criadas equipes especializadas, como por exemplo, equipe de banco de dados, equipe de interface ou equipe de programação.

O modelo enfatiza que LeSS consiste em aumentar *Scrum* usando *Scrum*, ou seja, não são criados novos artefatos, papeis ou processos para que o modelo funcione. As equipes trabalham em paralelo, mas não subdividem o projeto entre elas; todas trabalham no produto como um todo. Existe um único *product owner* e uma *sprint* única para todas as equipes. Larman e Vodde argumentam que quando um projeto é subdividido entre as equipes, por área de negócio, por exemplo, cada equipe vai procurar otimizar o produto considerando sua subárea, mas assim, o

produto como um todo não será otimizado. Então, eles propõem que todas as equipes possam pensar em todo o produto, e não apenas em partes especializadas.

Essa organização é recomendada para projetos com duas a oito *development teams* trabalhando em paralelo. Essa abordagem é também conhecida como *smaller* LeSS. Para projetos que demandam mais do que oito equipes, recomenda-se a abordagem LeSS *Huge*.

4.1.7 LeSS *Huge*

Segundo Larman e Vodde (2016), *LeSS Huge* chegou a ser aplicado em um projeto que envolveu milhares de desenvolvedores e gerou dezenas de milhões de linhas de código em C++, além de hardware especializado. Em relação a *smaller LeSS*, em *LeSS Huge*, o que permanece igual é:

- Um único *product backlog*.
- Uma única definição de feito (DoD).
- Um único incremento de produto potencialmente entregável.
- Um *product owner* (geral).
- Uma *sprint* com todas as equipes trabalhando em paralelo para gerar um único incremento.

O que muda:

- *Mudança de papel*: passam a existir *product owners* por área de requisitos.
- *Mudanças de artefatos*: passam a existir áreas de requisitos no *product backlog* ou *product backlogs* por área de requisitos.
- *Mudanças em reuniões*: o *LeSS Huge* é um conjunto de execuções de *sprints* LeSS por área de requisitos.

A principal mudança filosófica no caso de *LeSS Huge* vem da observação de que equipes do tipo *Scrum team* podem ser bem organizadas em *Scrums of Scrums* desde que o número de equipes não passe de oito. Acima desta quantidade de equipes será necessário dividir o projeto em áreas de requisitos.

Essa divisão deve ser elaborada pelo *product owner* geral e ela não é fixa, podendo mudar ao longo do projeto, embora não se espere mudanças muito frequentes. O modelo não recomenda que as equipes sejam divididas por área técnica, mas sim por áreas de requisitos, sempre do ponto de vista do cliente. Não são recomendadas, então, equipes especializadas em interface, arquitetura, banco de dados etc. Mas, caso se esteja elaborando um projeto de software na área comercial, por exemplo, pode-se ter equipes especializadas em sistemas de vendas, de recursos humanos, de marketing etc.

Cada área de requisitos terá também seu próprio *product owner* (APO – *area product owner*) que têm as mesmas atribuições do PO, mas dentro de uma área de requisitos delimitada. O conjunto dos APOs mais o PO forma o *product owner team*, que ajuda o PO a tomar decisões sobre priorização de requisitos no *product backlog* geral.

4.2 Lean

Lean é entendido por muitos como uma filosofia de produção que se propõe a eliminar desperdícios, gerar valor com eficiência e aproveitar ao máximo o potencial humano em projetos.

Lean surgiu na indústria automobilística, mas também acabou sendo adaptado para a produção de software, sendo conhecido hoje como LSD ou *Lean Software Development*. Esse termo foi cunhado por Poppendieck e Poppendieck (2003).

A filosofia *Lean* se baseia em sete princípios que são:

- Eliminar o desperdício.
- Amplificar a aprendizagem.
- Decidir o mais tarde possível.
- Entregar o mais rápido possível.
- Empoderar a equipe.
- Construir integridade.
- Ver o todo.

O princípio de *eliminação de desperdício* ou de *lixo*, aplica-se na indústria de software pela identificação e eliminação de todo documento e trabalho que não seja importante para a produção do resultado desejado. Assim, o foco está em eliminar atividades parcialmente realizadas, retrabalho, requisitos desnecessários e soluções desnecessariamente complexas.

O desperdício precisa ser identificado e depois eliminado. Assim, código não terminado que não pode ser executado e muitas vezes será abandonado é considerado desperdício. Papelada e documentação que não é consultada e não agrega valor ao produto é considerada desperdício. Micro gerenciamento que não agrega efetivamente algum valor ao trabalho dos desenvolvedores é desperdício. Software com defeitos é considerado desperdício. E assim por diante.

Em relação a *amplificar a aprendizagem*, o *Lean* considera que o desenvolvimento de software é um processo iterativo de aprendizagem sobre o produto e sobre o seu uso. Muitas vezes se diz que nem o cliente sabe exatamente o que ele quer; assim, o modelo prega a realização de interações que sejam as mais curtas possíveis para que o *feedback* possa ser gerado e as partes envolvidas chegarem ao conhecimento mais perfeito possível sobre como o produto realmente deve ser.

O princípio de *decidir o mais tarde possível* procura evitar que decisões, por exemplo, de *design* ou tecnologia, sejam tomadas muito cedo, antes que as verdadeiras necessidades realmente possam ser conhecidas. Deve-se procurar trabalhar o essencial, mantendo todas as possibilidades em mente até que realmente haja razões para escolher uma ou outra opção. Tomar decisões cedo que depois precisam ser mudadas é considerado contra produtivo.

Considera-se que nos tempos atuais não é o mais forte que sobrevive, mas o mais rápido. Assim, quanto mais rapidamente a empresa conseguir liberar uma versão do produto funcionando sem maiores falhas, mais rapidamente ela conseguirá obter *feedback* dos interessados e, se necessário, incorporar este *feedback* ao produto. Este princípio de *entregar o mais rápido possível* vai fazer com que a equipe prefira sempre as iterações mais curtas e um contato constante com os interessados, para evitar atrasos ou entregas pouco frequentes, o que pode também gerar desperdícios.

Algumas pessoas ainda acreditam que cabe ao gerente decidir como os subordinados devem fazer o seu trabalho. Talvez isso funcione em algumas indústrias, mas dificilmente dá certo em áreas complexas como a construção de software. O princípio *Lean* de *empoderar a equipe* segue o princípio ágil que diz que você deve contratar pessoas competentes e deixar que elas façam o seu trabalho. O gerente não deve fazer planos sozinho nem distribuir tarefas; a própria equipe deve fazer isso trabalhando com transparência e compartilhando pontos de vista.

Um gerente pode incentivar o progresso, avaliar possíveis desvios e corrigir rumos, mas não deve micro gerenciar a equipe. Os membros da equipe não devem ser vistos apenas como recursos humanos, da mesma forma que computadores são recursos físicos; as pessoas precisam de motivação, confiança e tranquilidade para realizar um trabalho de alto nível.

Na relação com o cliente, a equipe desenvolvedora deve *desenvolver integridade* como um princípio. O cliente irá perceber integridade na medida que prazos forem cumpridos e produtos forem entregues com qualidade aceitável. Para isso, é necessário também que as entregas sejam feitas de forma contínua e não em grandes lotes cuja integridade é difícil de avaliar.

Equipes *Lean* também devem ser incentivadas a *ver o todo* e não apenas partes do sistema que estão desenvolvendo. Este princípio é também aceito por modelos como LeSS, que propõem que não devem existir equipes especializadas em aspectos técnicos ou arquiteturais do sistema, mas sim que todas as equipes devem ter a visão do todo e a possibilidade de trabalhar em qualquer parte do sistema.

Assim como LeSS é uma versão escalável de *Scrum*, também existe pelo menos uma versão escalável de *Lean*, que se denomina SAFe, ou *Scaled Agile Framework*. SAFe é marca e propriedade da empresa Scaled Agile, Inc. mas é disponibilizado gratuitamente no *link* representado no *QR Code* ao lado [QRC 4.4].

Frequentemente a filosofia *Lean* é também incorporada ao modelo *Kanban*, apresentado na seção seguinte.

4.3 Kanban

Kanban é um modelo de gerenciamento de produção que antecede em muito outros modelos ágeis para produção de software. Os conceitos do método surgem nos anos 1940, quando a Toyota começou a estudar formas de otimizar a operacionalização de produção em supermercados (Ohno, 1988).

O modelo de gerenciamento *Kanban* é ainda mais simples do que o *Scrum*. Não há necessariamente reuniões e papeis como no *Scrum*. O *Kanban* sugere que você tome o processo que atualmente usa na empresa e comece com ele. Você vai passar a usar o quadro *Kanban* para representar os estados das atividades com o processo atual.

Há duas diferenças fundamentais entre o *Kanban* e o *Scrum*:

- No *Kanban*, não existem necessariamente *sprints* ou iterações, o processo é contínuo e vai produzindo entregáveis à medida que os itens vão ficando prontos.
- No *Kanban* existe um limite de tarefas que podem aparecer em cada coluna. Pode-se por exemplo, limitar o número de atividades em andamento, ou limitar as atividades na fila de espera para teste. As colunas ou mesmo células podem ter limites diferentes umas das outras. A ideia é que se a fila de teste, por exemplo, está cheia, você deve executar algum teste para desafogá-la antes de produzir uma nova tarefa que vai entrar na fila de teste quando pronta.

Além dessas duas diferenças, o *Kanban* também propõe que as atividades sejam priorizadas, sendo que as mais importantes devem ocupar o topo das colunas. Pode-se trabalhar adicionalmente com cartões coloridos sendo que o vermelho indica a prioridade mais alta, o amarelo a média e o verde a mais baixa.

Os quatro princípios fundamentais do *Kanban* são:

- *Comece com o que você faz agora*. A ideia é aproveitar processos que já existem fazendo-os evoluir e melhorar e não promover mudanças muito radicais na forma de trabalho da organização.
- *Concordar em buscar mudanças evolucionárias*. A equipe deve estar engajada no processo de melhoria e a mudança deve ser gradual.

- *Inicialmente, respeite os papeis, responsabilidades e cargos atuais*. O *Kanban* não sugere papeis, como *Scrum* e outros modelos, então os papeis existentes podem ser mantidos inicialmente.
- *Incentivar atos de liderança em todos os níveis*. Também uma forma de manter a equipe engajada e participativa no processo de mudança.

Esses princípios também se traduzem em seis práticas fundamentais de *Kanban*:

- *Visualizar o fluxo de trabalho*. O quadro de *Kanban* permite que se possa visualizar o que há por fazer, o que está sendo feito e o que já foi feito.
- *Limitar a quantidade de trabalho em andamento*. Em *Kanban* frequentemente usa-se a sigla WIP (*work in progress*), que significa "trabalho em andamento". A própria estrutura do quadro, com número máximo de itens por coluna, favorece esta limitação. A ideia é evitar que atividades se acumulem em determinados gargalos e deixem de receber o encaminhamento. Por exemplo, pode haver vinte atividades de desenvolvimento, mas apenas quatro podem ficar na fila de teste. Assim, antes de desenvolver uma quinta atividade, será necessário que pelo menos uma das quatro existentes seja encaminhada para o teste, dessa forma desafogando a fila.
- *Gerenciar e medir o fluxo*. Com a visualização do processo de trabalho fica mais fácil também fazer sua mensuração. Pode-se usar com *Kanban* o *burndown chart*, como no caso de *Scrum*.
- *Tornar as políticas do processo públicas*. Há várias formas de organizar o processo de trabalho com *Kanban*. A vantagem do uso do quadro é que nele esse processo pode ser desenhado e redesenhado de forma que toda equipe possa participar e entender o que está acontecendo na organização.
- *Implementar* loops *de feedback*. Com a possibilidade de mensuração do processo também é possível avaliar a efetividade do processo. Mas não é só isso, também é necessário buscar sempre o *feedback* em relação ao retorno do investimento, ou seja, se as atividades realmente estão gerando valor para os interessados e não apenas sendo executadas.
- *Usar modelos para reconhecer oportunidades de melhoria*. *Kanban* segue o princípio *kaizen*, que significa "melhoria contínua". Assim, aplicar *Kanban* significa estar sempre atento às oportunidades de evolução e melhoria. Um processo estagnado é um processo ruim. *Kanban*, assim como todos os modelos ágeis, propõe que os processos estejam sempre evoluindo.

Um livro de referência para quem pretende usar *Kanban* é o de Anderson (2010), que descreve a evolução do modelo a partir de aplicações reais em projetos da Microsoft desde 2004. Outra boa referência mais recente é Brechner (2015).

4.3.1 Quadros e Cartões *Kanban*

Kanban basicamente significa "sinal" ou "sinalização" em japonês e o modelo se estrutura exatamente sobre um cartaz com cartões do tipo *post-it*, conforme mostrado na Figura 4.11. O *sprint backlog* do *Scrum* é um exemplo de quadro *Kanban*.

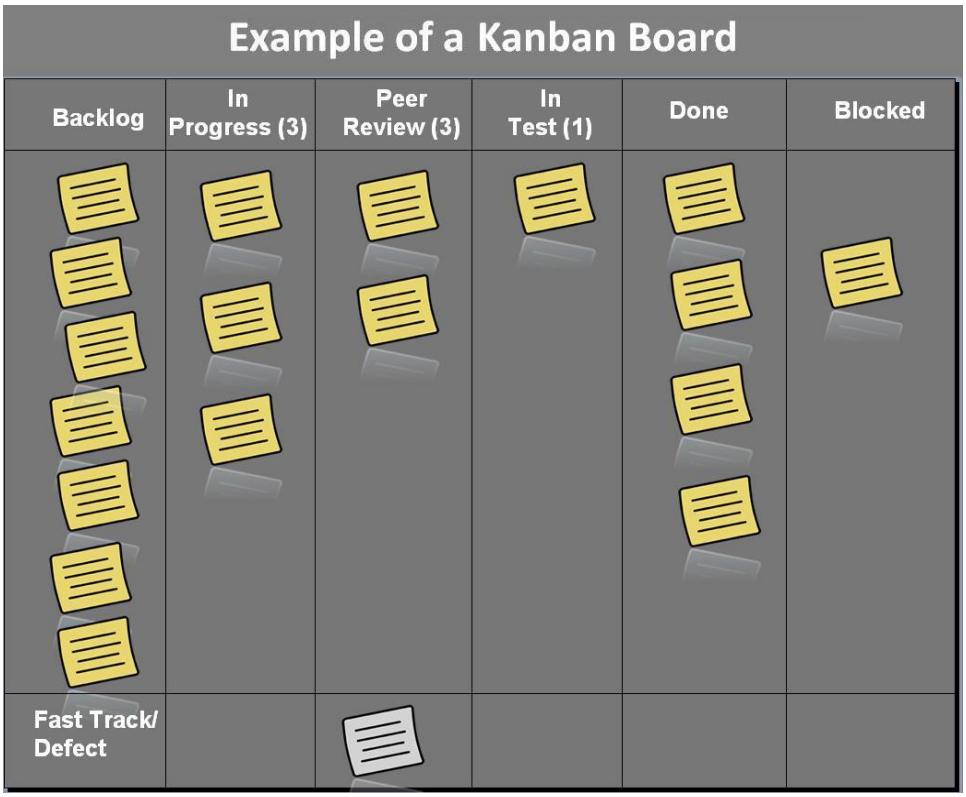

Figura 4.11 Exemplo de quadro *Kanban* (Fonte: Dr ian mitchell - Own work, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=20245783)
TRADUÇÃO:
Example of a Kanban Board//Exemplo de um quadro Kanban
Backlog//Backlog
In Progress//Em andamento
Peer Review//Em revisão
In Test//Em teste
Done//Feito
Bloqued//Bloqueado
Fast Track/Defect//Trilha rápida/defeitos

Usualmente inicia-se com apenas três colunas no quadro *Kanban*: "por fazer", "sendo feito" e "feito". Mas dependendo do processo que a organização adota, outras colunas podem ser adicionadas, como, por exemplo, "em prototipação", "aguardando teste" ou "em homologação".

Em relação aos itens que constam nos cartões nos quadros *Kanban*, pelo menos três tipos podem ser identificados: épicos, histórias de usuário e tarefas. Os *épicos* são os grandes objetivos, que se estruturam através de um conjunto de *histórias de usuário*, as quais por sua vez se estruturam em *tarefas*. Observe que aqui há uma diferença em relação com *Scrum*, que separa em dois quadros: os épicos e histórias de usuário no *product backlog* e as tarefas organizadas por história de usuário no *sprint backlog*.

O quadro *Kanban* também pode ser dividido horizontalmente em raias. A divisão pode atender a qualquer critério que a equipe de desenvolvimento considere útil. Pode-se por exemplo, dividir os fluxos de atividades por épico ou história de usuário. Ou pode-se também dividir por

tipo de tarefa, como por exemplo, tarefas de desenvolvimento, mitigação de risco e correção de *bugs*.

Uma vez que as raias são ortogonais às colunas que definem o estado dos itens, formam-se então células na intersecção de cada raia e cada coluna. Essas células podem ter limites de itens distintos. Por exemplo, pode-se limitar as atividades de desenvolvimento a quatro simultâneas e as atividades de correção de bugs a dez simultâneas.

4.3.2 Scrumban

Pode-se considerar que pela sua simplicidade inicial, *Kanban* é um excelente modelo de entrada para organizações que desejam adotar métodos ágeis. À medida que a organização começa a se acostumar às mudanças na forma de gerenciar a produção, ela pode decidir pela adoção de modelos mais completos, como *Scrum*, por exemplo.

Existe inclusive uma vertente denominada *Scrumban* (Ladas, 2009) que combina as características dos dois modelos. Inicialmente *Scrumban* foi pensado como um modelo intermediário que seria aplicado enquanto uma organização que usa *Scrum* procurava explorar a filosofia *Kanban*, mas hoje é considerado um modelo a parte onde o gerenciamento do processo se dá de acordo com as regras de *Scrum*, mas a visão de melhoria do processo ocorre de acordo com a filosofia *Kanban*.

Assim, *Scrumban* terá as reuniões previstas por *Scrum*, mas também terá as limitações de itens nos quadros, conforme proposto por *Kanban*.

Pode-se argumentar que *Scrum* é mais adequado para desenvolvimento de produtos, quando ao final de cada *sprint* um incremento deve ser entregue, enquanto *Kanban* é mais adequado para oferta de serviços, como por exemplo, manutenção de produtos de software, quando as demandas são contínuas e não organizadas em *sprints*. Como *Scrumban* procura manter o aspecto de entregas contínuas de *Kanban*, então também se pode concluir que sua aplicação primordial se dá em situações de manutenção, onde o trabalho é dirigido pela demanda.

Algumas das principais diferenças entre *Scrumban*, *Scrum* e *Kanban* são listadas na Tabela 4.3.

Tabela 4.3 Diferenças entre *Kanban*, *Scrum* e *Scrumban*

Aspecto	**Kanban**	**Scrum**	**Scrumban**
Papeis	Não há	*Product owner*, *Scrum master* e *development team*	A equipe e quaisquer papeis que se façam necessários
Reuniões diárias	Não há	*Daily meeting*	Usadas para garantir o trabalho contínuo e reduzir o tempo de ociosidade dos membros da equipe
Review e retrospective	Não há	As duas reuniões são prescritas no final das *sprints*	Podem ser feitas na medida do necessário para aprimorar o conhecimento e o processo
Fluxo de trabalho	Contínuo	*Sprints*	Contínuo, mas iterações regidas por *timeboxing* podem ser usadas eventualmente
Artefatos	Quadro *Kanban*	*Product backlog* e *sprint backlog*	Quadro *Kanban*

Equipes	Não menciona	Equipes multifuncionais focadas em requisitos	Equipes podem ser especializadas tecnicamente
WIP	Controlada pelo estado do quadro *Kanban*	Controlada pelo conteúdo da *sprint*	Controlada pelo estado do quadro *Kanban*

Assim, *Scrumban* adiciona ao modelo *Scrum* a limitação do WIP (trabalho em andamento) originária de *Kanban*. Isso dá mais controle às atividades realizadas pela equipe, eliminando um possível *stress* ocasionado pela aproximação dos prazos fatais de uma *sprint* e ao mesmo tempo garantindo que o fluxo de trabalho seja contínuo, com itens evoluindo da coluna "por fazer" para a coluna "feito" de forma organizada e sem acumular tarefas nas colunas intermediárias.

4.4 XP – *eXtreme Programming*

Programação Extrema, ou XP (*eXtreme Programming*), é um modelo ágil, inicialmente adequado a equipes pequenas, baseado em uma série de valores, princípios e regras. O XP surgiu nos Estados Unidos no final da década de 1990. Seu criador foi Kent Beck, que descreveu em um livro (Beck, 1999) sua experiência na gerência de um projeto que procurava analisar as melhores formas de aplicar a tecnologia de orientação a objetos em projetos, usando como objeto de estudo sistemas de folha de pagamento da Chrysler.

Entre os principais valores de XP podemos citar:

- *Simplicidade*: Muitas vezes, analistas generosos incluem requisitos que os clientes não necessariamente pediram. XP sugere como valor a simplicidade, ou seja, a equipe deve se concentrar nas funcionalidades efetivamente necessárias, e não naquelas que *poderiam* ser necessárias, mas de cuja necessidade real ainda não se tem evidência.
- *Respeito*: respeito entre os membros da equipe, assim como entre a equipe e o cliente, é um valor dos mais básicos, que dá sustentação a todos os outros. Se não houver respeito, a comunicação falha e o projeto afunda.
- *Comunicação*: em desenvolvimento de software, a comunicação é essencial para que o cliente consiga dizer aquilo de que realmente precisa. O XP preconiza comunicação de boa qualidade, preferindo encontros presenciais em vez de videoconferências, videoconferências em vez de telefonemas, telefonemas em vez de *e-mails*, e assim por diante. Ou seja, quanto mais pessoal e expressiva a forma de comunicação, melhor.
- *Feedback*: o projeto de software é reconhecidamente um empreendimento de alto risco. Cientes disso, os desenvolvedores devem buscar obter *feedback* o quanto antes para que eventuais falhas de comunicação sejam corrigidas o mais rapidamente possível, antes que os danos se alastrem e o custo da correção seja alto.
- *Coragem*: pode-se dizer que a única coisa constante no projeto de um software é a necessidade de mudança. Para os desenvolvedores XP, é necessário confiar nos mecanismos de gerenciamento da mudança para ter coragem de abraçar as inevitáveis modificações em vez de simplesmente ignorá-las por estarem fora do contrato formal ou por serem muito difíceis de acomodar.

A partir desses valores, uma série de princípios básicos é definida:

- *Feedback* rápido.
- Presumir simplicidade.

- Mudanças incrementais.
- Abraçar mudanças.
- Trabalho de alta qualidade.

Então, o XP preconiza mudanças incrementais e *feedback* rápido, além de considerar a mudança algo positivo, que deve ser entendido como parte do processo. Além disso, o XP valoriza o aspecto da qualidade, pois considera que pequenos ganhos a curto prazo pelo sacrifício da qualidade não são compensados pelas perdas a médio e a longo prazo.

A esses princípios pode-se adicionar ainda a *priorização de funcionalidades mais importantes*, de forma que, se o trabalho não puder ser todo concluído, pelo menos as partes mais importantes terão sido. Segue-se neste caso o Princípio de Pareto, ou Princípio 80/20, que sugere que os 20% mais importantes dentre os requisitos produzem cerca de 80% da funcionalidade realmente necessária.

4.4.1 Práticas XP

Para aplicar XP, é necessário seguir uma série de práticas que dizem respeito ao relacionamento com o cliente, a gerência do projeto, a programação e os testes. Considera-se, via de regra, a existência de doze práticas principais que são detalhadas nas subseções a seguir. Essas práticas podem ser classificadas em quatro grupos principais:

- *Feedback*: incluindo as práticas de (1) jogo de planejamento, (2) programação em pares, (3) desenvolvimento dirigido por teste e (4) equipe coesa.
- *Processo contínuo*: incluindo as práticas de (5) integração contínua, (6) entregas pequenas, e (7) excelência técnica.
- *Compreensão compartilhada*: incluindo as práticas de (8) metáfora de sistema, (9) padrões de codificação, (10) design simples e (11) posse coletiva do código.
- *Bem-estar dos desenvolvedores*: incluindo a prática de (12) ritmo sustentável.

As práticas XP não são consenso entre os desenvolvedores e pesquisadores. Keefer (2003) afirma, entre outras coisas, que as práticas nem sempre são aplicáveis, que o estilo de trabalho não é escalável para equipes maiores e que a programação em pares acaba sendo uma atividade altamente cansativa, que só é praticável se sua duração for mantida em períodos de tempo relativamente curtos.

Além disso, Tolfo e Wazlawick (2008) demonstram que a cultura organizacional, em especial seus aspectos mais profundos, que são os valores assumidos e praticados pelas pessoas independentemente do que esteja escrito nos quadros de missão e visão da empresa, é determinante no sentido de oferecer um ambiente fértil ou hostil à implementação de métodos ágeis, em especial o XP.

4.4.1.1 Jogo de Planejamento

A prática conhecida como *jogo de planejamento* (*planning game*) estabelece que semanalmente, a equipe deve se reunir com o cliente para priorizar as funcionalidades a serem desenvolvidas. Cabe ao cliente identificar as principais necessidades e à equipe de desenvolvimento estimar quais podem ser implementadas na iteração que se inicia. Ao final da iteração, essas funcionalidades são entregues ao cliente. Esse tipo de modelo de relacionamento com o cliente é adaptativo, em oposição aos contratos rígidos usualmente estabelecidos.

O jogo de planejamento é uma reunião que pode ser comparada ao *sprint planning meeting* do *Scrum*. Porém, sua estrutura é mais detalhada. A reunião usualmente tem duas partes. A

primeira parte consiste no *planejamento de entregas*, onde se vai procurar prever quais histórias de usuário serão entregues nas iterações mais próximas a seguir. É muito importante que o usuário ou cliente esteja presente nesta reunião. A segunda parte usualmente consiste no *planejamento da iteração*, onde as atividades dos desenvolvedores ao longo da próxima iteração serão planejadas. Nessa reunião a participação do cliente ou usuário não é necessária.

Tanto o planejamento de entregas quanto o planejamento da iteração se dividem em três fases: exploração (*exploration*), concordância (*agreement*) e direção (*steering*).

Na fase de *exploração no planejamento de entregas* inicialmente o cliente/usuário é convidado a escrever uma ou mais histórias de usuário. Essas histórias devem ser feitas por ele sem que haja sugestões ou interferência dos desenvolvedores. Cada história de usuário é registrada em um cartão. Em seguida, os desenvolvedores poderão analisar cada história e fazer sua estimação de esforço em pontos de história. Se necessário, os desenvolvedores poderão também propor *spikes*, que são protótipos que visam reduzir algum risco tecnológico ou de requisitos explorando o problema de forma pontual. Usualmente, *spikes* são código descartável que, depois de auxiliar na compreensão de um risco, são jogados fora. Opcionalmente, também, histórias de usuário que ficaram tão complexas ou vagas que impeçam a equipe de fazer estimativas confiáveis, devem ser divididas em histórias menores, para que possam ser efetivamente analisadas.

Na fase de *concordância no planejamento de entregas* são determinados os custos, benefícios e impacto no cronograma de cada uma das histórias. Para isso, o cliente deve separar as histórias de usuário em três grupos de acordo com o valor para o negócio: *histórias críticas*, sem as quais o negócio não funciona ou não faz sentido, *histórias significativas*, que não são críticas, mas consideradas importantes por agregar valor ao negócio e *histórias que poderia ser bom ter*, que são aquelas sem valor significativo para o negócio, mas que mesmo assim o cliente deseja que sejam eventualmente consideradas para inclusão.

Em seguida, ainda na mesma fase, os desenvolvedores devem classificar as histórias por risco, segundo a escala alto/médio/baixo. Mais detalhes sobre análise de risco são apresentados no Capítulo 8, mas por hora pode-se mencionar uma abordagem simples para classificação de riscos baseada na complexidade, volatilidade e completeza da história. As perguntas a serem feitas são: a história de usuário será de difícil implementação? É provável que a descrição dessa história mude à medida que procedermos à análise e desenvolvimento? Existem detalhes da história de usuário que não são conhecidos? Quanto mais positiva for a resposta a estas perguntas, maior o risco que a história de usuário encerra.

Na sequência da fase, caso a equipe ainda não tenha feito isso em reuniões anteriores, ela deverá estabelecer sua capacidade de trabalho (número de desenvolvedores e duração das iterações) e, assim, alocar as histórias de usuário em função de seu valor para o cliente e seu risco no planejamento das iterações.

A fase de *direção no planejamento de entregas* inicia com a revisão do planejamento feito por todos e eventual correção de rumos. Mas esta fase continua ao longo da iteração. À medida que o desenvolvimento avança, correções no plano de entregas podem e devem ser feitas. Usualmente a necessidade dessas correções são detectadas pela equipe e negociadas com o cliente. O nome "direção" vem da comparação com o processo de dirigir um automóvel em uma estrada, no qual, ao longo do percurso, correções são feitas para manter o veículo no traçado.

Passando agora para o planejamento da iteração, as fases são as mesmas, mas o que ocorre nelas é diferente. Na fase de *exploração no planejamento da iteração* os desenvolvedores vão analisar cada história de usuário e identificar as tarefas de desenvolvimento necessárias para

transformá-las em produto entregável. Essas tarefas são anotadas em cartões de tarefa e sua duração deve ser estimada em pontos de história. Se as tarefas forem longas demais para uma estimação confiável, elas devem ser subdivididas. Por outro lado, se forem curtas demais (menos de um ponto de história), devem ser combinadas.

Pode-se usar neste momento a técnica de *planning poker*, na qual, após uma descrição e debate sobre a tarefa e sua complexidade, cada membro da equipe vai escrever o valor em pontos de história que ele estima para esta tarefa. Cada um escreve um valor e o esconde de forma que ninguém saiba quanto cada um estimou. Quando todos concluírem, os valores são abertos. Caso todos os valores sejam muito próximos, chegou-se a um consenso. Caso haja valores discrepantes, aquele que propôs o valor mais baixo e aquele que propôs o valor mais alto são convidados a explicar suas razões. Após o debate, repete-se a estimativa, com cada membro escrevendo um valor em um papel, sem o conhecimento dos outros. Abre-se novamente os valores e pode-se então ter chegado a um novo consenso ou continuar a discrepância. Caso a discrepância continue, pode-se repetir as rodadas até chegar ao consenso ou aplicar a fórmula $E = (E_{pessimista} + E_{otimista} + 4 \times E_{médio})/6$, que estabelece o esforço em pontos de história como a média ponderada na qual o otimista e pessimista entram com peso 1 cada e a média dos demais com peso 4.

Na fase de *concordância do planejamento da iteração* as tarefas serão distribuídas entre os programadores. Neste momento, os programadores podem revisar as estimativas de tempo de acordo com suas próprias expectativas. Deve-se buscar uma distribuição uniforme de esforço entre os desenvolvedores, de forma que nenhum fique sobrecarregado ou ocioso. Além disso, cada desenvolvedor deve realisticamente estimar quantas horas produtivas ele efetivamente tem ao longo da iteração; horas gastas em reuniões e outras atividades que não sejam relacionadas ao desenvolvimento do produto não devem ser consideradas. Assim, as atividades que ele alocar para si devem ter um esforço estimado compatível com as horas que ele espera ter disponíveis na iteração.

Observe que esta atividade de distribuição das tarefas entre os desenvolvedores diferencia o XP de *Scrum* e *Kanban*, sendo que XP propõe uma distribuição prévia das atividades entre eles enquanto que os outros modelos mencionados atribuem uma única atividade a cada desenvolvedor inicialmente, atribuindo outras de forma contínua à medida que estas vão sendo concluídas.

A fase de *direção no planejamento de iteração* também ocorre ao longo da iteração. Nela, um desenvolvedor vai pegar um dos cartões de atividade atribuídos a ele, juntamente com um colega que será seu par (não precisa ser necessariamente o mesmo sempre). Se necessário, o par de desenvolvedores vai fazer o *design* da tarefa, criando modelos, diagramas e descrições que forem necessários para um entendimento o mais perfeito possível do trabalho a ser feito. Na sequência, a tarefa é desenvolvida usando-se a técnica de desenvolvimento dirigido por teste (Seção 13.6). Finalmente, o par fará o teste de sistema, que consiste em avaliar se o produto desenvolvido atende aos requisitos, especificados na história de usuário e quaisquer outros meios.

4.4.1.2 Programação em Pares

A prática de *programação em pares* (*pair programming*) propõe que a programação seja sempre feita por duas pessoas em cada computador. Enquanto um (o motorista ou *driver*) usa o computador para escrever programas o outro (navegador ou *navigator*) deve ajudar corrigindo rumos e sugerindo melhores práticas de codificação. Com isso, o código gerado terá sempre sido verificado por pelo menos duas pessoas, reduzindo drasticamente a possibilidade de erros.

Estudos indicam que usualmente a prática de programação em pares aumenta o esforço. de desenvolvimento, pois duas pessoas realizam o trabalho que usualmente apenas uma faria. Porém, nem sempre este aumento é de 100% como seria de esperar, havendo situações em que apenas 15% de aumento é observado. Porém, a principal razão a favor desta prática reside no fato de que o código produzido é frequentemente de qualidade muito superior ao que uma única pessoa teria feito; isso porque o código já estará revisado por duas pessoas e experiências são trocadas entre os membros da equipe com muito mais facilidade quando esta prática é bem conduzida (Cockburn & Williams, 2001).

Em relação às combinações de pares entre programadores experientes ou iniciantes, outro estudo (Lui & Chan, 2005) apresenta os seguintes resultados:

- *Experiente* × *experiente*: este seria um cenário ideal, com duas pessoas experientes trocando experiências; porém, na prática acaba não criando muitas oportunidades de inovação, pois os dois acabam não questionando práticas bem estabelecidas.
- *Novato* × *novato*: este cenário apresenta resultados muito melhores do que dois novatos trabalhando sozinhos. Porém, não é também o mais indicado, pois novatos poderão demorar muito a desenvolver o mesmo nível de habilidade de um experiente. Assim, eles vão acabar reinventando a roda.
- *Experiente* × *novato*: este acaba sendo o melhor cenário para ambos, pois o experiente poderá orientar e corrigir o novato e ao mesmo tempo o novato poderá questionar as práticas estabelecidas pelo experiente, abrindo caminho para possíveis inovações.

Um risco que se corre com a abordagem de colocar um novato junto com um experiente é que o novato poderá assumir uma atitude passiva, apenas ouvindo e aceitando as opiniões do experiente. Este fenômeno, conhecido como "observe o mestre" deve ser desencorajado.

Outro risco que ocorre em qualquer dos cenários é o desligamento (*shut-down*) de um dos membros do par. Assim, quando o motorista monopoliza o trabalho e efetivamente não aceita as contribuições do navegador, o navegador poderá facilmente interessar-se mais pelas redes sociais, uma garrafa de café ou mesmo o balançar das folhas das árvores pela janela, desligando-se assim do trabalho a ser feito.

4.4.1.3 Desenvolvimento Dirigido por Teste

A prática de *desenvolvimento dirigido por teste* (*test-driven development*), se consolidou como uma boa prática em toda a indústria de software, e não apenas nos projetos orientados por XP. A rigor, a ideia é desenvolver o código de teste de uma função antes de implementar a função. Além disso, quando o comportamento de uma função precisa ser alterado, o procedimento consiste em primeiro alterar o conjunto de testes para abranger o novo comportamento e só depois proceder às modificações na função.

Na Seção 13.6 são apresentadas mais informações sobre esta importante prática de desenvolvimento de software. Sua principal motivação reside no fato de fazer os programadores pensarem sobre a funcionalidade antes de começarem a escrever o código que vai implementar a funcionalidade.

4.4.1.4 Equipe Coesa

A prática de *equipe coesa* (*whole team*) considera que o cliente faz parte da equipe de desenvolvimento e a equipe deve ser estruturada de forma que eventuais barreiras de comunicação sejam eliminadas.

Destaca-se que na literatura XP o cliente não é necessariamente aquele que paga pelo sistema, mas aquele que usa o sistema. Outras literaturas poderão chamar de *cliente* aquele que paga e *usuário* aquele que usa. Assim, convém ter em mente o verdadeiro significado da palavra *cliente* no ambiente XP.

Na Seção 4.6.6, no contexto do modelo *Crystal Clear* a discussão sobre esta prática é um pouco mais aprofundada.

4.4.1.5 Integração Contínua

A prática de *integração contínua* (*continuous integration*) estabelece que não se deve esperar até o final de uma iteração para integrar uma nova funcionalidade que acaba de ser desenvolvida. Assim que estiver viável, ela deverá ser integrada ao sistema e testada para evitar surpresas mais tarde.

Há modelos como Cascata com Subprojetos que preveem uma integração do tipo *big-bang*, onde vários módulos de um sistema desenvolvidos separadamente passam por uma longa fase de integração. Essa fase acaba sendo longa porque usualmente fazer essa integração faz surgir muitos erros de compatibilidade entre módulos que precisam ser detectados e consertados.

Outros modelos, como RUP, preveem integrações ao final de cada iteração. Isso reduz o problema do Cascata com Subprojetos, mas segundo a filosofia XP, ainda não é bom o suficiente. Assim, com XP integrações poderão ocorrer a qualquer momento, e serão feitas não por um especialista em integrações, mas pelo mesmo par de programadores que desenvolveu o incremento que está sendo considerado maduro o suficiente para integrar a versão completa do sistema.

4.4.1.6 Entregas Pequenas

A prática de *entregas pequenas* (*small releases*) propõe que a liberação de pequenos incrementos do sistema pode ajudar o cliente a testar as funcionalidades de forma contínua. O XP leva esse princípio ao extremo, sugerindo versões ainda menores do que as de outros processos incrementais, como RUP e *Scrum*.

Enquanto esses modelos preveem incrementos gerados em períodos mínimos de duas semanas, XP recomenda como padrão a geração de incrementos uma vez por semana, abrindo-se exceções em casos de maior complexidade para duas e nunca mais do que três semanas. Enquanto isso, Scrum e RUP consideram aceitáveis iterações de até oito semanas, dependendo das características da equipe e do projeto.

4.4.1.7 Excelência Técnica

A prática de *excelência técnica* (*design improvement*) estabelece que não basta o código funcionar, ele tem que ser claro e elegante. Porém, outra prática que é a de *design* simples estabelece que não se deve implementar agora aquilo que ainda não é necessário. Essa filosofia de fazer o mínimo necessário, por vezes, leva a um *design* simplista que precisa ser arrumado.

Design simplista pode incluir código repetido, alta coesão entre classes e baixo acoplamento, o que compromete a manutenção desse código.

Assim, a prática de excelência técnica complementa a de *design* simples estabelecendo que quando se detecta esse tipo de falhas no *design* deve-se proceder à refatoração do código. Ela permite manter a complexidade do código em um nível gerenciável, além de ser um investimento que traz benefícios em médio e longo prazo.

Usualmente, quem trabalha com modelos ágeis e segue esta prática tem como lema "refatore sem pena".

4.4.1.8 Metáfora de Sistema

A prática de *metáfora de sistema* (*system metaphor*) estabelece que é preciso conhecer a linguagem do cliente e seus significados. A equipe deve aprender a se comunicar com o cliente na linguagem que ele compreende.

Além disso, a equipe deve manter entre si essa mesma linguagem; nomes de classes, funções, variáveis etc. devem significar algo do ponto de vista do negócio, para que quem for ler o código ou documentação saiba imediatamente de que conceito de negócio se trata. Foi-se o tempo em que Fortran permitia nomes de variáveis com apenas dois caracteres. Assim, deve-se evitar usar *a*, *b*, *c*, *x*1 e *x*2 como nomes de variáveis e preferir palavras ou expressões mais significativas.

4.4.1.9 Padrões de Codificação

A prática de *padrões de codificação* (*coding standards*) sugere que a equipe deve estabelecer e seguir padrões de codificação, de forma que o código pareça ter sido todo desenvolvido pela mesma pessoa, mesmo que tenha sido feito por dezenas delas.

Nomes de variáveis, funções, classes, uso de *snake_case* ou *CamelCase*, espaço de endentação e até a escolha de certas estruturas de programação em detrimento de outras podem ser definidos por padrões. Aplica-se os padrões nas situações em que há mais de uma forma de escrever o mesmo código.

Exemplos de padrões gerais de linguagens são o *Java Code Conventions* de 1997 (ver primeiro *QR code*) [QRC 4.5] e o PEP 8 – *Style Guide for Python Code* (ver segundo *QR code*) [QRC 4.6], ambos gratuitamente disponíveis na Internet.

Embora seja recomendado que as equipes sigam os padrões mais aceitos da linguagem que utilizam, sem alterar suas regras, elas podem e devem, sempre que necessário, criar novas regras nas situações emque sintam necessidade e as regras gerais sejam omissas.

O objetivo final é que o código seja auto documentado e de entendimento relativamente fácil, se comparado a código obscuro ou fora de padrão.

4.4.1.10 Design Simples

Design simples (*simple design*) implica em atender à funcionalidade solicitada pelo cliente sem sofisticar desnecessariamente. Deve-se fazer aquilo de que o cliente precisa, não o que o desenvolvedor gostaria que ele precisasse. Por vezes, *design* simples pode ser confundido com *design* fácil. Nem sempre o *design* simples é o mais fácil de se implementar, e o *design* fácil pode não atender às necessidades ou então gerar problemas de arquitetura.

Por exemplo, um *design* pode estar demasiadamente complexo por ter sido feito sem cuidado ou por ter sido alterado várias vezes. Neste caso, será mais fácil deixá-lo como está, mas a prática de *design* simples exige que ele seja refatorado para que sua complexidade interna seja simplificada.

De acordo com essa prática, quando um desenvolvedor for implementar uma funcionalidade ele sempre deve se perguntar se há alguma maneira mais simples de implementá-la, e se houver, deve fazer desta forma. Além disso, a prática de programação dirigida por teste reforça este

princípio ao estabelecer que uma vez que os testes tenham sido definidos, deve-se implementar o código mais simples possível que passe nos testes, sem nenhuma outra característica.

4.4.1.11 Posse Coletiva do Código

A prática de *posse coletiva do código* (*collective code ownership*) estabelece que o código não tem dono e não é necessário pedir permissão a ninguém para modificá-lo.

Outros modelos, como FDD, por exemplo, que também é considerado ágil, estabelecem o contrário: cada classe tem um dono e um desenvolvedor só pode modificá-la com permissão do dono.

Segundo XP, porém, isso pode criar gargalos no processo de desenvolvimento. Não só gargalos de permissão (vá que o dono da classe esteja de férias) como também gargalos de conhecimento, já que neste caso, programadores se especializam em suas classes e pouca colaboração acaba acontecendo entre eles.

Essa prática, porém, não pode ser estabelecida simplesmente como um bom princípio a ser buscado; ela precisa ser suportada por tecnologia adequada e outras práticas. Para funcionar, a prática de posse coletiva do código precisa de um bom sistema de controle de versões, teste automatizado e integração contínua de código. Caso contrário ela pode se tornar um problema para os desenvolvedores.

4.4.1.12 Ritmo Sustentável

A prática de *ritmo sustentável* (*sustainable pace*) indica que a equipe deve trabalhar com qualidade um número razoável de horas por dia (não mais de oito). Trabalhar horas extras não é proibido, mas também não é recomendado. Se em uma semana houver horas extras, recomenda-se que na outra não haja.

Esta prática se baseia na constatação de que equipes cansadas são pouco produtivas. Assim, se uma equipe se acostuma a fazer horas extras, ela passa a produzir menos durante todo o dia e, no fim, essas horas a mais nada ou muito pouco adicionam à produtividade da equipe.

A viabilidade desta prática ocorre pelo fato de que o XP se estrutura em ciclos curtos de entrega. Assim, é pouco provável que em entregas curtas de pequenos incrementos do produto exista a necessidade de colocar a equipe em horas extras para terminar as tarefas. Já em equipes que fazem entregas de grandes pacotes em períodos de tempo mais longos (vários meses) podem sofrer dessa síndrome do *deadline*, que ocorre quando a equipe vê o prazo se aproximar e ainda há muito trabalho a ser feito.

4.4.2 Regras XP

Auer, Meade e Reeves (2003) argumentam que XP é um modelo que não se define nem pelos seus valores nem pelas suas práticas. Os valores de XP são os mesmos do manifesto ágil; assim, eles não são suficientes para diferenciar XP de *Scrum*, por exemplo. Já as práticas XP são algumas vezes muito vagas. Por exemplo, o significado que "equipe coesa" para uma equipe pode ser algo bem diferente do que é para outra equipe; em um caso, pode-se ter o cliente efetivamente participando do trabalho na mesma sala diariamente, e em outro caso o cliente poderia estar acessível por telefone e fazer algumas reuniões presenciais na semana.

Assim, segundo os autores, o que define XP são suas regras. As subseções 4.4.2.1 e 4.4.2.2 procuram resumir as regras propostas por eles, as quais se dividem em dois grupos: *regras de engajamento* e *regras de funcionamento*. Já as subseções seguintes apresentam o resumo das regras XP elaboradas por outro autor (Wells, 2009 – acessível pelo *QR code* ao lado), [QRC 4.7]

que as categoriza em *regras de planejamento*, de *gerenciamento*, de *design*, de *codificação* e de *teste*.

4.4.2.1 Regras de Engajamento

Auer, Meade e Reeves (2003) explicam que as *regras de engajamento* definem as condições iniciais para que uma equipe trabalhe com XP. Fazendo um paralelo com basebol, as regras de engajamento iriam definir o tamanho e forma do campo, o número de jogadores na equipe etc. Já as regras de funcionamento, apresentadas na seção seguinte, definem como o jogo deve transcorrer. Assim, as regras de engajamento propostas por eles são as seguintes:

- Uma equipe XP consiste em um grupo de pessoas que se reúne para desenvolver um produto de software. Esse produto pode ou não ser parte de um produto maior. Pode haver diversos papeis na equipe, mas pelo menos dois são necessários: o cliente e o desenvolvedor.
- O cliente deve definir e continuamente ajustar os objetivos e prioridades, baseando-se em informações fornecidas pelos desenvolvedores e outros participantes. Objetivos são definidos em termos de *o quê* e não de *como*.
- O cliente está sempre disponível e fornece informações sob demanda para ajudar os desenvolvedores a formar estimativas ou fornecer os produtos necessários. O cliente é uma parte integrante da equipe.
- Em qualquer momento, qualquer membro da equipe deve ser capaz de mensurar o progresso da equipe na direção dos objetivos do cliente.
- A equipe deve atuar como uma *rede social efetiva*, o que significa que deve haver comunicação honesta que leva à aprendizagem contínua, grau mínimo de separação entre o que é necessário para que a equipe faça progresso e as pessoas ou recursos que podem atender essas necessidades e alinhamento de autoridade e responsabilidade.
- *Timeboxing* é usado para identificar segmentos de esforço de desenvolvimento e cada segmento não deve durar mais de um mês.

Muitas destas regras, certamente também são compartilhadas por outros modelos ágeis com maior ou menor rigor.

4.4.2.2 Regras de Funcionamento

As regras de funcionamento de XP indicam como o modelo deve funcionar na prática uma vez que a estrutura esteja montada. Em resumo, as regras são as seguintes:

- O trabalho produzido deve ser continuamente validado pelo teste.
- Escreva os testes de unidade antes do código, programe em pares e refatore o código para atender aos padrões de codificação enquanto estiver trabalhando nas prioridades do cliente.
- Todo o código escrito para uso potencial no produto deve passar em todos os testes de unidade, expressar claramente cada conceito, não conter duplicação e não conter partes supérfluas.
- A posse do código é coletiva. Todos os membros da equipe têm autoridade para modificar o código e, para cada tarefa, ao menos duas pessoas têm o entendimento necessário realizá-la.

Estas regras, especialmente as duas últimas, diferenciam XP de outros modelos ágeis. A rigor, dentre os principais modelos, explicados neste capítulo, apenas XP adota a programação em pares como um princípio fundamental. Essa prática, como já foi comentado, pode não ser unanimidade, mas ela de fato elimina o gargalo do *especialista* na equipe, ou seja, aquela pessoa que não pode ser desligada porque só ela sabe como trabalhar com determinado trecho do código. Com XP sempre haverá dois especialistas no mínimo para cada parte do código.

4.4.2.3 Regras de Planejamento

Wells (2009) também vai além das práticas XP ao apontar um conjunto extenso e bastante objetivo de regras para XP. Ele divide as regras em cinco grandes grupos: planejamento, gerência, *design*, codificação e teste. Nessa subseção e nas seguintes, essas regras são apresentadas.

O *planejamento* é composto pelas atividades que ocorrem antes do início de um projeto ou iteração. Durante o planejamento, a equipe analisa o problema, seus riscos e alternativas, prioriza atividades e planeja como o desenvolvimento e as entregas vão acontecer. As regras de *planejamento* são as seguintes:

- *Escrever histórias de usuário*: elas são usadas no lugar do documento de requisitos. Ao contrário dos casos de uso, que são definidos pelos analistas, as histórias de usuário devem ser escritas pelos próprios usuários, considerando-se que elas definem itens prioritários que eles precisam para o sistema. Elas podem ser usadas para definir os testes de aceitação (Seção 13.2.4). A equipe deve estimar se a história pode ser implementada em uma, duas ou três semanas. Tempos maiores do que esses significam que a história deve ser subdividida em duas ou mais histórias. Menos de uma semana significa que a história está em um nível de detalhamento muito alto e precisa ser combinada com outras. Como as histórias são escritas pelo cliente, espera-se que não sejam contaminadas com aspectos técnicos.
- *O planejamento de entregas cria o cronograma de entregas*: é feita uma reunião de planejamento de entregas para delinear o projeto como um todo. É importante que os técnicos tomem as decisões técnicas e o cliente tome as decisões de negócio. Deve-se estimar o tempo de cada história de usuário em termos de semanas de programação ideais (uma semana de programação ideal é aquela em que uma pessoa trabalha todas as horas da semana unicamente em um projeto, dedicando-se apenas a ele) e priorizar as histórias mais importantes do ponto de vista do cliente. Essa priorização pode ser feita com histórias impressas em cartões, que devem ser movidos na mesa ou num quadro para indicar as prioridades. As histórias são agrupadas em iterações, que só são planejadas pouco antes de serem iniciadas.
- *Faça entregas pequenas frequentes*: entregas de funcionalidades completas e usáveis devem ocorrer com frequência. Algumas equipes entregam software diariamente, o que pode ser um exagero. No pior dos casos, as entregas deveriam acontecer a cada uma ou duas semanas. A decisão de colocar a entrega em operação ou não é do cliente.
- *O projeto é dividido em iterações*: prefira iterações de uma a duas semanas. Não planeje as atividades com muita antecedência; deixe para planejá-las pouco antes de elas se iniciarem. Planejamento *just-in-time* é uma forma de estar sempre sintonizado com as mudanças de requisitos e arquitetura. Não tente implementar coisas que virão depois. Leve os prazos a sério. Acompanhe a produtividade. Se perceber que não vai vencer o cronograma, convoque uma nova reunião de planejamento de entregas e repasse algumas

entregas para outros ciclos. Concentre-se em completar as tarefas em vez de deixar várias coisas inacabadas.

- *O planejamento da iteração inicia cada iteração*: no planejamento que inicia cada iteração, selecionam-se as histórias de usuário mais importantes a serem desenvolvidas e partes de sistema que falharam em testes de aceitação, as quais são quebradas em *tarefas de programação*. As tarefas serão escritas em cartões, assim como as histórias de usuário. Enquanto as histórias de usuário estão na linguagem do cliente, as tarefas de programação estão na linguagem dos desenvolvedores. Cada desenvolvedor que seleciona uma tarefa estima quanto tempo ela demorará a ser concluída. Tarefas devem ser estimadas em um, dois ou três dias ideais de programação. Tarefas mais curtas que um dia devem ser combinadas, e tarefas mais longas do que três dias devem ser divididas.

4.4.2.4 Regras de Gerenciamento

O *gerenciamento* do projeto ocorre durante sua execução. O gerenciamento busca, basicamente, garantir que as atividades sejam realizadas no prazo, dentro do orçamento e com a qualidade desejada. As regras de *gerenciamento XP* são:

- *Dê à equipe um espaço de trabalho aberto e dedicado*: é importante eliminar barreiras físicas entre os membros da equipe para melhorar a comunicação. Sugere-se colocar os computadores em um espaço central para a programação em pares e mesas nas laterais da sala para que pessoas que precisam trabalhar a sós não se desconectem do ambiente. Inclua uma área para as reuniões em pé, com um quadro branco e uma mesa de reuniões.
- *Defina uma jornada sustentável*: trabalhar além da jornada normal é desgastante e desmoralizante. Se o projeto atrasar, é melhor reprogramar as tarefas em uma *release planning meeting*. Descubra a velocidade ideal para sua equipe e atenha-se a ela. Não tente fazer uma equipe trabalhar na velocidade de outra. Faça planos realistas.
- *Inicie cada dia com uma reunião em pé*: em uma reunião típica, nem sempre todos contribuem, mas pelo menos ouvem. Mantenha o mínimo de pessoas o mínimo de tempo em reuniões. As reuniões em pé não são perda de tempo, mas uma forma rápida de manter a equipe sincronizada, pois cada um dirá o que fez ontem, o que vai fazer hoje e o que o impede de prosseguir.
- *A velocidade do projeto é medida*: conta-se ou estima-se quantos pontos de histórias de usuário e/ou tarefas de programação são desenvolvidos em cada iteração. A cada encontro de planejamento de iteração, os clientes podem selecionar um conjunto de histórias de usuário cujo número total de pontos seja aproximadamente igual à estimativa de velocidade do projeto. O mesmo vale para os programadores em relação às tarefas de programação. Isso permite que a equipe se recupere de eventuais iterações difíceis. Aumentos e diminuições de velocidade são esperados.
- *Mova as pessoas*: a mobilidade de pessoas em projetos é importante para evitar a perda de conhecimento e gargalos de programação. Ficar na dependência de um único funcionário é perigoso. Deve-se evitar criar ilhas de conhecimento, porque elas são suscetíveis a perda. Se precisar de conhecimento especializado, contrate um consultor por prazo determinado.
- *Conserte XP quando for inadequado*: não hesite em mudar aquilo que não funciona em XP. Isso não significa que se pode fazer qualquer coisa. As regras devem ser seguidas até que se perceba que elas precisam ser mudadas. Todos os desenvolvedores devem saber o

que se espera deles e o que eles podem esperar dos outros. A existência de regras é a melhor forma de garantir isso.

4.4.2.5 Regras de *Design*

Em desenvolvimento de software, *design* pode ser uma palavra com vários significados indo desde o desenho das interfaces gráficas até a estrutura interna do código. As regras de *design* de XP estão mais ligadas a este segundo significado. Elas são:

- *Adote a simplicidade*: um *design* simples sempre é executado mais rapidamente do que um *design* complexo. Porém, a simplicidade é subjetiva e difícil de ser medida. Então, é a equipe que deve decidir o que é simples. Uma das melhores formas de obter simplicidade em um *design* é levar a sério o *Design Pattern* "Coesão Alta" (Wazlawick, 2015), porque ele vai levar a elementos de sistema (classes, módulos, métodos, componentes etc.) mais fáceis de se compreender, modificar e estender. Recomendam-se algumas qualidades subjetivas para determinar a simplicidade de um *design*:
  - *Testabilidade*: o sistema deve poder ser quebrado em unidades testáveis, como casos de uso, fluxos, operações de sistema, classes e métodos. Assim, podem-se escrever testes de unidade para verificar se o código está correto.
  - *Browseabilidade*: podem-se encontrar os elementos do *design* quando se precisa deles. Bons nomes e uso de boas disciplinas de modelagem, como polimorfismo, herança e delegação, ajudam nisso.
  - *Compreensibilidade e explicabilidade*: a compreensibilidade é uma qualidade subjetiva, porque um sistema pode ser bastante compreensível para quem está trabalhando nele, mas difícil para quem está de fora. Então, essa propriedade pode ser definida em termos de quão fácil é explicar o sistema para quem não participou de seu desenvolvimento.
- *Escolha uma metáfora de sistema*: uma boa metáfora de sistema ajuda a explicar seu funcionamento a alguém que está fora do projeto. Deve-se evitar que a compreensão sobre o sistema resida em pilhas de documentos. Nomes significativos e padrões de nomeação de elementos de programa devem ser cuidadosamente escolhidos e seguidos para que fragmentos de código sejam efetivamente reusáveis.
- *Use cartões CRC durante reuniões de projeto*: trata-se de uma técnica para encontrar responsabilidades e colaborações entre objetos. A equipe se reúne em torno de uma mesa e cada membro recebe um ou mais cartões representando instâncias de diferentes classes. Uma atividade (operação ou consulta) é simulada e, à medida que ela ocorre, os detentores dos cartões anotam responsabilidades do objeto (no lado esquerdo do cartão) e colaborações do objeto (no lado direito do cartão). A documentação dos processos pode ser feita com diagramas de sequência ou de comunicação da UML.
- *Crie* spikes *para reduzir riscos*: riscos de projeto importantes devem ser explorados de forma definitiva e exclusiva, ou seja, deve ser buscada uma *spike* para o problema identificado. Uma *spike* é, então, um desenvolvimento ou teste projetado especificamente para analisar e, se possível, resolver um risco. Caso o risco se mantenha, deve-se colocar um par de programadores durante uma ou duas semanas trabalhando exclusivamente para examiná-lo e mitigá-lo. A maioria das *spikes* não será aproveitada no projeto, podendo ser classificada como uma das formas de prototipação *throw-away*.

- *Nenhuma funcionalidade é adicionada antes da hora*: deve-se evitar a tentação de adicionar uma funcionalidade desnecessária só porque seria fácil fazer isso no momento e deixaria o sistema "melhor". Apenas o *necessário* deve ser feito no sistema. Desenvolver o que não é necessário é jogar tempo fora. Manter o código aberto a possíveis mudanças futuras tem a ver com simplicidade de *design*, mas adicionar funcionalidade ou flexibilidade desnecessária sempre deixa o *design* mais complexo e tem o efeito de uma bola de neve. Requisitos futuros só devem ser considerados quando estritamente exigidos pelo cliente. Flexibilidade em *design* é bom, mas toma tempo de desenvolvimento. Deve-se decidir quais requisitos efetivamente merecem ter uma implementação flexível.
- *Use refatoração sempre e onde for possível*: refatore sem pena. O XP não recomenda que se continue usando *design* antigo e ruim só porque ele funciona. Devem-se remover redundâncias, eliminar funcionalidades desnecessárias e rejuvenescer *designs* antiquados.

4.4.2.6 Regras de Codificação

Certamente a codificação é a principal atividade em desenvolvimento de software porque ela efetivamente gera o produto, enquanto que a maioria das outras atividades servem apenas para dar suporte a esta. Assim, as regras relacionadas à *codificação* de programas são as seguintes:

- *O cliente está sempre disponível*: o XP necessita que o cliente esteja disponível, de preferência pessoalmente, ao longo de todo o processo de desenvolvimento. Entretanto, em razão do longo tempo de duração de um projeto, a empresa cliente pode ser tentada a associar a ele um funcionário pouco experiente ou um estagiário. Contudo, ele não serve. Precisa ser um especialista, que deverá escrever as histórias de usuário, bem como priorizá-las e negociar sua inclusão em iterações. Pode parecer um investimento alto no tempo dos funcionários, mas isso é compensado pela ausência de necessidade de um levantamento de requisitos detalhado no início, bem como pelo fato de que não será entregue um sistema inadequado.
- *O código deve ser escrito de acordo com padrões aceitos*: os padrões de codificação mantêm o código compreensível e passível de refatoração por toda a equipe. Além disso, um código padronizado e familiar encoraja a sua posse coletiva.
- *Escreva o teste de unidade primeiro*: em geral, escrever o teste antes do código ajuda a escrever o código melhor. O tempo para escrever o teste e o código passa a ser o mesmo que se gastaria para escrever apenas o código, mas assim se obtém um código de melhor qualidade e é uma forma de garantir que ele continue correto mesmo após mudanças posteriores.
- *Todo código é produzido por pares*: embora pareça contrassensual, duas pessoas trabalhando em um computador podem produzir quase tanto código quanto duas pessoas trabalhando separadamente, mas em pares elas produzem com mais qualidade. Embora seja recomendado que haja um programador mais experiente, a relação não deve ser de professor-aluno, mas de colaboração entre iguais.
- *Só um par faz integração de código de cada vez*: a integração em paralelo pode trazer problemas de compatibilidade imprevistos, pois duas partes do sistema que nunca foram testadas juntas acabam sendo integradas sem serem testadas. Deve haver versões claramente definidas do produto. Então, para que equipes trabalhando em paralelo não tenham problemas na hora de integrar seu código ao produto, elas devem esperar sua vez. Devem-se estabelecer turnos de integração que sejam obedecidos.

- *Integração deve ser frequente*: os desenvolvedores devem integrar o código ao repositório em curtos períodos de tempo. Postergar a integração pode agravar o problema de todos estarem trabalhando em versões desatualizadas do sistema.
- *Defina um computador exclusivo para integração*: esse computador funciona como uma ficha de exclusividade (*token*) para a integração. A existência dele no ambiente de trabalho permite que toda a equipe veja quem está integrando uma funcionalidade e que funcionalidade é essa. O resultado da integração deve passar nos testes de unidade de forma que se obtenha estabilidade em cada versão, além de localidade nas mudanças e possíveis erros. Se os testes de integração falharem, essa unidade deverá ser depurada. Se a atividade de integração levar mais de dez minutos, isso significa que a unidade ainda precisa de alguma depuração adicional antes de ser integrada. Nesse caso, a integração deve ser abortada, retornando o sistema à última versão estável, e a depuração da unidade deve continuar sendo feita pelo par fora do ambiente de integração.
- *A posse do código deve ser coletiva*: não devem ser criados gargalos pela existência de donos de código. Todos devem ter autorização para modificar, consertar ou refatorar partes do sistema. Para isso funcionar, os desenvolvedores devem sempre desenvolver os testes de unidade com o código, seja novo, seja modificado. Dessa forma, existe sempre uma garantia de que o software satisfaça as condições de funcionamento. Também deve-se manter o código sob rigoroso controle de versão, para que eventuais modificações inadequadas possam ser desfeitas. Não ter um dono de partes do sistema também diminui o impacto da perda de membros da equipe.

4.4.2.7 Regras de Teste

Por fim, as regras referentes ao teste do software em XP são as seguintes:

- *Todo código deve ter testes de unidade* (Seção 13.2.1): esse é um dos pilares do XP. Inicialmente, o desenvolvedor XP deve obter um *framework de teste de unidade*, como por exemplo *jUnit*, para Java ou o *unittest*, que já vem com o pacote da linguagem Python. Depois, para cada nova funcionalidade, ele escreve primeiro o teste de unidade e depois implementa a funcionalidade. Testes de unidade favorecem a posse coletiva do código, porque o protegem de ser acidentalmente danificado.
- *Todo código deve passar pelos testes de unidade antes de ser entregue*: exigir isso ajuda a garantir que sua funcionalidade seja corretamente implementada. Os testes de unidade também favorecem a refatoração, porque protegem o código de mudanças de funcionalidade indesejadas. A modificação de uma funcionalidade deverá ser sempre precedida da modificação dos testes de unidade correspondentes.
- *Quando um erro de funcionalidade é encontrado, testes de aceitação são criados*: um erro de funcionalidade identificado exige que testes de aceitação (ou seja, testes nos quais o cliente vai avaliar o sistema) sejam criados para proteger o sistema. Assim, os clientes podem explicar claramente aos desenvolvedores o que eles esperam que seja modificado para corrigir o erro.
- *Testes de aceitação são executados com frequência e os resultados são publicados*: testes de aceitação são criados a partir de histórias de usuário. Durante uma iteração, as histórias de usuário selecionadas serão traduzidas em testes de aceitação. Esses testes são do tipo *funcional* (Seção 13.5) e representam uma ou mais funcionalidades desejadas.

Testes de aceitação devem ser automatizados de forma que possam ser executados com frequência.

4.5 FDD - *Feature Driven Development*

O FDD ou *Feature-Driven Development* (Desenvolvimento Dirigido por Funcionalidade) é um método ágil que enfatiza o uso de orientação a objetos. Esse modelo foi apresentado, em 1997, por Peter Coad e Jeff de Luca como a evolução de um processo mais antigo denominado *Método Coad* (Coad, De Luca & Lefebvre, 1997). A rigor, Coad teve mais influência nos aspectos de modelagem orientada a objetos e de Luca, um gerente de projetos, com as técnicas de gerenciamento enxuto, incremental e iterativo. Duas atualizações importantes do modelo foram apresentadas por Palmer e Mac Felsing (2002) e por Anderson (2004).

De todos os modelos ágeis originais, este é o que mais se parece com um processo prescritivo, pois é dividido em fases com etapas para as quais são descritas atividades, entradas e saídas. FDD talvez possa ser identificado com os assim chamados *processos leves*, coisa que já não acontece com a maioria dos outros modelos ágeis que, por não terem atividades sequenciadas, entradas e saídas, não podem ser considerados como processos.

O FDD possui apenas duas grandes fases:

- *Concepção e planejamento*: implica em pensar um pouco (em geral de uma a duas semanas) antes de começar a construir.
- *Construção*: desenvolvimento iterativo do produto em ciclos de uma a duas semanas.

A fase de concepção e planejamento possui três etapas:

- DMA – *Desenvolver Modelo Abrangente*, em que se preconiza o uso da modelagem orientada a objetos.
- CLF – *Construir Lista de Funcionalidades*, em que se pode aplicar a decomposição funcional para identificar as funcionalidades que o sistema deve disponibilizar.
- PPF – *Planejar por Funcionalidade*, em que o planejamento das iterações rápidas é feito em função das funcionalidades identificadas.

Já a fase de construção incorpora duas etapas:

- DPF – *Detalhar por Funcionalidade*, que corresponde a realizar o *design* orientado a objetos do sistema.
- CPF – *Construir por Funcionalidade*, que corresponde a construir e testar o software utilizando linguagem e técnica de teste orientadas a objetos.

A estrutura geral do modelo FDD pode ser representada conforme a Figura 4.12. Cada uma das cinco etapas é descrita nas subseções a seguir.

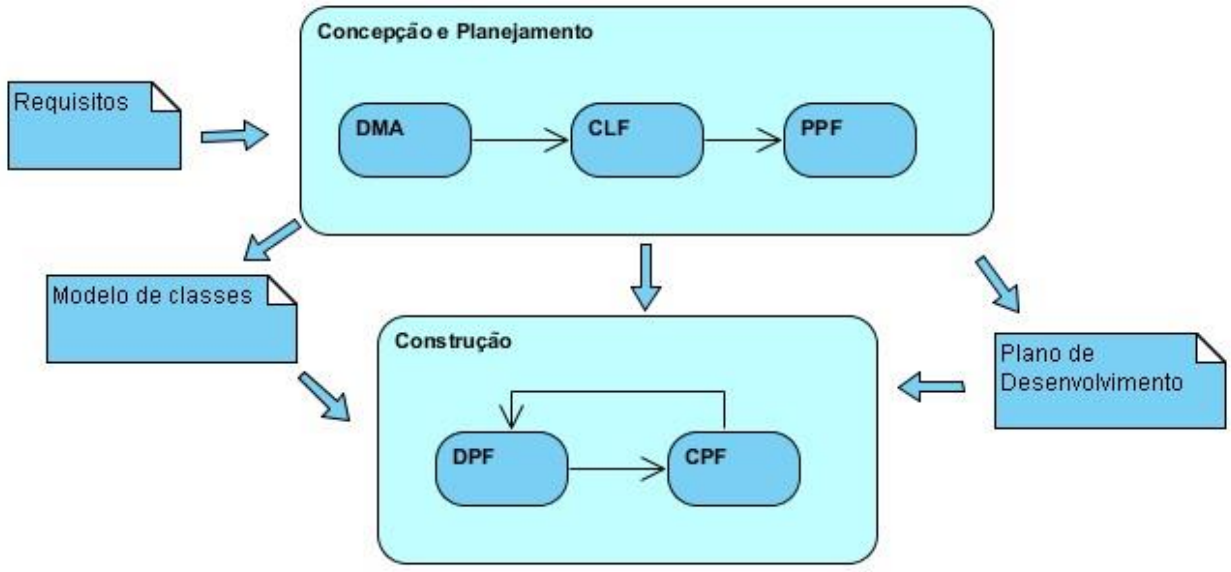

Figura 4.12 Estrutura geral do modelo FDD

4.5.1 DMA – Desenvolver Modelo Abrangente

A primeira fase se inicia com o desenvolvimento de um modelo de negócio abrangente, seguido do desenvolvimento de modelos mais específicos (de domínio). Estabelecem-se grupos de trabalho formados por desenvolvedores e especialistas de domínio. Os vários modelos específicos assim desenvolvidos são avaliados em um *workshop* e uma combinação deles é formada para ser usada como modelo da aplicação.

Os modelos de negócio e de domínio são representados por diagramas de classes e sua construção deve ser liderada por um modelador com experiência em orientação a objetos. Depois, esse modelo de classes ou modelo conceitual passará a ser refinado na etapa DPF (Detalhar por Funcionalidade).

Para iniciar o desenvolvimento do modelo abrangente, é necessário que alguns papéis já tenham sido definidos, especialmente os de arquiteto líder, programadores líderes e especialistas de domínio.

As atividades individuais que compõem essa etapa são as seguintes:

- *Formar a equipe de modelagem*: é uma atividade obrigatória sob a responsabilidade do gerente de projeto. As equipes devem ser montadas com especialistas de domínio, clientes e desenvolvedores. Deve haver rodízio entre os membros das equipes, de forma que todos possam ver o processo de modelagem em ação.
- *Estudo dirigido sobre o domínio*: é uma atividade obrigatória sob responsabilidade da equipe de modelagem. Um especialista de domínio deve apresentar sua área de domínio para a equipe, especialmente os aspectos conceituais.
- *Estudar a documentação*: é uma atividade opcional sob responsabilidade da equipe de modelagem. A equipe estuda a documentação que eventualmente estiver disponível sobre o domínio do problema, inclusive sistemas legados.
- *Desenvolver o modelo*: é uma atividade obrigatória sob responsabilidade das equipes de modelagem específicas. Grupos de até três pessoas vão trabalhar para criar modelos candidatos para suas áreas de domínio. O arquiteto líder pode considerar apresentar às

equipes um modelo base para facilitar seu trabalho. Ao final, as equipes apresentam seus modelos, que são consolidados em um modelo único.

- *Refinar o modelo de objetos abrangente*: é uma atividade obrigatória sob responsabilidade do arquiteto líder e da equipe de modelagem. As decisões tomadas para o desenvolvimento dos modelos específicos de domínio poderão afetar a forma do modelo geral do negócio, que deve então ser refinado.

O resultado dessas atividades deve ser verificado pela própria equipe de modelagem (verificação interna) e também pelo cliente ou pelos especialistas de domínio (verificação externa).

As saídas esperadas desse conjunto de atividades são:

- O *modelo conceitual*, apresentado como um diagrama de classes e suas associações.
- *Métodos e atributos*, eventualmente identificados para as classes.
- *Diagramas de sequência ou máquina de estados* para as situações que exigirem esse tipo de descrição.
- *Comentários sobre o modelo* para indicar por que determinadas decisões de *design* foram tomadas em vez de outras.

As técnicas para construir esses diagramas de classes, sequência e máquina de estados podem ser encontradas detalhadamente nos Capítulos 2 a 8 de Wazlawick (2015).

4.5.2 CLF – Construir Lista de Funcionalidades

A etapa CLF, ou *construir lista de funcionalidades*, vai identificar as funcionalidades que satisfazem os requisitos. A equipe vai decompor o domínio em *áreas de negócio*, conforme a etapa DMA. As áreas de negócio serão decompostas em *atividades de negócio* e estas, por sua vez, em *passos de negócio* (Figura 4.13). O processo é relativamente parecido com uma análise de casos de uso, na qual as áreas de negócio correspondem a casos de uso de negócio, as atividades de negócio a casos de uso de sistema e os passos de negócio às transações de casos de uso de sistema.

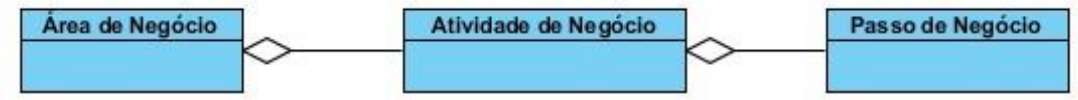

Figura 4.13 Estrutura conceitual da lista de funcionalidades

Para iniciar as atividades é necessário que os especialistas de domínio, programadores líderes e arquiteto líder estejam disponíveis.

A etapa é composta por uma única atividade: *construir a lista de funcionalidades*, a qual é obrigatória e de responsabilidade da equipe da lista de funcionalidades (formada pelos programadores líderes da etapa anterior).

Assim, a equipe vai listar as funcionalidades em três níveis:

- *Áreas de negócio*, oriundas das atividades de DMA. Por exemplo, "vendas".
- *Atividades de negócio*, que são a decomposição funcional das áreas de negócio. Por exemplo, registrar pedido, faturar pedido, registrar pagamento de venda etc.
- *Passos de atividades de negócio*, que são a descrição sequencial das funcionalidades necessárias para realizar as atividades de negócio. Por exemplo, identificar cliente para pedido, registrar produto e quantidade do pedido, aplicar desconto padrão ao pedido etc.

As funcionalidades não devem ser ações realizadas sobre tecnologia (como “abrir janela” ou “acessar menu”), mas ações que tenham significado para o cliente, independentemente de tecnologia. Boas funcionalidades devem poder ser nomeadas como uma instância da tríade *ação*/*resultado*/*objeto*. Por exemplo, “apresentar total das vendas no mês” ou “registrar concordância do contratado”.

Espera-se que o tempo para implementação de uma funcionalidade nunca seja superior a duas semanas, sendo esse um limite absoluto, já que o tempo esperado seria de poucos dias. Funcionalidades com esforço estimado de menos de um dia não precisam ser consideradas na lista de funcionalidades, mas possivelmente serão parte de outras mais abrangentes. De outro lado, quando uma funcionalidade parece levar mais de duas semanas para ser desenvolvida (por uma pessoa), ela deve ser quebrada em funcionalidades menores.

A avaliação dessa atividade também pode ser feita de forma interna pela equipe de lista de funcionalidades ou, externamente, pelos usuários, clientes e especialistas de domínio.

As saídas dessa atividade são as seguintes:

- *Lista de áreas de negócio*.
- Para cada área, uma *lista de atividades de negócio* dentro da área.
- Para cada atividade, uma *lista de passos de atividade* ou funcionalidades que permitem realizar a atividade.

4.5.3 PPF – Planejar por Funcionalidade

A etapa PPF, ou *planejar por funcionalidade*, ainda na primeira fase do FDD, visa gerar o plano de desenvolvimento para a fase seguinte, que é formada por várias iterações. O planejamento vai indicar quais atividades de negócio da lista definida em CLF serão implementadas e quando. O planejador deve levar em consideração os seguintes aspectos para agrupar e priorizar as atividades de negócio:

- *Complexidade das funcionalidades*: em função do risco, as atividades com funcionalidades de maior complexidade devem ser tratadas primeiro.
- *Dependências entre as funcionalidades em termos de classes*: funcionalidades dependentes devem preferencialmente ser abordadas juntas, pois isso evita a fragmentação do trabalho nas classes entre as diferentes equipes.
- *Carga de trabalho da equipe*: deve ser usado um método de estimativa (Capítulo 7) para que seja conhecido o esforço para a implementação de cada atividade ou funcionalidade, e o trabalho deve ser atribuído à equipe em função de sua capacidade e dessa estimativa.

Ao contrário de outros métodos ágeis, o FDD propõe que a posse das classes seja atribuída aos programadores líderes, ou seja, cada um se responsabiliza por um subconjunto das classes. Então, ao fazer a divisão da carga de trabalho, em geral se estará definindo também a posse das classes.

A entrada para as atividades dessa etapa consiste na lista de funcionalidades construída em CLF.

São quatro as atividades da etapa PPF:

- *Formar a equipe de planejamento*: é uma atividade obrigatória de responsabilidade do gerente do projeto. Essa equipe deve ser formada pelo gerente de desenvolvimento e pelos programadores líderes.

- *Determinar a sequência de desenvolvimento*: é uma atividade obrigatória de responsabilidade da equipe de planejamento. A equipe deve determinar o prazo de conclusão do desenvolvimento de cada uma das atividades de negócio. Esse prazo deve ser determinado em função de mês e ano, ou seja, de prazos mensais. A sequência de desenvolvimento deve ser construída levando-se em consideração os seguintes fatores:
    - Priorizar as atividades com funcionalidades mais complexas ou de alto risco.
    - Alocar juntas atividades ou funcionalidades dependentes umas das outras, se possível.
    - Considerar marcos externos, quando for o caso, para a criação de releases.
    - Considerar a distribuição de trabalho entre os proprietários das classes.
- *Atribuir atividades de negócio aos programadores líderes*: é uma atividade obrigatória de responsabilidade da equipe de planejamento e vai determinar quais programadores líderes serão proprietários de quais atividades de negócio. Essa atribuição de propriedade deve ser feita considerando-se os seguintes critérios:
    - Dependência entre as funcionalidades e as classes das quais os programadores líderes já são proprietários.
    - A sequência de desenvolvimento.
    - A complexidade das funcionalidades a serem implementadas em função da carga de trabalho alocada aos programadores líderes.
- *Atribuir classes aos desenvolvedores*: é uma atividade obrigatória de responsabilidade da equipe de planejamento, que deve atribuir a propriedade das classes aos desenvolvedores. Essa atribuição é baseada em:
    - Distribuição de carga de trabalho entre os desenvolvedores.
    - Complexidade das classes (priorizar as mais complexas ou de maior risco).
    - Intensidade de uso das classes (priorizar as classes altamente usadas).
    - Sequência de desenvolvimento.

A verificação das atividades é feita unicamente de forma interna. A própria equipe de planejamento faz uma autoavaliação para verificar se as atividades foram desenvolvidas adequadamente.

O resultado ou artefato de saída das atividades é o *plano de desenvolvimento*, que consiste em:

- Prazos (mês e ano) para a conclusão do desenvolvimento referente a cada uma das atividades de negócio.
- Atribuição de programadores líderes a cada uma das atividades de negócio.
- *Prazos* (*mês e ano*) *para a conclusão do desenvolvimento referente a cada uma das áreas de negócio* (isso é derivado da última data de conclusão das atividades de negócio incluídas na respectiva data).
- Lista dos desenvolvedores e das classes das quais eles são proprietários.

Finda esta fase, o projeto entra nas iterações de desenvolvimento, nas quais duas etapas se alternam: *Detalhar por Funcionalidade* e *Construir por Funcionalidade*, as quais são explicadas nas próximas duas subseções.

4.5.4 DPF – Detalhar por Funcionalidade

A etapa DPF, ou *detalhar por funcionalidade*, é a primeira executada na fase iterativa do FDD. Ela consiste basicamente em produzir o design de implementação da funcionalidade, que costuma ser realizado por diagramas de sequência ou comunicação (Wazlawick, 2015).

A atividade de detalhamento é realizada para cada funcionalidade identificada dentro das atividades de negócio. A atribuição do trabalho aos desenvolvedores é feita pelo programador líder, que poderá considerar, para tanto, a conexão entre as funcionalidades ou ainda a posse (propriedade) das classes envolvidas.

Como entrada, as atividades dessa etapa necessitam que o planejamento da etapa anterior tenha sido concluído. As atividades de DPF são as seguintes:

- *Formar a equipe de funcionalidades*: é uma atividade obrigatória de responsabilidade do programador líder. O programador líder cria um pacote de funcionalidades a serem trabalhadas e, em função das classes envolvidas com essas funcionalidades, define a equipe de funcionalidades com os proprietários dessas classes. O programador líder também deve atualizar o controle de andamento de projeto, indicando que esse pacote de funcionalidades está sendo trabalhado.
- *Estudo dirigido de domínio*: é uma atividade opcional de responsabilidade do especialista de domínio. Se a funcionalidade for muito complexa, o especialista de domínio deve apresentar um estudo dirigido sobre a funcionalidade no domínio em que ela se encaixa. Por exemplo, uma funcionalidade como "calcular impostos" pode ser bastante complicada e merecerá uma apresentação detalhada por um especialista de domínio antes que os desenvolvedores comecem a projetá-la.
- *Estudar a documentação de referência*: é uma atividade opcional de responsabilidade da equipe de funcionalidades. Dependendo da complexidade da funcionalidade, pode ser necessário que a equipe estude documentos disponíveis, como relatórios, desenhos de telas, padrões de interface com sistemas externos etc.
- *Desenvolver os diagramas de sequência*: é uma atividade opcional de responsabilidade da equipe de funcionalidades. Os diagramas necessários para descrever a funcionalidade podem ser desenvolvidos. Assim como outros artefatos, devem ser submetidos a um sistema de controle de versão (Seção 10.2). Decisões de *design* devem ser anotadas.
- *Refinar o modelo de objetos*: é uma atividade obrigatória de responsabilidade do programador líder. Com o uso intensivo do sistema de controle de versões, o programador líder cria uma área de trabalho a partir de uma cópia das classes necessárias do modelo, disponibilizando essa cópia para a equipe de funcionalidades. Esta, por sua vez, terá acesso compartilhado à cópia, mas o restante do pessoal não, até que ela seja salva como uma nova versão das classes no sistema de controle de versões. Então, a equipe de funcionalidades deverá adicionar métodos, atributos, associações e novas classes que forem necessárias.
- *Escrever as interfaces* (*assinatura*) *das classes e métodos*: é uma atividade obrigatória sob responsabilidade da equipe de funcionalidades. Utilizando a linguagem de programação alvo da aplicação, os proprietários das classes escrevem as interfaces das classes, incluindo os atributos e seus tipos, além da declaração dos métodos, incluindo tipos de parâmetros e retornos, exceções e mensagens.

A verificação do produto dessas atividades é feita por avaliação interna de inspeção do *design*. Essa avaliação deve ser feita pela própria equipe de funcionalidades, mas outros membros do projeto podem participar dela. Após o aceite do produto final, uma lista de atividades é gerada

para cada desenvolvedor (lembrando que, até aqui, apenas assinaturas dos métodos foram definidas e eles ainda precisam ser efetivamente implementados), e a nova versão das classes é gravada (*commit*) no sistema de controle de versões.

A saída desta etapa é um *pacote de design* inspecionado e aprovado. Esse pacote consiste de:

- Uma capa com comentários que descreve o pacote de forma suficientemente clara.
- Os requisitos abordados na forma de atividades e/ou funcionalidades.
- Os diagramas de sequência.
- Os projetos alternativos (se houver).
- O modelo de classes atualizado.
- As interfaces de classes geradas.
- A lista de tarefas para os desenvolvedores, gerada em função dessas atividades.

Passa-se em seguida para a etapa de finalização, na qual o código é efetivamente construído.

4.5.5 CPF – Construir por Funcionalidade

A etapa CPF, ou *construir por funcionalidade*, também executada dentro da fase iterativa do FDD, tem como objetivo a produção do código para as funcionalidades identificadas. A partir do *design* gerado pela etapa anterior e de acordo com o cronograma definido pelo programador líder, os desenvolvedores devem construir e testar o código necessário para cada classe atribuída. Após a inspeção do código pelo programador líder, ele é salvo no sistema de controle de versão e passa a ser considerado *build*.

A entrada para essas atividades é o pacote de *design* gerado na etapa anterior.

As atividades envolvidas são:

- *Implementar classes e métodos*: é uma atividade obrigatória sob responsabilidade da equipe de funcionalidades. A atividade é realizada pelos proprietários de classes em colaboração uns com os outros.
- *Inspecionar o código*: é uma atividade obrigatória sob responsabilidade da equipe de funcionalidades. Uma inspeção do código pode ser feita pela própria equipe ou por analistas externos, mas é sempre coordenada pelo programador líder e pode ser feita antes ou depois dos testes de unidade.
- *Teste de unidade*: é uma atividade obrigatória sob responsabilidade da equipe de funcionalidades. Os proprietários de classes definem e executam os testes de unidade de suas classes para procurar eventuais defeitos ou inadequação a requisitos. O programador líder poderá determinar testes de integração entre as diferentes classes quando julgar necessário.
- *Promover à versão atual* (*build*): é uma atividade obrigatória sob responsabilidade do programador líder. À medida que os desenvolvedores vão reportando sucesso nos testes de unidade, o programador líder poderá promover a versão atual de cada classe individualmente à *build*, não sem antes passar pelos testes de integração.

A verificação do produto de trabalho é feita internamente pelo programador líder e pela equipe de funcionalidades.

Os resultados ou saídas dessas atividades são:

- Classes que passaram com sucesso em testes de unidade e integração e foram, por isso, promovidas à versão atual (*build*).
- Disponibilização de um conjunto de funcionalidades com valor para o cliente.

A partir da finalização estas atividades e produção das saídas o processo retorna à etapa DPF – detalhar por funcionalidade, para abordar as funcionalidades planejadas para a iteração seguinte.

4.6 *Crystal Clear*

Crystal Clear é um método ágil criado por Alistair Cockburn em 1997. Ele pertence à família de métodos *Crystal*, mais ampla, iniciada em 1992 (Cockburn, 2004). Os outros métodos são conhecidos como *Yellow*, *Orange*, *Orange-Web*, *Red*, *Maroon*, *Diamond e Sapphire*. *Clear* é o primeiro método da família.

A escolha da cor a ser usada como método de desenvolvimento depende basicamente do tamanho da equipe de desenvolvimento e do tipo de risco associado ao projeto. *Clear* é recomendado para equipes de até seis pessoas, *yellow* para até vinte, *orange* até quarenta, *red* até oitenta e *maroon* até 200. Para equipes acima de 200 pessoas recomenda-se *diamond* ou *sapphire*.

Na escala de risco também se considera que quanto maior este for mais escura deve ser a cor. A escala de risco inicia com desconforto, passando por perdas financeiras aceitáveis, perdas financeiras importantes e risco a vida humana.

À medida que cresce o tamanho da equipe e do risco do projeto, os métodos vão ficando cada vez mais formais. Assim, *Crystal Clear* é o mais ágil de todos.

Crystal Clear é, portanto, uma abordagem ágil adequada a equipes pequenas (de no máximo seis pessoas) que trabalham juntas (na mesma sala ou em salas contíguas). Em geral, a equipe é composta por um *designer* líder e por mais dois a cinco programadores. O método propõe, entre outras coisas, o uso de radiadores de informação, como quadros e murais à vista de todos, acesso fácil a especialistas de domínio, eliminação de distrações, cronograma de desenvolvimento baseado na técnica de *timeboxing* e ajuste do método quando necessário

Segundo Cockburn (2004), a família *Crystal* é centrada em pessoas (*human powered*), ultraleve e na medida (*stretch to fit*):

- *Centrada em pessoas*: significa que o foco para o sucesso de um projeto está em melhorar o trabalho das pessoas envolvidas. Enquanto outros métodos podem ser centrados em processo, em arquitetura ou em ferramenta, *Crystal* é centrado em pessoas.
- *Ultraleve*: significa que independentemente do tamanho do projeto, a família *Crystal* fará o possível para reduzir a burocracia, a papelada e o *overhead*, que existirão na medida suficiente para as necessidades do projeto.
- *Na medida*: significa que o *design* começa com algo menor do que se pensa que seja preciso e, depois, é aumentado apenas o suficiente para suprir as necessidades. Parte-se do princípio de que é mais fácil e barato aumentar um sistema do que cortar coisas que já foram feitas, mas são desnecessárias.

Os sete pilares do método são listados a seguir. Os três primeiros são condições *sine qua non* do método; os outros quatro são recomendados para levar a equipe à zona de conforto em relação a sua capacidade de desenvolver software de forma adequada:

- *Entregas frequentes*: as entregas ao cliente devem acontecer no máximo a cada dois meses, com versões intermediárias.
- *Melhoria reflexiva*: os membros da equipe devem discutir frequentemente se o projeto está no rumo certo e comunicar descobertas que possam impactar o projeto.
- *Comunicação osmótica*: a equipe deve trabalhar em uma única sala para que uns possam ouvir a conversa dos outros e participar dela quando julgarem conveniente. Considera-se

uma boa prática interferir no trabalho dos outros. O método propõe que os programadores trabalhem individualmente, mas bem próximos uns dos outros. Isso pode ser considerado um meio-termo entre a programação individual e a programação em pares, pois cada um tem a sua atribuição, mas todos podem se auxiliar mutuamente com frequência.

- *Segurança pessoal*: os desenvolvedores devem ter certeza de que poderão falar sem medo de repreensões. Quando as pessoas não falam, suas fraquezas viram fraquezas da equipe.
- *Foco*: espera-se que os membros da equipe tenham dois ou três tópicos de alta prioridade nos quais possam trabalhar tranquilamente, sem receber novas atribuições.
- *Acesso fácil a especialistas*: especialistas de domínio, usuários e cliente devem estar disponíveis para colaborar com a equipe de desenvolvimento.
- *Ambiente tecnologicamente rico*: o ambiente de desenvolvimento deve permitir testes automáticos, gerenciamento de configuração e integração frequente.

Considera-se que aplicar *Crystal Clear* tem mais a ver com adquirir as qualidades mencionadas acima do que seguir procedimentos. As subseções seguintes aprofundam cada um destes pilares.

4.6.1 Entregas Frequentes

As vantagens de *entregas frequentes* como forma de redução de risco em um projeto de software são indiscutíveis, e os ciclos de vida modernos aderem a esse princípio.

Na maioria das vezes em que se fala em entregas frequentes, imaginam-se sistemas feitos sob medida para um cliente conhecido e interessado em dar *feedback* para o processo de desenvolvimento, mas nem sempre o software é feito para esse tipo de cliente. Muitos sistemas desenvolvidos hoje são distribuídos pela Internet; então a comunidade de usuários pode não ser totalmente conhecida. Nesses casos, a tática de efetuar entregas (disponibilização de versões) com muita frequência (por exemplo, semanal) pode ser irritante para alguns usuários. De outro lado, diminuir a frequência das entregas pode fazer que a equipe de desenvolvimento perca um importante *feedback*. A solução, nesse caso, é definir um conjunto de usuários amigáveis que não se importem em receber versões com frequência maior do que os usuários normais.

Se usuários amigáveis não forem encontrados, a sugestão é que o ciclo de desenvolvimento seja finalizado como se a entrega fosse ser feita, ou seja, uma falsa entrega deve ser criada, com toda a formalidade, como se fosse uma entrega verdadeira, mas não deve ser disponibilizada aos usuários.

Uma *iteração*, que possivelmente produz uma entrega, não deve ser confundida com uma *integração*. A integração pode ocorrer de hora em hora, sempre que algum programador tiver criado uma nova versão de um componente que possa ser integrado ao sistema. Já a iteração pressupõe o fim de um ciclo de atividades predefinidas e controladas, incluindo várias integrações ao longo do ciclo. Para *Crystal Clear*, é importante que o fim de uma iteração seja marcado com uma *celebração*, pois, assim, a equipe ganhará ritmo emocional com a sensação de etapa concluída. Afinal, ela é formada por pessoas, e não máquinas.

Crystal Clear assume que uma iteração possa durar de uma hora a três meses. Contudo, o mais comum é que as iterações durem de duas semanas a dois meses. O importante é que seja usada a técnica de *timeboxing* e que o prazo final das iterações não seja mudado, pois um atraso levará a outros e o ritmo emocional da equipe poderá baixar. Uma estratégia melhor é manter o prazo, deixar a equipe disponibilizar aquilo que for possível naquele intervalo de tempo e, depois, se necessário, replanejar as iterações seguintes.

Algumas equipes poderão tentar usar a técnica de requisitos fixos (*requirements locking*), ou seja, assumir que durante uma iteração os requisitos ou suas prioridades não poderão mudar. Isso

permitirá à equipe saber que poderá completar suas atribuições sem mudanças de rumo no meio do processo. Normalmente, porém, em ambientes não hostis e bem-comportados, não é necessário estabelecer isso como regra, pois acaba acontecendo naturalmente.

É importante frisar que entregas frequentes têm a ver com entregar software ao cliente, e não simplesmente completar iterações. Algumas equipes poderão fazer que cada iteração corresponda a uma entrega; outras entregarão software a cada duas, três ou quatro iterações; outras ainda definirão no calendário datas específicas para iterações que produzirão entregas. Em todos os casos, não basta a equipe fazer iterações rápidas; é necessário entregar software com frequência pois senão, nesse meio-tempo, o cliente não terá dado nenhum *feedback* sobre o que foi desenvolvido.

4.6.2 Melhoria Reflexiva

Uma das coisas que podem fazer um projeto que está falhando dar a volta por cima é a *melhoria reflexiva*. Essa prática indica que a equipe deve se reunir, discutir o que está e o que não está funcionando, avaliar formas de melhorar o que não está funcionando e, o que é mais importante, colocar mudanças em prática. Não é necessário gastar muito tempo com essa atividade. Poucas horas por mês normalmente são suficientes.

É interessante observar que muitos projetos enfrentam grandes dificuldades já nas primeiras iterações. Entretanto, o que poderia levar a uma catástrofe logo de início deve ser considerado um ponto de partida para reflexão e aprimoramento das práticas (refletir e melhorar). De outro lado, caso esses problemas não sejam seriamente abordados logo no início, poderão minar o projeto rapidamente e desmoralizar a equipe de forma que se tornará impossível retomar o ritmo, o que fatalmente levará ao cancelamento do projeto.

As mudanças que precisam ser feitas às vezes envolvem pessoas, outras vezes, tecnologia, e, em outras, as práticas de projeto da equipe. A sugestão de *Crystal Clear* é que, a cada semana, mensalmente ou uma ou duas vezes por ciclo de desenvolvimento, a equipe se reúna em um *workshop* de reflexão ou retrospectiva de iteração para discutir as coisas que estão funcionando e aquelas que não estão funcionando. É preciso ser feita uma lista das coisas que serão mantidas e daquelas que devem mudar. Essas listas devem ser colocadas à vista de todos para que sejam gravadas e efetivamente mudadas nas iterações seguintes.

4.6.3 Comunicação Osmótica

Comunicação osmótica é aquela em que a informação deve fluir pelo ambiente, ou seja, as pessoas devem ser capazes de ouvir a conversa das outras e intervir, se desejarem, ou continuar seu trabalho. Isso costuma ser obtido quando se colocam todos os desenvolvedores em uma mesma sala. Além disso, é importante que as telas dos computadores sejam acessíveis, pois em alguns casos é interessante que um pequeno grupo possa se reunir em frente a um computador para visualizar problemas e dar sugestões. Assim, devem ser evitados *designs* de sala que impeçam esse tipo de visualização.

Segundo Cockburn (2004), quando a comunicação osmótica ocorre, as questões e respostas fluem naturalmente pelo ambiente e, surpreendentemente, com pouca perturbação para a equipe. Ele coloca a seguinte questão: “Leva mais de trinta segundos para a sua pergunta chegar aos olhos e ouvidos de alguém que possa respondê-la?”. Se a resposta for sim, o projeto pode enfrentar dificuldades.

A comunicação osmótica tem custo baixo, mas é altamente eficiente em termos de *feedback*, pois os erros são corrigidos antes de se tornarem problemas mais sérios e a informação é disseminada rapidamente.

Embora a comunicação osmótica seja valiosa também para projetos e equipes de grande porte, fica cada vez mais difícil obtê-la nessas condições. Pode-se tentar deixar as equipes em salas próximas, mas, ainda assim, a comunicação osmótica só vai ocorrer entre pessoas da mesma sala. Outra possibilidade, no caso de equipes grandes ou distribuídas, seria utilizar ferramentas de comunicação, como videoconferência e *chat on-line*, de forma que as questões sejam colocadas de uma pessoa para a outra, mas vistas por toda a equipe.

Um problema que pode surgir com a comunicação osmótica é o excesso de ruído na sala ou um fluxo de informação muito grande dirigido ao desenvolvedor mais experiente. Porém, equipes conscientes acabam se autorregulando e autodisciplinando para evitar tais problemas. Isolar o programador líder em outra sala acaba não sendo uma boa solução, pois, se ele é o mais experiente, é natural que acabe sendo muito requisitado, e esse é exatamente o seu papel: ajudar os demais programadores a crescer. Ter o programador líder na mesma sala onde trabalha a equipe é uma estratégia denominada *expert in the earshot* (especialista ao alcance do ouvido). Porém, sempre existem situações extremas. Se o programador líder for tão requisitado pela equipe que não consegue mais fazer avanços em seu próprio trabalho, deverá reservar para si horários em que não estará disponível para a equipe. Essa técnica é denominada "cone de silêncio". O horário deve ser estabelecido de acordo com a necessidade e respeitado por todos.

4.6.4 Segurança Pessoal

Segurança pessoal tem relação com o fato de que as pessoas podem falar sobre coisas que estão incomodando sem temer represálias ou reprimendas. Segundo Cockburn (2004), isso envolve, entre outras coisas, dizer ao gerente que o cronograma não é realístico, que o *design* de um colega precisa melhorar ou até mesmo que ele precisa tomar banho com mais frequência. A segurança pessoal é muito importante, porque com ela a equipe consegue descobrir quais são suas fraquezas e repará-las. Sem ela, as pessoas não vão falar e as fraquezas vão continuar minando a equipe.

A segurança pessoal é um passo na direção da *confiança*, que consiste em dar ao outro poder sobre si mesmo. Algumas pessoas confiam no outro até que uma prova em contrário as faça rever essa confiança; outras evitam confiar até que tenham segurança de que o outro não vai prejudicá-las.

Existem várias formas pelas quais uma pessoa pode prejudicar outras no ambiente de trabalho ou até mesmo prejudicar o trabalho. Há pessoas que mentem, pessoas que são incompetentes, pessoas que sabotam o trabalho dos outros tanto ativamente quanto por não lhes fornecer informações ou orientações importantes quando necessário. Considerando essas formas de prejuízo, pode ser pedir demais às pessoas de uma equipe que simplesmente confiem umas nas outras. Assim, é mais fácil iniciar com comunicação franca, em que cada uma dirá o que a incomoda e a equipe vai regular ações e comportamentos a partir disso.

Segundo Cockburn (2004), estabelecer confiança envolve expor as pessoas a situações em que outros poderiam prejudicá-las e mostrar que isso não acontece. Por exemplo, um chefe pode expor um erro no *design* de um desenvolvedor e, em vez de puni-lo, dar-lhe suporte para que corrija o erro, mostrando que isso é parte do processo de autodesenvolvimento.

É importante mostrar que as pessoas não serão prejudicadas mesmo se demonstrarem ignorância sobre algum assunto em relação a sua área de conhecimento, ressaltando que lacunas de conhecimento sempre são oportunidades para aprender mais.

Além disso, é importante conscientizar as pessoas a interpretarem a forma de os outros se comunicarem como não agressivas, mesmo durante uma discussão. Uma discussão pode ser motivo para uma briga, mas em um ambiente saudável deve ser uma forma de confrontar diferentes pontos de vista. Mesmo que não haja consenso, o respeito pela opinião do outro deve prevalecer mesmo se for contra as próprias convicções. As pessoas devem ouvir umas às outras com boa vontade, e opiniões diversas sempre devem ser interpretadas como possibilidades ou oportunidades de aprender algo.

Isso tudo é importante para que as pessoas percebam que, com a ajuda dos outros, poderão resolver melhor problemas complexos do que se tentassem sozinhas.

A confiança é reforçada pelo princípio de entregas frequentes, porque, no momento de uma entrega, será possível ver quem realmente fez seu trabalho e quem falhou. Com segurança pessoal, todos poderão falar de seus problemas e limitações nos *workshops* de melhoria reflexiva, para que as falhas sejam minimizadas ou eliminadas no futuro.

4.6.5 Foco

Foco implica primeiramente saber em que se vai trabalhar e, depois, contar com tempo, espaço e paz de espírito para fazer o trabalho. Saber quais são as prioridades é algo usualmente determinado pela gerência. O tempo e a paz de espírito vêm de um ambiente de trabalho onde as pessoas não são arrancadas de suas atividades para realizar outras, muitas vezes sem relação com o que se faz originalmente.

Apenas definir prioridades para os desenvolvedores não é suficiente. Deve-se permitir a eles efetivamente concentrar-se nessas atividades. Um desenvolvedor interrompido de sua linha de raciocínio para apresentar relatórios ou demos de última hora, participar de reuniões ou consertar defeitos recém-descobertos gastará vários minutos para retomar sua linha de raciocínio após a interrupção. Se essas interrupções acontecerem várias vezes ao dia, não é incomum que o desenvolvedor passe a ficar ocioso nos intervalos, apenas esperando a próxima interrupção – se ele for interrompido sempre que estiver retomando o ritmo de trabalho, logo vai perceber a futilidade de tentar se concentrar.

Note-se, porém, que esse princípio não contradiz o da comunicação osmótica. No caso da comunicação osmótica, cada um decide se deseja parar o que está fazendo para responder a alguma pergunta ou auxiliar alguém. Isso costuma tomar poucos segundos ou, no máximo, alguns minutos. Contudo, uma tarefa de última hora, trazida de forma coercitiva ao desenvolvedor que tenta se concentrar em seu trabalho, é diferente: é uma interrupção que poderá afastá-lo do trabalho por muitos minutos ou até mesmo horas.

Sem dúvida, podem ocorrer interrupções de alta prioridade, mas são poucas as tarefas que não podem esperar algumas horas ou até mesmo alguns dias para serem realizadas. Ou seja, o importante é saber qual é a real urgência da tarefa e colocá-la em seu devido lugar na lista de prioridades. E, a não ser que seja algo realmente muito importante, a tarefa atual não deve ser interrompida. A nova tarefa deverá ficar na pilha de prioridades aguardando a finalização da tarefa atual para então ser iniciada.

Pessoas que trabalham em vários projetos ao mesmo tempo dificilmente farão algum progresso em qualquer um deles. Segundo Cockburn (2004), pode-se gastar até uma hora e meia para retomar a linha de pensamento quando se passa de um projeto a outro.

Gerentes de projeto experientes concordam que uma pessoa consegue ser efetiva em um ou dois projetos simultaneamente, mas, quando assume um terceiro projeto, ela passa a não ser efetiva nos três.

Quando um desenvolvedor está atulhado com vários projetos e atividades simultâneas, a solução gerencial é definir a lista de prioridades e qual é a atividade (ou atividades) que deve ser terminada o quanto antes. Enquanto ele estiver focado nessa atividade, as outras vão aguardar sua vez.

Quando a empresa faz rodízio de funcionários entre projetos (o que é saudável), uma das formas de manter o foco nas atividades é garantir que cada um deles fique um tempo mínimo (por exemplo, dois dias) num projeto antes de ser realocado para outro. Isso dá ao funcionário a tranquilidade de saber que seu esforço inicial para entrar no ritmo do projeto não será bruscamente interrompido antes que ele tenha oportunidade de produzir algo de valor.

4.6.6 Acesso Fácil a Especialistas

Não há dúvida de que o acesso fácil a especialistas ajuda muito o desenvolvimento de um projeto de software, uma vez que, embora os desenvolvedores sejam especializados em sistemas, não o são *naquele* que estão desenvolvendo. Infelizmente, essa característica não é das mais fáceis de se obter em um projeto.

Os usuários e clientes serão necessários antes, durante e depois do desenvolvimento. *Antes* para apresentar os requisitos e os objetivos de negócio, *durante* para esclarecer dúvidas que invariavelmente surgem ao longo do desenvolvimento, *depois* para validar o que foi desenvolvido.

Cockburn (2004) afirma que, no mínimo uma hora por semana seria essencial para que um especialista no domínio, usuário ou cliente estivesse disponível para responder às dúvidas da equipe de desenvolvimento. Mais tempo do que isso seria certamente salutar; menos, poderia levar o projeto a ter sérios problemas.

Outro problema relacionado a isso é o tempo que uma dúvida dos desenvolvedores leva para ser resolvida. Se uma dúvida demorar para ser respondida, os desenvolvedores poderão incorporar ao código sua melhor estimativa (*best shot*) e depois se esquecer disso. Assim, o sistema só vai mostrar que tem uma não-conformidade bem mais adiante, quando for liberado para uso pelo cliente.

Para evitar que dúvidas importantes demorem muito a ser respondidas é fundamental que, se o especialista não puder estar fisicamente presente no ambiente de desenvolvimento todas as horas da semana, exista uma forma de comunicação imediata com ele, como a videoconferência ou o telefone.

Cockburn (2004) também indica três principais estratégias para ter acesso fácil a especialistas:

- *Reuniões uma ou duas vezes por semana com usuário e telefonemas adicionais*: o usuário dará muitas informações importantes à equipe nas primeiras semanas. Depois, a necessidade que a equipe terá dele vai diminuir gradativamente. Um ritmo natural vai se constituindo com o usuário informando requisitos e avaliando o software desenvolvido a cada iteração. Alguns poucos telefonemas adicionais durante a semana ajudarão a equipe a não investir tempo e esforços na direção errada.
- *Um ou mais especialistas permanentemente na equipe de desenvolvimento*: essa é uma situação mais difícil de ser conseguida. As opções são colocar a equipe para trabalhar dentro da empresa-cliente ou coalocada com algum usuário.

- *Enviar desenvolvedores para trabalhar como* trainees *com o cliente por algum tempo*: por mais estranha que essa opção possa parecer, ela é bastante válida, pois os desenvolvedores terão uma visão muito clara do negócio do cliente e entenderão como o sistema que vão desenvolver poderá ajudar a melhorar seu modo de trabalho.

Uma coisa importante, nesse aspecto, é entender que não existe um único tipo de usuário. Há os clientes, que usualmente são as pessoas que pagam pelo sistema; os gerentes de alto e baixo escalão; os especialistas de domínio (aqueles que conhecem ou definem as políticas); e os usuários finais (os que efetivamente usam o sistema). É importante que a equipe de desenvolvimento tenha acesso a cada um deles no momento certo e entenda seus conceitos e necessidades.

4.6.7 Ambiente Tecnologicamente Rico

Uma equipe, para estar na zona de conforto de desenvolvimento, deve ter um *ambiente tecnologicamente rico*, não apenas linguagens de programação e ferramentas para desenhar diagramas, mas os três pilares de um bom ambiente de desenvolvimento de software: *teste automatizado*, *sistema de gerenciamento de configuração* e *integração frequente*.

Uma equipe será produtiva de fato quando conseguir fazer integrações frequentes de versões automaticamente controladas e testadas. Dessa forma, o processo de geração de novas versões de um sistema poderá levar poucos minutos e a confiabilidade nessa integração será bastante alta.

O teste automatizado não pode ser considerado propriedade essencial de um processo de desenvolvimento, porque as equipes que fazem testes manuais também conseguem produzir com qualidade. Mas a automatização do teste poupa tanto tempo e dá tanta tranquilidade aos desenvolvedores que é um movimento muito importante em direção à zona de conforto.

O teste automatizado implica que a pessoa responsável pelo teste possa, a qualquer momento, iniciá-lo e sair para fazer outra coisa enquanto ele é realizado. Não deve haver necessidade de intervenção humana no teste. Os resultados poderão até mesmo ser publicados na Web ou na intranet, de forma que todos os desenvolvedores possam acompanhar em tempo real o que está sendo testado e os resultados disso.

Além disso, deve ser possível combinar sequências de testes individuais para gerar um conjunto de teste completo para o sistema, que poderá, eventualmente, ser rodado nos fins de semana para garantir que novos defeitos não sejam inadvertidamente introduzidos.

O sistema de gerenciamento de configuração também ajuda a equipe a trabalhar com mais tranquilidade, pois caminhos que eventualmente são seguidos, mas não se mostram tão promissores quanto pareciam no início, podem ser desfeitos sem maiores consequências (Capítulo 10).

Quanto mais frequentemente a equipe fizer a integração de partes do sistema, mais rapidamente os erros serão detectados. Deixar de realizar integrações frequentes faz com que erros se acumulem e, dessa forma, comecem a se multiplicar, e um conjunto de erros é muito mais difícil de consertar do que cada um deles.

Não há uma resposta definitiva sobre qual deve ser a frequência de integração, assim como não há uma resposta única sobre a duração das iterações. Caberá à equipe avaliar a forma mais orgânica de fazer as integrações, mas sempre lembrando que postergá-las muito tempo poderá gerar problemas.

REMISSIVO DO CAPÍTULO