4 Modelos Ágeis

Este capítulo apresenta modelos que têm menos ênfase nas definições de atividades e mais ênfase na pragmática e nos fatores humanos do desenvolvimento. Os métodos ágeis, antigamente conhecidos também como processos *leves*, estão em franco desenvolvimento e várias abordagens podem ser encontradas na literatura. Este capítulo apresenta alguns métodos representativos, iniciando com *FDD* (Seção 4.1), ou *Desenvolvimento Dirigido por Funcionalidade*, o qual ainda tem certo grau de prescrição e detalhamento de atividades que não é comum a outros métodos considerados ágeis. Outro modelo semelhante é o *DSDM* (Seção 4.2), ou *Método de Desenvolvimento Dinâmico de Sistemas*. Um método um tanto mais radical em relação a fugir de prescrições e gerenciamento em função de investir na equipe para que ela possa se auto-organizar é o *Scrum* (Seção 4.3). O modelo *XP* (Seção 4.4), ou *Programação Extrema*, também é um método pouco baseado em prescrição de atividades, mas assim como *Scrum* possui regras bem definidas sobre como a equipe deve se organizar, como as iterações e reuniões devem ocorrer e até sobre como o ambiente de trabalho deve ser organizado para que a produtividade e o sucesso pessoal e empresarial sejam maximizados. *Crystal clear* (Seção 4.5) é o método mais leve de uma família de métodos de desenvolvimento Crystal que são personalizados conforme o tamanho e complexidade do projeto. Finalmente, *ASD* (Seção 4.6) ou *Desenvolvimento Adaptativo de Sistemas* é um método ágil que aplica idéias de sistemas adaptativos complexos.

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

Apesar de serem usualmente mais leves, é errado entender os métodos ágeis como modelos de processos meramente menos complexos ou simplistas. Não se trata de apenas de simplicidade, mas de focar mais nos resultados do que no processo.

Os princípios dos modelos ágeis foram claramente colocados no Manifesto Ágil (Agile, 2011)[38] e assinados por 17 pesquisadores da área, dentre os quais Martin Fowler, Alistair Cockburn e Robert Martin. O manifesto estabelece o seguinte:

> *Nós estamos descobrindo formas melhores de desenvolver software fazendo e ajudando outros a fazer. Através deste trabalho chegamos aos seguintes valores:*
>
> *a) Indivíduos e interações estão acima de processos e ferramentas.*
> *b) Software funcionando está acima de documentação compreensível.*
> *c) Colaboração do cliente está acima de negociação de contrato.*
> *d) Responder às mudanças está acima de seguir um plano.*
>
> *Ou seja, enquanto forem valorizados os itens à esquerda, os itens à direita valerão mais.*

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

[38] agilemanifesto.org/

depois que indivíduos, interações, software funcionando, colaboração do cliente e resposta às mudanças forem considerados importantes também.

Processos bem estruturados de nada adiantam se as pessoas não os seguirem. Software bem documentado de nada adianta se não satisfaz os requisitos, ou se não funciona. E assim por diante.

O manifesto ágil é complementado por doze princípios, os quais são citados abaixo:

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

Um conjunto significativo de modelos atuais é considerado ágil. Alguns são até muito diferentes entre si, mas praticamente todos consideram os princípios acima como pontos fundamentais em seu funcionamento. Nas próximas subseções são apresentados os modelos ágeis mais conhecidos e representativos: FDD, DSDM, *Scrum*, XP, *Crystal clear* e ASD.

4.1 FDD - *Feature-Driven Development*

FDD ou *Feature-Driven Development*[39] (*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 uma evolução de um processo mais antigo (Coad, de Luca, & Lefebvre, 1997).

Duas atualizações importantes do modelo foram apresentadas por Palmer e Mac Felsing (2002) e por Anderson (2004).

FDD possui apenas duas grandes fases:

[39] www.featuredrivendevelopment.com/

a) *Concepção e planejamento*: que implica em pensar um pouco antes de começar a construir, tipicamente 1 a 2 semanas.
b) *Construção*: desenvolvimento iterativo do produto em ciclos de 1 a 2 semanas.

A fase de concepção e planejamento possui 3 disciplinas (processos):

a) *DMA – Desenvolver Modelo Abrangente*, onde se preconiza o uso de modelagem orientada a objetos.
b) *CLF – Construir Lista de Funcionalidades*, onde se pode aplicar a decomposição funcional para identificar as funcionalidades que o sistema deve disponibilizar.
c) *PPF – Planejar Por Funcionalidade*, onde o planejamento dos ciclos iterativos é feito em função das funcionalidades identificadas.

Já a fase de construção incorpora duas disciplinas (processos):

a) *DPF – Detalhar Por Funcionalidade*, que corresponde a realizar o *design* orientado a objetos do sistema.
b) *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 pode ser representada conforme a Figura 4-1.

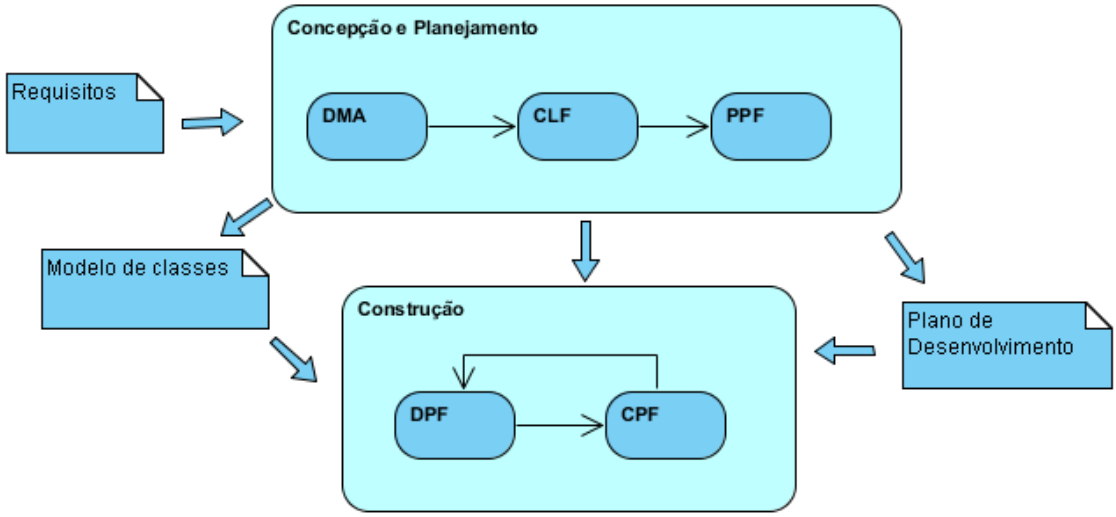


Figura 4-1: Modelo FDD.

4.1.1 DMA – Desenvolver Modelo Abrangente

A primeira fase 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.

Este modelo de classes ou modelo conceitual passará depois a ser refinado na disciplina DPF (Detalhar Por Funcionalidade).

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

As atividades individuais que compõe esta disciplina são as seguintes:

a) *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.
b) *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.
c) *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.
d) *Desenvolver o modelo*. É uma atividade obrigatória sob responsabilidade das equipes de modelagem específicas. Grupos de não mais de 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.
e) *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 poderá afetar a forma do modelo geral do negócio, que deve então ser refinado.

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

As saídas esperadas desse conjunto de atividades são:

a) O *modelo conceitual*, apresentado como um diagrama de classes e suas associações.
b) *Métodos* e *atributos* eventualmente identificados para as classes.
c) *Diagramas de sequência* ou *máquina de estados* para as situações que exigirem este tipo de descrição.
d) *Comentários* sobre o modelo para indicar porque determinadas decisões de *design* foram tomadas ao invés de outras.

As técnicas para construir estes diagramas de classes, sequência e máquina de estados podem ser encontradas nos capítulos 2 e 7 de Wazlawick (2011).

4.1.2 CLF – Construir Lista de Funcionalidades

Esta disciplina vai identificar as funcionalidades que satisfazem aos requisitos. A equipe vai decompor o domínio em áreas de negócio, conforme a disciplina DMA. As áreas de negócio serão decompostas em atividades de negócio e estas, por sua vez em passos de negócio

(Figura 4-2). O processo é relativamente parecido com uma análise de casos de uso, onde se teria áreas de negócio (*pacotes*), *casos de uso* como atividades de negócio e *passos* de casos de uso expandidos como passos de negócio.

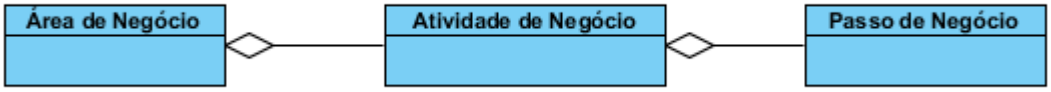


Figura 4-2: Estrutura conceitual da lista de funcionalidades.

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

A disciplina 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 líder da disciplina anterior).

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

a) *Áreas de negócio*, oriundas das atividades de DMA. Por exemplo, “vendas”.
b) *Atividades de negócio*, que são a decomposição funcional das áreas de negócio. Por exemplo, registrar pedido, faturar pedido, registrar pagamento de venda, etc.
c) *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” (Não é necessário usar as barras, que foram colocadas apenas para indicar a estrutura das frases).

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 funcionalidades mais abrangentes. Por outro lado, quando uma funcionalidade parece levar mais do que duas semanas para ser desenvolvida (por uma pessoa), então 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, cliente e especialistas de domínio.

As saídas dessa atividade são as seguintes:

a) Lista de áreas de negócio.

b) Para cada área, uma lista de atividades de negócio dentro da área.
c) Para cada atividade, uma lista de passos de atividade ou funcionalidades que permite realizar a atividade.

4.1.3 PPF – Planejar por Funcionalidade

Esta disciplina, ainda na primeira fase do FDD visa gerar o plano de desenvolvimento para a fase seguinte que é formada por ciclos iterativos. O planejamento vai indicar quais atividades de negócio da lista definida em CLF serão implementadas quando. O planejador deve levar em consideração os seguintes aspectos para agrupar e priorizar as atividades de negócio:

a) *A complexidade das funcionalidades*. Em função do risco, as atividades com funcionalidades de maior complexidade devem ser tratadas primeiro.
b) *As 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.
c) *A 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 líder, ou seja, cada um se responsabiliza por um subconjunto das classes. Então, ao fazer a divisão da carga de trabalho já se estará também definindo, via de regra, a posse das classes.

A entrada para as atividades desta disciplina consiste na *lista de funcionalidades* construída em CLF.

São quatro as atividades da disciplina PPF:

a) *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 líder.
b) *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, apenas, ou seja, prazos mensais. A sequência de desenvolvimento deve ser construída levando em consideração os seguintes fatores:
    a. Priorizar as atividades com funcionalidades mais complexas ou de alto risco.
    b. Alocar juntas atividades ou funcionalidades dependentes umas das outras se possível.
    c. Considerar marcos externos, quando for o caso, para criação de *release*s.
    d. Considerar a distribuição de trabalho entre os proprietários das classes.
c) *Atribuir atividades de negócio aos* programadores líder. É uma atividade obrigatória de responsabilidade da equipe de planejamento. Essa atividade vai determinar quais

programadores líder serão proprietários de quais atividades de negócio. Essa atribuição de propriedade deve ser feita considerando os seguintes critérios:

   a. Dependência entre as funcionalidades e as classes das quais os programadores líder já são proprietários.
   b. A sequência de desenvolvimento.
   c. A complexidade das funcionalidades a serem implementadas em função da carga de trabalho alocada aos programadores líder.

d) *Atribuir classes aos desenvolvedores.* É uma atividade obrigatória de responsabilidade da equipe de planejamento. A equipe de planejamento deve atribuir a propriedade das classes aos desenvolvedores. Essa atribuição é baseada em:

   a. Distribuição de carga de trabalho entre os desenvolvedores.
   b. Complexidade das classes (priorizar as mais complexas ou de maior risco).
   c. Intensidade de uso das classes (priorizar as classes altamente usadas).
   d. Sequência de desenvolvimento.

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

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

a) Prazos (mês e ano) para a conclusão do desenvolvimento referente a cada uma das atividades de negócio.
b) Atribuição de programadores líder a cada uma das atividades de negócio.
c) 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).
d) Lista dos desenvolvedores e das classes das quais eles são proprietários.

4.1.4 DPF – Detalhar por Funcionalidade

A disciplina DPF (Detalhar por Funcionalidade) é a primeira executada na fase iterativa do FDD. Ela basicamente consiste em produzir o *design* de implementação da funcionalidade o qual, usualmente, é realizado por diagramas de sequência ou comunicação (ver Wazlawick, 2011, capítulo 9).

A atividade de detalhamento é realizada para cada funcionalidade identificada dentro das atividades de negócio. A atribuição do trabalho aos desenvolvedores é feito 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 desta disciplina necessitam que o planejamento da etapa anterior tenha sido concluído.

As atividades de DPF são as seguintes:

a) *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 este pacote de funcionalidades está correntemente sendo trabalhado.

b) *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.
c) *Estudar a documentação de referência*. É uma atividade opcional de responsabilidade da equipe de funcionalidades. Também, dependendo da complexidade da funcionalidade poderá ser necessário que a equipe estude documentos disponíveis como relatórios, desenhos de telas, padrões de interface com sistemas externos, etc.
d) *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 (por exemplo, uso do padrão *stateless* ou *statefull*, etc.).
e) *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, e disponibiliza essa cópia para a equipe de funcionalidades. A equipe de funcionalidade terá acesso compartilhado a essa cópia, mas o restante do pessoal não, até que a cópia seja salva como uma nova versão das classes no sistema de controle de versões. A equipe de funcionalidades então adiciona os métodos, atributos, associações e novas classes que forem necessárias.
f) *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 e a declaração dos métodos, incluindo tipos de parâmetros e retornos, exceções e mensagens.

A verificação do produto destas 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. 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 precisam ser efetivamente implementados ainda), e a nova versão das classes é gravada (*commit*) no sistema de controle de versões.

A saída das atividades é um *pacote de design* inspecionado e aprovado. Este pacote consiste de:

a) Uma capa com comentários que descreve o pacote de forma suficientemente clara.
b) Os requisitos abordados na forma de atividades e/ou funcionalidades.
c) Os diagramas de sequência.

d) Os projetos alternativos (se houver).
e) O modelo de classes atualizado.
f) As interfaces de classes geradas.
g) A lista de tarefas para os desenvolvedores, gerada em função destas atividades.

4.1.5 CPF – Construir por Funcionalidade

Esta disciplina, 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 disciplina 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 um *build*.

A entrada para estas atividades é o *pacote de design* gerado na disciplina anterior.

As atividades envolvidas são:

a) *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.
b) *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 (Seção 11.4.2). Mas ela é sempre coordenada pelo programador líder, e pode ser feita antes ou depois dos testes de unidade.
c) *Teste de unidade*. É uma atividade obrigatória sob responsabilidade da equipe de funcionalidades (Seção 13.2.1). 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.
d) *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 a *build* (Capítulo 10), não sem antes passar pelos testes de integração.

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

Os resultados ou saídas destas atividades são:

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

4.1.6 Ferramentas para FDD

Existem ferramentas disponíveis específicas para uso com o método FDD. Entre elas:

a) *CaseSpec*[40]. Uma ferramenta proprietária, mas com *free trial*, para gerenciamento de requisitos ao longo de todo o ciclo de vida.
b) *TexExcel DevSuite*[41]. Um conjunto de ferramentas específicas para aplicar FDD.
c) *FDD Tools Project*[42]. Um projeto que visa criar ferramentas gratuitas de código aberto para FDD.

4.2 DSDM – *Dynamic Systems Development Method*

Dynamic Systems Development Method (*DSDM*)[43] também é um modelo ágil baseado em desenvolvimento iterativo e incremental e com participação ativa do usuário. O método é uma evolução do *Rapid Application Development* (*RAD*) (Martin, 1990), que por sua vez, é um sucessor de *Prototipação Rápida* (Seção 3.9).

O DSDM possui três fases:

a) *Pré-projeto*. Nesta fase o projeto é identificado e negociado, seu orçamento é definido e um contrato assinado.
b) *Ciclo de vida.* O ciclo de vida inicia com uma fase de análise de viabilidade e de negócio. Depois entra em ciclos iterativos de desenvolvimento.
c) *Pós-projeto*. Equivale ao período que normalmente é considerado como operação ou manutenção. Nesta fase a evolução do software é vista como uma continuação do processo de desenvolvimento, podendo, inclusive, retomar fases anteriores se necessário.

A fase de ciclo de vida, por sua vez, subdivide-se em estágios:

a) Análise de viabilidade.
b) Análise de negócio (por vezes aparece junto com a anterior).
c) Iteração do modelo funcional.
d) Iteração de elaboração e construção.
e) Implantação.

DSDM fundamenta-se no *Princípio de Pareto*[44], ou Princípio 80/20, assim, o DSDM procura iniciar pelo estudo e implementação dos 20% dos requisitos que serão mais determinantes para o sistema como um todo. Esta prática é compatível com a prática de abordar inicialmente requisitos mais complexos ou de mais alto risco, presente em outros modelos como UP.

Outro aspecto do DSDM é a *objetividade*, que afirma que a gerência de riscos deve se focar nas funcionalidades a serem entregues, em detrimento de outros fatores, como documentos ou o processo em si.

A filosofia DSDM pode ser resumida pelos seguintes princípios:

[40] www.casespec.net/
[41] www.techexcel.com/solutions/alm/fdd.html
[42] fddtools.sourceforge.net/
[43] www.dsdm.org/
[44] O princípio foi originalmente formulado pelo economista italiano do Século XIX, Alfredo Pareto, que observou que em muitas situações, 80% das consequencias são devidas a 20% das causas. Em engenharia de requisitos pode-se verificar que, em geral, 80% do sistema advém de 20% dos requisitos.

a) *Envolvimento* do usuário durante todo o tempo do projeto.
b) *Autonomia* dos desenvolvedores para tomar decisões sem a necessidade de consultar comitês superiores.
c) *Entregas frequentes* de *release*s suficientemente boas são o melhor caminho para se conseguir entregar um bom produto no final. Postergar entregas aumenta o risco de que o produto final não seja o que o cliente deseja.
d) *Eficácia* das entregas na solução de problemas de negócio. Como as primeiras entregas vão se concentrar nos requisitos mais importantes, elas serão mais eficazes para o negócio.
e) *Feedback dos usuários* como forma de realimentar o processo de desenvolvimento iterativo e incremental.
f) *Reversibilidade* de todas as ações realizadas durante o processo de desenvolvimento.
g) *Previsibilidade* para que o escopo e objetivos das iterações sejam conhecidos antes de iniciar.
h) *Ausência de testes no escopo*, já que o método considera a realização de testes como uma atividade fora do ciclo de vida.
i) *Comunicação* de boa qualidade entre todos os envolvidos é fundamental para o sucesso de qualquer projeto.

O DSDM não é recomendável para projetos onde a segurança é um fator crítico, pois a necessidade de testes exaustivos deste tipo de sistema conflita com os objetivos de custo e prazo do DSDM.

O DSDM é bem mais fortemente baseado em documentação do que *Scrum* ou XP, o que até o deixa mais parecido com o Processo Unificado do que com métodos ágeis. Uma das características do DSDM que o diferencia do Processo Unificado, porém, é o fato de que ele não preconiza o uso de nenhuma técnica específica. Ele é, de fato, um *framework* de processo, no qual os participantes poderão desenvolver suas atividades utilizando suas técnicas preferidas.

A versão corrente do método, conhecida como *DSDM Atern* pode ser visualizada gratuitamente no site do DSDM Consortium[45]. Segundo o DSDM Consortium, ela foi projetada para ser compatível com o método de gerência de projetos Prince2 (Seção 9.3), CMMI (Seção 12.3) e ISO9001 (Seção 12.1).

4.2.1 Análise de Viabilidade

Na etapa de *análise de viabilidade* será verificado se é viável usar o DSDM para o desenvolvimento do sistema e se existem outros fatores impeditivos eventuais. Também é avaliado nesta etapa se o projeto sendo proposto realmente atenderá aos objetivos de negócio da empresa. Além disso, os primeiros riscos do projeto são levantados. A maioria dessas atividades é realizada por grupos de trabalho. Nesta etapa também é verificado se existem especialistas de domínio disponíveis, usualmente na empresa cliente, para participar dos grupos de trabalho.

Nesta etapa são gerados os seguintes artefatos:

[45] www.dsdm.org/dsdm-atern

a) Relatório de viabilidade.
b) Protótipo de viabilidade.
c) Plano de desenvolvimento (Seção 6.4).
d) Controle de risco (Capítulo 8).

4.2.2 Análise de Negócio

Se o projeto for aprovado na análise de viabilidade então ele passa para a etapa de análise de negócio. Na *análise de negócio* são estudadas as características do negócio e as possíveis tecnologias a serem utilizadas. A análise de negócio é semelhante à análise de requisitos tradicional, mas ela enfatiza o trabalho em equipe. Os grupos de trabalho devem envolver diversos tipos de especialistas, entre analistas, *designers*, usuários e especialistas no domínio.

Outra característica típica do DSDM nesta fase é o uso da técnica de *timeboxing*, que consiste em fixar previamente o prazo e orçamento a serem utilizados. Feito isso, a variável que resta é o número de requisitos que poderão ser tratados. Desta forma, os requisitos menos importantes poderão ficar de fora, caso o tempo ou orçamento previamente definidos sejam insuficientes. O princípio de Pareto indica que isso não será um problema, já que 20% dos requisitos devem dar origem a 80% do sistema. Assim, se os requisitos menos importantes forem deixados de lado, sua influência final no sistema será muito pequena.

Para que o Princípio de Pareto seja aplicado à análise de requisitos, porém, é necessário que os requisitos sejam priorizados, ou seja, deve-se determinar quais são os mais e os menos importantes. DSDM preconiza o uso da técnica MoSCoW (Clegg & Barker, 2004), que é um acrônimo para:

a) *Must*: requisitos que devem estar necessariamente de acordo com as regras de negócio.
b) *Should*: requisitos que devem ser considerados tanto quanto possível, mas que não tem um impacto decisivo no sucesso do projeto.
c) *Could*: requisitos que podem ser incluídos desde que não afetem negativamente os objetivos de tempo e orçamento do projeto.
d) *Would*: requisitos que podem ser incluídos se sobrar tempo.

Nesta etapa também é produzida a *arquitetura inicial de sistema*, indicando seus principais componentes como classes, pacotes, componentes, nodos de processamento, servidores, meios de comunicação, etc.

Finalmente, nesta etapa a lista de riscos do projeto é atualizada de acordo com as descobertas atuais.

4.2.3 Iteração do Modelo Funcional

Os requisitos identificados na etapa anterior serão convertidos em um primeiro protótipo funcional na etapa de iteração do modelo funcional. Essa etapa é composta por quatro atividades:

a) *Identificar o protótipo funcional*, ou seja, devem ser identificadas as funcionalidades que serão implementadas no ciclo atual. A base para determinação dessas

funcionalidades é o resultado da etapa de análise de negócio, ou seja, o modelo funcional.

b) *Agenda*, que determina quando e como cada uma das funcionalidades será implementada.
c) *Criação do protótipo funcional*, que consiste na implementação preliminar das funcionalidades definidas de acordo com a agenda.
d) *Revisão do protótipo funcional*, que é feita a partir tanto de revisões da documentação quanto por avaliação do usuário final (semelhante ao teste de aceitação, visto na Seção 13.2.4). Esta atividade deve produzir o *documento de revisão do protótipo funcional*.

A atividade de *identificação do protótipo funcional* ainda incorpora quatro subatividades:

a) *Análise de requisitos*. Os requisitos do protótipo atual são analisados e revisados de acordo com a lista de prioridades criada na fase de análise de negócio ou na iteração anterior.
b) *Listar requisitos funcionais da iteração atual*. Selecionar na lista de prioridades os requisitos que serão abordados na iteração atual.
c) *Listar requisitos não funcionais da iteração atual*. Selecionar os requisitos não funcionais que serão abordados na iteração atual.
d) *Criar um modelo funcional*. Implica na criação de modelos funcionais do sistema preferencialmente usando notação gráfica.

A atividade de *agenda* também se subdivide em subatividades:

a) *Determinar tempo.* Segundo a técnica de *timeboxing*, o tempo disponível para as atividades da iteração deve ser predeterminado.
b) *Definir desenvolvimento.* O plano de prototipagem deve incluir todas as atividades necessárias para o desenvolvimento do protótipo no tempo disponível.
c) *Confirmação da agenda.* Todos os participantes devem concordar com o plano de prototipagem.

A atividade de *criação do protótipo funcional* também possui três subatividades:

a) *Investigar*. Deve-se estudar detalhadamente os requisitos, bem como o modelo funcional para definir um plano de implementação para o protótipo da iteração atual.
b) *Refinar*. Implementar e refinar o protótipo da iteração atual.
c) *Consolidar*. Consolidar ou integrar o protótipo da iteração atual com o protótipo das iterações anteriores.

Finalmente, a atividade de *revisão do protótipo* se subdivide em duas subatividades:

a) *Testar protótipo*. O artefato *registro de teste* deve estar sendo atualizado e considerado em todas as etapas do DSDM. Nesta subatividade ele será útil para que se saiba quais são os testes importantes pelos quais o protótipo deve passar para ser aceito.

b) *Revisão final*. Em função dos testes e do *feedback* dos usuários será criado o *documento de revisão de prototipagem*, o qual será usado para revisar a lista de requisitos e sua priorização, bem como a lista de riscos.

O modelo funcional e protótipo funcional são os principais artefatos produzidos nesta etapa. O protótipo funcional é usado especialmente para que o usuário possa avaliar se os requisitos implementados são realmente os que ele necessita. A partir da avaliação do protótipo funcional a lista de requisitos é revisada, bem como sua priorização e a lista de riscos.

4.2.4 Iteração de *Design* e Construção

A *iteração de design e construção* é uma etapa que vai tratar da integração das funcionalidades já aprovadas em um sistema que satisfaça aos interessados. Além disso, essa fase vai comportar uma verificação mais detalhada dos requisitos não funcionais, que poderiam ter sido relegados a segundo plano no protótipo funcional. Essa fase também comporta quatro atividades:

a) *Identificar o modelo de design*. Deve-se aqui identificar os requisitos funcionais e não funcionais que devem estar no sistema final. As *evidências de teste* obtidas a partir do protótipo da etapa anterior serão usadas para criar a *estratégia de implementação*.
b) *Agenda*. A agenda define quando e como serão implementados os requisitos.
c) *Criação do protótipo de design*. Aqui é criado um protótipo do sistema que pode ser utilizado pelos usuários finais e também para efeito de teste.
d) *Revisão do protótipo*. O protótipo assim construído deve ser testado e revisado gerando dois artefatos: *documentação para usuário* e *evidências de teste.*

A *prototipagem* é um dos pontos fundamentais do DSDM. O primeiro protótipo (funcional), deve ser desenvolvido para mostrar as principais funções do sistema, de forma que o cliente possa ter rapidamente uma visão daquilo que será implementado. O segundo protótipo (de *design*) deve incluir progressivamente todas as características não funcionais não incluídas no primeiro protótipo, bem como todas as modificações solicitadas pelo cliente durante o processo de revisão do protótipo.

4.2.5 Implantação

A etapa de *implantação* tem como objetivo entregar o sistema aos usuários finais. Essa etapa é composta pelas seguintes atividades:

a) *Orientações e aprovação do usuário*. Os usuários finais aprovam o protótipo final como sistema definitivo a partir de seu uso e da observação da documentação fornecida.
b) *Treinamento*. Os usuários finais são treinados para o uso do sistema. *Usuários treinados* é considerado o artefato de saída desta atividade.
c) *Implantação*. Quando o sistema for implantado e liberado para os usuários finais é obtido o artefato *sistema entregue.*
d) *Revisão de Negócio*. Nesta atividade, o impacto do sistema sobre os objetivos de negócio é avaliado. Dependendo do resultado, o projeto passa para um ciclo posterior ou então reinicia o mesmo ciclo a fim de refinar e melhorar os resultados.

4.2.6 Papeis no DSDM

O DSDM tem um conjunto muito peculiar de *papeis* que lhe são próprios. Como em outros processos, uma mesma pessoa pode ocupar mais de um papel, mas é importante que cada um saiba quais as responsabilidades que lhe cabem. Os papeis são os seguintes:

a) *Campeão do projeto*. Funciona como um gerente executivo. Deve ser uma pessoa com capacidade de administrar prazos e tomar decisões, já que a ela caberão sempre as decisões finais em caso de conflito.
b) *Visionário*. Ele é o que tem a missão de saber se os requisitos iniciais estão adequados para que o projeto inicie. O visionário também funciona como uma espécie de engenheiro do processo, porque ele tem a responsabilidade de manter as atividades de acordo com o DSDM.
c) *Intermediador*. Ele é o que faz a interface da equipe de desenvolvimento com os clientes, usuários e especialistas de domínio. Sua responsabilidade é buscar e trazer as informações adequadas e necessárias para a equipe.
d) *Anunciante*. É qualquer usuário que, por representar um importante ponto de vista, deve trazer informações frequentemente para a equipe, possivelmente diariamente.
e) *Gerente de projeto*. É responsável por manter as atividades nos prazos, com o orçamento definido e com a qualidade necessária.
f) *Coordenador técnico*. É responsável pelo projeto da arquitetura do sistema e seus aspectos técnicos.
g) *Líder de equipe*. É um desenvolvedor com a função especial de motivar e manter a harmonia de seu grupo.
h) *Desenvolvedor*. Trabalha para transformar requisitos e modelos em código executável.
i) *Testador*. É o responsável pelos testes do sistema.
j) *Escrivão*. É responsável por tomar notas para registro de requisitos identificados, bem como de decisões de *design* tomadas ao longo do processo de desenvolvimento.
k) *Facilitador*. Disponibiliza o ambiente de trabalho e avalia o progresso das diversas equipes.

Outros papeis típicos de processo de desenvolvimento podem ser usados quando necessário, como por exemplo, *designer* de interface, gerente de banco de dados, especialista em segurança, etc.

Além disso, como em outros modelos, uma mesma pessoa pode estar em diferentes papeis, sem prejuízo, caso a equipe seja pequena.

4.3 *Scrum*

Scrum é um modelo ágil para gestão de projetos de software. No *Scrum* um dos conceitos mais importantes é o *sprint*, que consiste em um ciclo de desenvolvimento, usualmente de duas semanas a um mês.

A concepção inicial do *Scrum* deu-se na indústria automobilística (Takeuchi & Nonaka, 1986), e o modelo pode ser adaptado a várias outras áreas diferentes da produção de software.

Na área de desenvolvimento de software, *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.

4.3.1 Perfis

Há três perfis importantes no Modelo *Scrum*:

a) O *Scrum master*, que não é um gerente no sentido dos modelos prescritivos. O *Scrum master* não é um líder, já que as equipes são auto-organizadas, mas um facilitador (pessoa que conhece bem o modelo) e resolvedor de conflitos.
b) O *product owner*, ou seja, a pessoa responsável pelo projeto em si. O *product owner* tem, entre outras atribuições, a de indicar quais são os requisitos mais importantes a serem tratados em cada *sprint*. O *product owner* é o responsável pelo *ROI* (*Return Of Investment*), e também por conhecer e avaliar as necessidades do cliente.
c) O *Scrum team*, que é a equipe de desenvolvimento. Essa equipe não é necessariamente dividida em papeis como analista, *designer* e programador, mas todos interagem para desenvolver o produto em conjunto. Usualmente são recomendadas equipes de 6 a 10 pessoas.

No caso de projetos muito grandes é possível aplicar o conceito de *Scrum of Scrums*[46], onde 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.

4.3.2 Product Backlog

As funcionalidades a serem implementadas em cada projeto (requisitos ou histórias de usuário) são mantidas em uma lista chamada de *product backlog*.

Um dos princípios do manifesto ágil é usado aqui: *adaptação* ao invés de planejamento. O *product backlog* não precisa então ser completo no início do projeto. Pode-se iniciar apenas com as funcionalidades mais evidentes, aplicando o princípio de Pareto, para depois, à medida que o projeto avança tratar novas funcionalidades que vão sendo descobertas. Isso, porém, não significa fazer um levantamento inicial excessivamente superficial. Deve-se tentar junto ao cliente obter o maior número possível de informações sobre suas necessidades. Aquelas que efetivamente surgirem nessa interação terão maior relevância, possivelmente, do que outras que forem descobertas mais adiante.

A Figura 4-3 apresenta um exemplo de *product backlog*.

[46] www.scrumalliance.org/articles/46-advice-on-conducting-the-scrum-of-scrums-meeting (consultado em 19/02/2012).

***Product Backlog* (exemplo)**					
ID	**Nome**	**Imp.**	**PH**	**Como demonstrar**	**Notas**
1	Depósito	30	5	Logar, abrir página de depósito, depositar R$10,00, ir para página de saldo e verificar que ele aumentou em R$10,00.	Precisa de um diagrama de sequência UML. Não há necessidade de se preocupar com criptografia por enquanto.
2	Ver extrato	10	8	Logar, clicar em "transações". Fazer um depósito. Voltar para transações, ver que o depósito apareceu.	Usar paginação para evitar consultas grandes ao BD. *Design* similar para visualizar página de usuário.

Figura 4-3: Exemplo de *product backlog*[47].

Segundo Kniberg (2007)[48], vários outros campos poderiam ser adicionados, mas na sua experiência, os seis campos representados na Figura 4-3 acabam sempre sendo importantes:

a) *Id*. Um *identificador* numérico (contador), considerado importante para que a equipe não perca a trilha da história de usuário se eventualmente ela mudar de nome ou descrição.
b) *Nome*. Um *nome* curto que representa mnemonicamente a história de usuário.
c) *Imp.* A *importância* da história de usuário. Números mais altos indicam importância maior. Não há limite. A relação entre os números não deve ser interpretada como uma proporção de importância. Se uma história tem importância 10 e outra tem importância 50 isso só quer dizer que a segunda é mais importante, mas não significa que seja 5 vezes mais importante.
d) *PH.* Uma estimativa de esforço necessário para transformar a história em software. O valor é dado em *pontos de histórias* (*PH*). Mais detalhes na seção 7.6.
e) *Como demonstrar.* Uma descrição do que teria que ser possível fazer para que se considerasse que a história foi efetivamente implementada. Este campo poderia ser considerado como o código de alto nível para o módulo de teste da história.
f) *Notas.* Quaisquer outras informações julgadas importantes para a implementação da história de usuário.

4.3.3 Sprint

O *sprint* é o ciclo de desenvolvimento de poucas semanas de duração sobre o qual se estrutura o *Scrum*.

No início de cada *sprint* é feito um *sprint planning meeting*, no qual a equipe prioriza os elementos do *product backlog* a serem implementados, e transfere estes elementos do *product backlog* para o *sprint backlog*, ou seja, a lista de funcionalidades a serem implementadas no ciclo que se inicia.

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 forem descobertas, serão abordadas em *sprints* posteriores.

[47] Adaptado de: Kniberg (2007, p. 10).
[48] www.metaprog.com/csm/*Scrum*AndXpFromTheTrenches.pdf

Pode-se dizer que os dois *backlogs* têm naturezas diferentes:

a) O *product backlog* apresenta requisitos de alto nível, bastante voltados às necessidades diretas do cliente.
b) Já o *sprint backlog* apresenta uma visão desses requisitos de forma mais voltada à maneira como a equipe vai desenvolvê-los.

Durante o *sprint*, cabe ao *product owner* manter o *sprint backlog* atualizado, indicando as tarefas já concluídas e as ainda por concluir, preferencialmente mostradas em um gráfico atualizado diariamente e à vista de todos. Um exemplo de um quadro de andamento de atividades é apresentado na Figura 4-4.

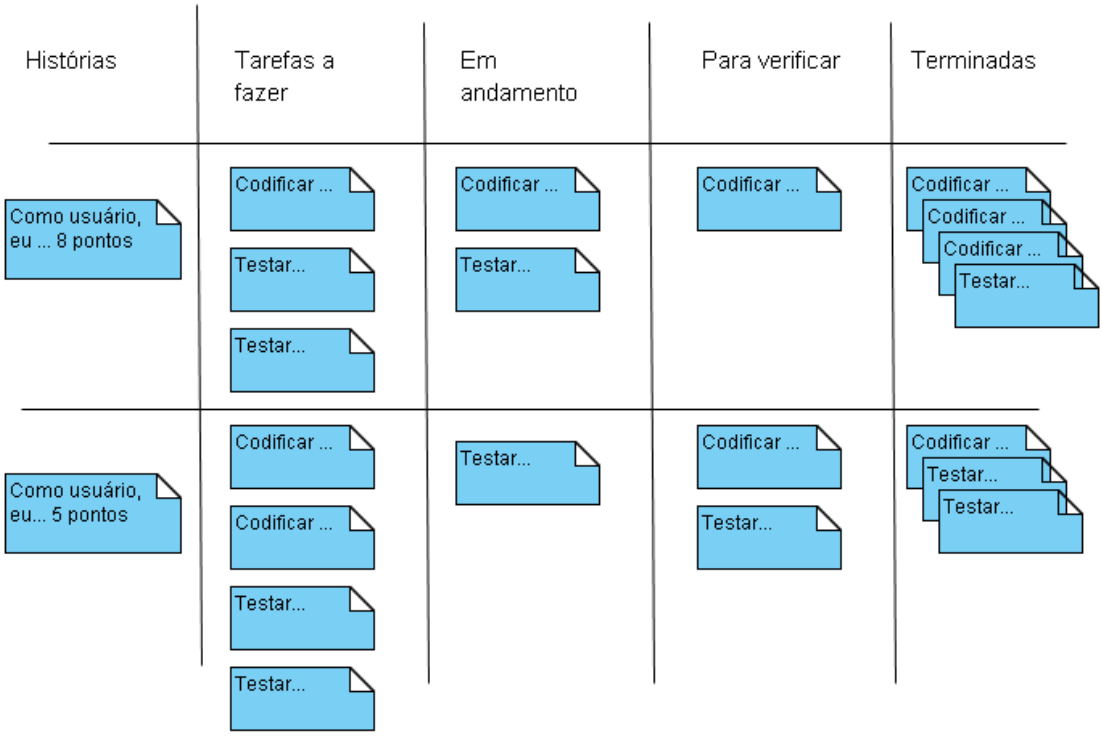


Figura 4-4: Um típico quadro de andamento de tarefas de um *sprint*.

Existem ferramentas como *Virtual Scrum Board*[49] que permitem editar estes quadros eletronicamente.

O quadro de andamento 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 constitui o *sprint backlog*, e ela é usada para inicializar a segunda coluna "tarefas a fazer" do quadro de andamento. À medida que as atividades vão sendo feitas, verificadas e terminadas, as fichas correspondentes vão sendo movidas para a coluna da direita.

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

[49] www.*download*plex.com/Windows/Business/Project-Management/virtual-*scrum*-board_220058.html

colocadas na coluna de tarefas por fazer. Porém, novas histórias de usuário não podem ser adicionadas durante o sprint.

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

O diagrama *sprint burndown* consiste basicamente de duas linhas: a primeira indica a quantidade de trabalho por fazer e a segunda indica a quantidade de trabalho feito ao longo do tempo. O gráfico é atualizado todo o dia do início até o final do *sprint*, e espera-se, numa situação ideal que seja semelhante ao da Figura 4-5.

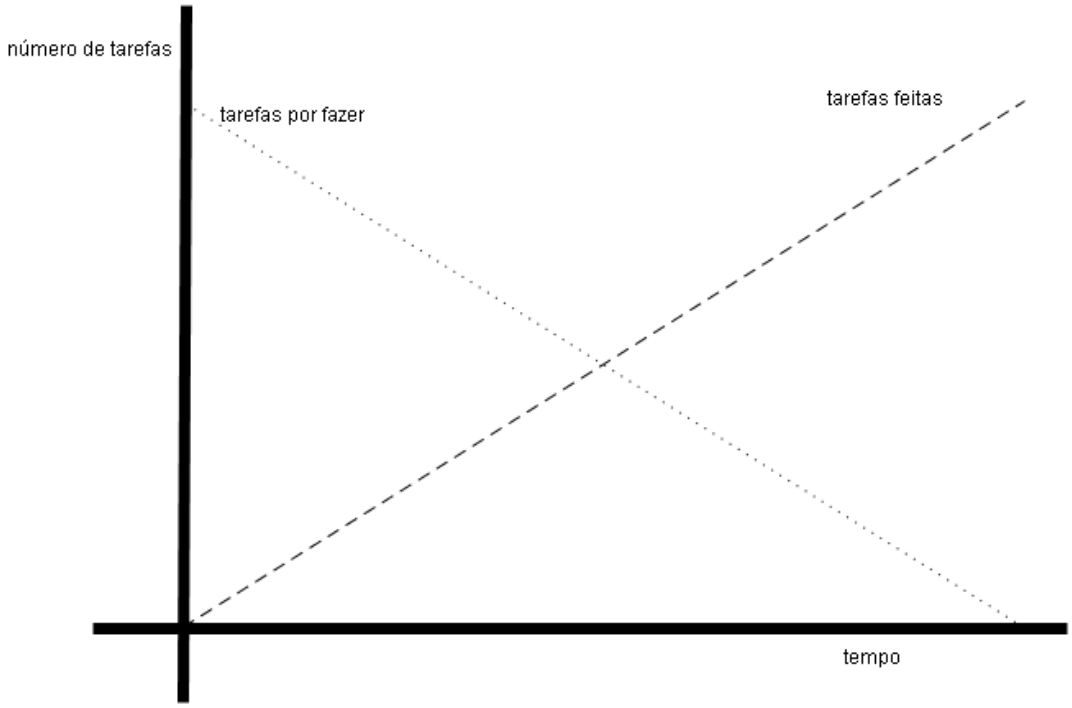


Figura 4-5: Um *sprint burndown* ideal.

Porém, quando as histórias de usuário não foram bem compreendidas e/ou as tarefas não foram bem planejadas, ou ainda, quando a equipe atribui-se tarefas extras, além das inicialmente previstas, o *sprint burndown* acaba ficando com uma aparência semelhante ao gráfico da Figura 4-6.

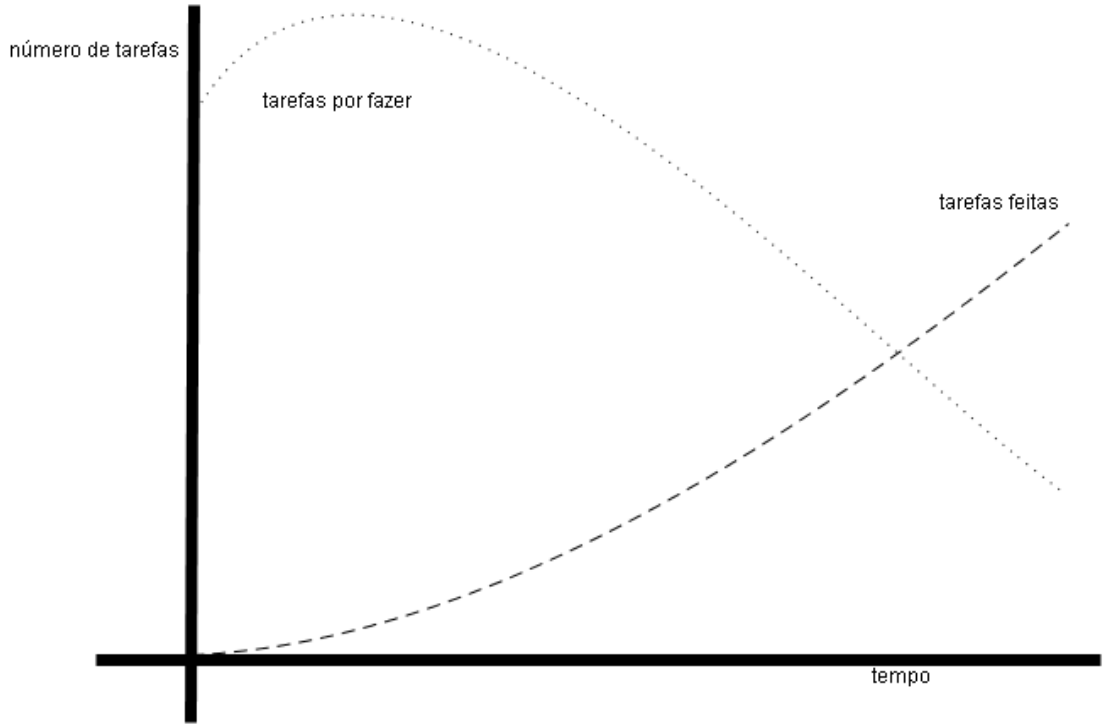
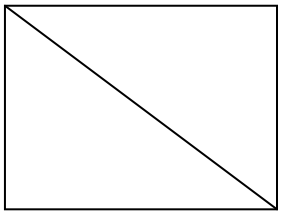


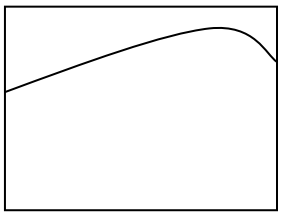
Figura 4-6: Um *sprint burndown* onde tarefas adicionais são incluídas após o início do *sprint*.

Kane Mar (2006)[50] analisa as linhas de tarefas por fazer, identificando sete tipos de comportamentos de equipes conforme seus *sprint burndown*:

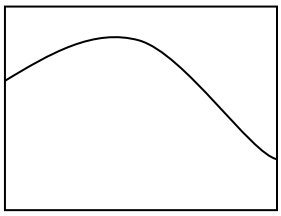
Fakey-fakey, caracterizado por uma linha reta e regular, indicando que provavelmente a equipe não está sendo muito honesta porque o mundo real é bem complexo e dificilmente projetos se comportam com tanta regularidade.



Late-learner, indicando um acúmulo de tarefas até perto do final do *sprint*. É típico de equipes iniciantes que ainda estão tentando aprender o funcionamento do *Scrum*.

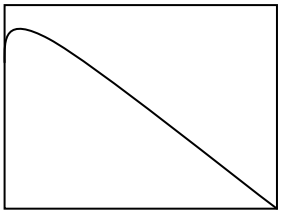


Middle-learner, indicando que a equipe pode estar amadurecendo e trazendo para mais cedo as atividades de descoberta e, especialmente, as necessidades de testes.

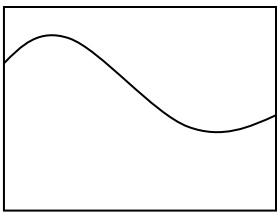


[50] kanemar.com/2006/11/07/seven-common-*sprint-burndown*-graph-signatures/

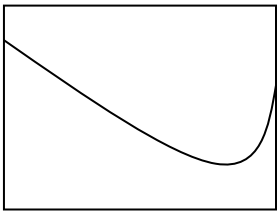
Early-learner, indicando uma equipe que procura logo no início do *sprint* descobrir todas as necessidades e posteriormente desenvolve-las gradualmente até o final do ciclo.



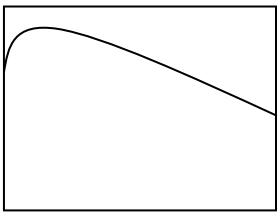
Plateau, indicando uma equipe que tenta balancear o aprendizado cedo e tardio, o que acaba levando o ritmo de desenvolvimento a um platô. Inicialmente fazem bom progresso, mas não conseguem manter o ritmo até o final do *sprint*.



Never-never, indicando uma equipe que acaba tendo surpresas desagradáveis no final de um *sprint*.



Scope increase, indicando uma equipe que percebe um súbito aumento na carga de trabalho por fazer. Usualmente a solução nestes casos é tentar renegociar o escopo da *sprint* com o *product owner*, mas não se descarta também uma finalização da *sprint* para que seja feito um replanejamento do *product backlog*.



Ao final de cada *sprint* a equipe deve fazer um *sprint review meeting* (ou *sprint demo*) para verificar o que foi feito e, a partir daí, partir para uma nova *sprint*. O *sprint review meeting* é a demonstração e avaliação do produto do *sprint*.

Outra reunião que pode ser feita ao final de uma *sprint* é a *sprint retrospective*, cujo objetivo é avaliar a equipe e os processos (impedimentos, problemas, dificuldades, ideias novas etc.).

4.3.4 Daily Scrum

Mais importante ainda, o modelo sugere que a equipe realize uma reunião diária, chamada *daily scrum*, onde o objetivo é que cada membro da equipe fale sobre o que fez no dia anterior, o que vai fazer no dia que se segue e, se for o caso, o que o impede de prosseguir.

Essas reuniões devem ser rápidas. Por isso, se sugere que sejam feitas com as pessoas em pé e em frente a um quadro de anotações. Além disso, recomenda-se que sejam feitas logo após o almoço, quando os participantes estarão mais imersos nas questões do trabalho (longe dos problemas pessoais), além de ser uma boa maneira de dissipar o cansaço que atinge os desenvolvedores no início da tarde.

Esse formato de reunião em pé, semelhante ao que jogadores de alguns esportes fazem nos intervalos é de onde o método tira seu nome (*scrum*, em inglês).

4.3.5 Funcionamento Geral do *Scrum*

O funcionamento geral do Modelo *Scrum* pode ser entendido a partir do resumo apresentado na Figura 4-7.

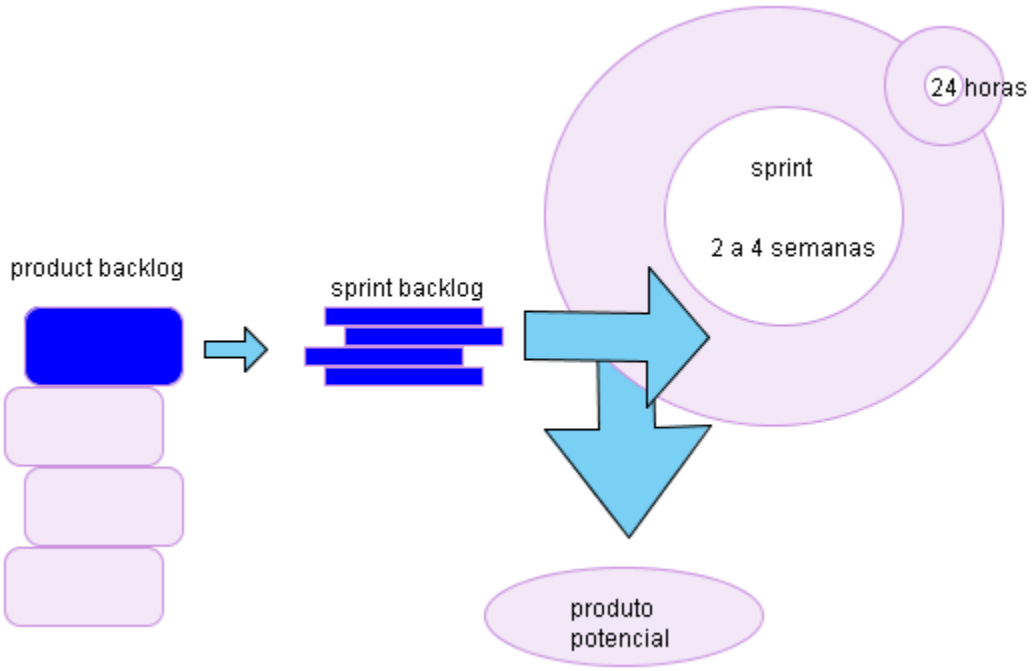


Figura 4-7: Modelo *Scrum*.

Basicamente, o lado esquerdo da figura mostra o *product backlog*, com as histórias de usuário, que devem ser priorizadas e 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 em um dia de trabalho por pessoa. Assim, por exemplo, um *sprint* de 2 semanas (10 dias de trabalho) com 3 pessoas, teria a capacidade de acomodar 30 pontos de história.

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.

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

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

4.4 XP – *eXtreme Programming*

Programação Extrema, ou *XP* (*eXtreme Programming*) é um modelo ágil inicialmente adequado a equipes pequenas e médias que é baseado em uma série de valores, princípios e regras. *XP* surgiu no final da década de 1990, nos Estados Unidos.

Dentre os principais valores de *XP* pode-se citar: simplicidade, respeito, comunicação, *feedback* e coragem:

a) *Simplicidade*. Segundo o *Chaos Report* (Standish Group, 1995)[51] mais da metade das funcionalidades introduzidas em sistemas *nunca* são usadas. XP sugere como valor a simplicidade, ou seja, a equipe deve se concentrar nas funcionalidades efetivamente necessárias e não naquelas que *poderiam talvez* ser necessárias, mas das quais ainda não se tem evidência de que são.
b) *Respeito*. Respeito entre os membros da equipe e 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 é falha, e o projeto afunda.
c) *Comunicação*. Em desenvolvimento de software a comunicação é essencial para que o cliente consiga dizer o que realmente precisa. XP preconiza comunicação de boa qualidade, preferindo encontros presenciais ao invés de videoconferências, videoconferências, ao invés de telefonemas, telefonemas ao invés de emails e assim por diante. Ou seja, quanto mais pessoal e expressiva a forma de comunicação, melhor.
d) *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.
e) *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 a coragem de abraçar as inevitáveis modificações ao invés de simplesmente ignorá-las, por estarem eventualmente fora do contrato formal, ou por serem muito difíceis de acomodar.

A partir destes valores, uma série de princípios básicos são definidos:

a) *Feedback* rápido.
b) Presumir simplicidade.
c) Mudanças incrementais.
d) Abraçar mudanças.
e) Trabalho de alta qualidade.

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

A estes 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.

[51] www.projectsmart.co.uk/docs/chaos-report.pdf

4.4.1 Práticas XP

Para aplicar XP é necessário seguir uma série de práticas que dizem respeito ao relacionamento com o cliente, gerência do projeto, programação e testes:

a) *Jogo de planejamento* (*planning game*). 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 no ciclo semanal que se inicia. Ao final da semana essas funcionalidades são entregues ao cliente. Esse tipo de modelo de relacionamento com o cliente é adaptativo, em oposição aos contratos rígidos usualmente estabelecidos.
b) *Metáfora* (*metaphor*). 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.
c) *Equipe coesa* (*whole team*). O cliente faz parte da equipe de desenvolvimento e a equipe deve ser estruturada de forma que eventuais barreiras de comunicação sejam eliminadas.
d) *Reuniões em pé* (*stand-up meeting*). Como no caso de *Scrum*, reuniões em pé tendem a serem mais objetivas e efetivas.
e) *Design simples* (*simple design*). Isso implica em atender a funcionalidade solicitada pelo cliente sem sofisticar desnecessariamente. Deve-se fazer o 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 implementar. O *design* fácil pode não atender às necessidades, ou pode gerar problemas de arquitetura.
f) *Versões pequenas* (*small releases*). A liberação de versões pequenas do sistema pode ajudar o cliente a testar as funcionalidades de forma contínua. *XP* leva ao extremo este princípio, sugerindo versões ainda menores do que as de outros processos incrementais como UP e *Scrum*.
g) *Ritmo sustentável* (*sustainable pace*). Trabalhar com qualidade um número razoável de horas por dia (não mais do que 8). Horas extras só são recomendadas quando efetivamente trouxerem um aumento de produtividade, mas não podem ser rotina.
h) *Posse coletiva* (*collective ownership*). O código não tem dono e não é necessário pedir permissão a ninguém para modificá-lo.
i) *Programação em pares* (*pair programming*). A programação é sempre feita por duas pessoas em cada computador. Usualmente trata-se de um programador mais experiente e um aprendiz. O aprendiz deve usar a máquina enquanto o mais experiente o ajuda a evoluir em suas capacidades. Com isso, o código gerado terá sempre sido verificado por pelo menos duas pessoas, reduzindo drasticamente a possibilidade de erros. Existem sugestões também de que a programação em pares seja feita por desenvolvedores de mesmo nível de conhecimento, os quais devem se alternar na *pilotagem* do computador (Bravo, 2010).
j) *Padrões de codificação* (*coding standards*). A equipe deve estabelecer e seguir padrões de codificação, de forma que parecerá que o código foi todo desenvolvido pela mesma pessoa, mesmo que tenham sido dezenas.

k) *Testes de aceitação* (*customer tests*). São testes planejados e conduzidos pela equipe em conjunto com o cliente para verificar se os requisitos foram atendidos.
l) *Desenvolvimento orientado a testes* (*test driven development*). Antes de programar uma unidade deve-se definir e implementar os testes pelos quais ela deverá passar.
m) *Refatoração* (*refactoring*). Não se deve fugir da refatoração quando necessária. Ela permite manter a complexidade do código em um nível gerenciável. É um investimento que traz benefícios a médio e longo prazo.
n) *Integração contínua* (*continuous integration*). Nunca esperar até ao final do ciclo para integrar uma nova funcionalidade. Assim que estiver viável ela deve ser integrada ao sistema para evitar surpresas.

As práticas XP, porém não são consenso entre desenvolvedores. Keefer (2003)[52] afirma, entre outras coisas que as práticas não são sempre 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 fétil ou hostil à implantação de métodos ágeis, em especial o XP.

4.4.2 Regras de Planejamento

Wells (2009)[53] vai além das práticas XP, apontando um conjunto sucinto e bastante objetivo de regras para XP. Ele divide as regras em 5 grandes grupos: planejamento, gerência, *design*, codificação e teste. Nesta subseção e nas seguintes, estas regras, que são um detalhamento das práticas XP, são apresentadas.

O *planejamento* é composto pelas atividades que ocorrem antes de iniciar um projeto ou ciclo. 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:

a) *Escrever histórias de usuário*. Servem ao mesmo propósito de casos de uso, mas não são a mesma coisa. 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 usuários como sendo as coisas que eles precisam que o sistema faça para eles. 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 estes 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 detalhe muito alto e precisa ser combinada com outras. Como as histórias são escritas pelo cliente, espera-se que não sejam contaminadas com aspectos técnicos.

[52] www.avocallc.com/*downloads*/ExtremeProgramming.pdf
[53] www.extremeprogramming.org/rules.html

b) *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 os administradores as decisões de negócio. Deve-se estimar o tempo de cada história de usuário em termos de semanas de programação ideais[54] e priorizar as histórias mais importantes do ponto de vista do cliente. Essa priorização pode ser feita com histórias impressas em cartões que são movidos na mesa ou num quadro para indicar prioridades. As histórias são agrupadas em iterações, que só são planejadas pouco antes de iniciarem.

c) *Faça entregas pequenas frequentes*. Algumas equipes entregam software diariamente, o que pode ser um exagero, mas no pior caso, as entregas deveriam acontecer a cada uma ou duas semanas. A decisão de colocar a entrega em operação ou não é do cliente.

d) *O projeto é dividido em iterações*. Prefira iterações de uma a duas semanas. Não planeje as atividades com muita antecedência. Deixe para planejar a iteração pouco antes de ela iniciar. 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, e não em deixar várias coisas inacabadas.

e) *O planejamento da iteração inicia cada iteração*. No planejamento, que inicia cada iteração seleciona-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 levará para ser concluída. Tarefas devem ser estimadas em um, dois ou três dias ideais de programação[55]. Tarefas mais curtas que um dia devem ser combinadas, e tarefas mais longas do que três dias devem ser divididas.

4.4.3 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* mencionadas são:

a) *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

[54] 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.

[55] Um dia ideal de programação é uma jornada de trabalho normal em que uma pessoa dedique-se unicamente ao projeto.

b) *Defina uma jornada sustentável*. Trabalhar além da jornada normal é desgastante e desmoralizante. Se o projeto atrasa 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.

c) *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.

d) *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 esperadas.

e) *Mova as pessoas*. Mobilidade de pessoas em projetos é importante para evitar 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 susceptíveis a perda. Se precisar de conhecimento especializado, contrate um consultor por prazo determinado.

f) *Conserte XP quando for inadequado.* Não hesite em mudar aquilo que não funciona em XP. Isso não significa que se pode fazer qualquer coisa. Segue-se as regras 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.

4.4.4 Regras de *Design*

As regras de *design* são as seguintes:

a) *Simplicidade*. Um *design* simples sempre é executado mais rápido do que um *design* complexo. Porém, simplicidade é subjetiva e difícil de ser medida. Então a equipe é 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, 2011, p. 132), porque ele vai levar a elementos de sistema (classes, módulos, métodos, componentes etc.) mais fáceis de compreender, modificar e estender. Recomenda-se algumas qualidades subjetivas para determinar a simplicidade de um *design*:

- *Testabilidade.* Pode-se escrever testes de unidade para verificar se o código está correto. O sistema deve poder ser quebrado em unidades testáveis, como por exemplo, casos de uso, fluxos, operações de sistema, métodos delegados e métodos básicos.
- *Browseabilidade.* Pode-se encontrar os elementos quando se precisa deles. Bons nomes e uso de boas disciplinas como polimorfismo, herança e delegação ajudam nisso.
- *Compreensibilidade e explicabilidade.* 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 do desenvolvimento.

b) *Escolha uma metáfora de sistema.* Uma boa metáfora de sistema ajuda a explicar seu funcionamento a alguém de 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.

c) *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. A documentação dos processos pode ser feita com diagramas de sequência ou de comunicação da UML.

d) *Crie soluções afiadas* (*spikes*) para *reduzir risco.* Riscos importantes de projeto devem ser explorados de forma definitiva e exclusiva, ou seja, deve ser buscada uma *solução afiada* ou *spike* para o problema identificado. Um *spike* é então um desenvolvimento ou teste projetado especificamente para analisar e possivelmente resolver um risco. Caso o risco se mantenha, deve-se colocar um par de programadores durante uma ou duas semanas exclusivamente trabalhando para examinar e mitigar este risco. A maioria dos *spikes* não serão aproveitados no projeto, podendo ser classificados como uma das formas de prototipação *throw-away*.

e) *Nenhuma funcionalidade é adicionada antes da hora.* Deve-se evitar a tentação de adicionar funcionalidade desnecessária só porque seria fácil fazer isso agora e deixaria o sistema melhor. Apenas o *necessário* é 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.

f) *Use refatoração sempre e onde for possível.* Refatore sem pena. XP não recomenda que se continue usando *design* antigo e ruim só porque ele funciona. Deve-se remover redundâncias, eliminar funcionalidades desnecessárias e rejuvenescer *design*s antiquados.

4.4.5 Regras de Codificação

As regras relacionadas à *codificação* de programas são as seguintes:

a) *O cliente está sempre disponível*. XP necessita que o cliente esteja disponível preferencialmente pessoalmente ao longo de todo o processo de desenvolvimento. Entretanto, devido ao longo tempo de um projeto, a empresa cliente pode ser tentada a associar um funcionário pouco experiente ou um estagiário ao projeto. Ele não serve. Deve ser um especialista. Ele deverá escrever as histórias de usuário, bem como

priorizá-las e negociar sua inclusão em iterações. Pode parecer um investimento alto em tempo de funcionários, mas é compensado por ser desnecessário um levantamento de requisitos detalhado ao início, bem como pela não entrega de um sistema inadequado.

b) *O código deve ser escrito de acordo com padrões aceitos*. Padrões de codificação mantêm o código compreensível e passível de refatoração por toda a equipe. Além disso, código padronizado e familiar encoraja a posse coletiva do mesmo.

c) *Escreva o teste de unidade primeiro*. Usualmente, escrever o teste antes do código ajuda a escrever o código melhor. O tempo para escrever o teste e código passa a ser o mesmo que se gastaria para escrever apenas o código, mas se obtem um código de melhor qualidade e uma forma de garantir que continue correto mesmo após mudanças posteriores.

d) *Todo o código é produzido por pares*. Embora pareça contra-sensual, duas pessoas trabalhando em um computador podem produzir tanto código quanto duas pessoas trabalhando separadamente, mas com mais qualidade. Embora seja recomendado que haja um programador mais experiente, a relação não deve ser de professor/aluno, mas de iguais.

e) *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 a sua vez. Deve-se estabelecer turnos de integração que sejam obedecidos.

f) *Integração deve ser frequente*. Desenvolvedores devem estar integrando código ao repositório a cada poucas horas. Postergar a integração pode agravar o problema de todos estarem trabalhando em versões desatualizadas do sistema.

g) *Defina um computador exclusivo para integração*. Este 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 funcionalidade e qual a funcionalidade. O resultado da integração deve passar nos testes de unidade, de forma que se obtém estabilidade em cada versão e localidade nas mudanças e possíveis erros. Se os testes de unidade falharem, a unidade deve ser depurada. Se a atividade de integração levar mais do que cerca de dez minutos, é porque a unidade ainda precisa de alguma depuração adicional antes de ser integrada. Neste caso, a integração deve ser abortada, retornando o sistema a ultima versão estável, e a depuração da unidade deve seguir sendo feita pelo par.

h) *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 juntamente com o código, seja novo ou modificado. Desta forma existe sempre uma garantia de que o software satisfaz as condições de funcionamento. Não ter um dono de partes do sistema também diminui o impacto da perda de membros da equipe.

4.4.6 Regras de Teste

Finalmente as regras referentes ao *teste* do software são as seguintes:

a) *Todo o código deve ter testes de unidade* (Seção 13.2.1). Esta é uma das pedras fundamentais do XP. Inicialmente o desenvolvedor *XP* deve criar ou obter um *framework de teste de unidade*[56]. Depois ele deve testar todas as classes do sistema, ignorando usualmente os métodos básicos triviais como *getters* e *setters*, especialmente se estes forem gerados automaticamente. Testes de unidade favorecem a posse coletiva do código porque protegem o código de ser acidentalmente danificado.

b) *Todo código deve passar pelos testes de unidade antes de ser entregue*. Exigir que o código passe por todos os testes de unidade antes de ser integrado ajuda a garantir que sua funcionalidade está corretamente implementada. Os testes de unidade também favorecem a refatoração, porque protegem o código de mudanças indesejadas de funcionalidade. A introdução de nova funcionalidade deverá provocar a adição de mais testes no teste de unidade específico.

c) *Quando um erro de funcionalidade é encontrado, testes são criados*. Um erro de funcionalidade identificado exige que testes de aceitação sejam criados para proteger o sistema. Assim, os clientes explicam claramente aos desenvolvedores o que eles esperam que seja modificado.

d) *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. Estes 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 Crystal Clear

Crystal Clear é um método ágil criado por Alistair Cockburn em 1997. O método pertence a uma família mais ampla, iniciada em 1992, a família de métodos *Crystal* (Cockburn, 2004). Os outros métodos da família são conhecidos como *Yellow*, *Orange* e *Red*. Sendo *Clear* o primeiro método da série, cada um é indicado para equipes cada vez maiores (até 8, 20, 40 e 100 desenvolvedores, respectivamente), e de maior risco (desconforto, pequenas perdas financeiras, grandes perdas financeiras, morte). À medida que o tamanho de equipe e risco do projeto crescem, os métodos vão ficando cada vez mais formais. Assim, *Crystal Clear* é o mais ágil de todos.

Crystal Clear é, então, uma abordagem ágil adequada a equipes pequenas (no máximo 8 pessoas) e que trabalham juntas (na mesma sala ou em salas contíguas). A equipe usualmente é composta por um *designer* líder e mais dois a sete programadores. O método propõe, entre outras coisas, o uso de radiadores de informação, como quadros e murais facilmente a 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.

[56] Um bom local para baixar *frameworks* para testes de unidade para várias linguagens é www.xprogramming.com/software. Além disso, em www.junit.org/ pode-se encontrar o *JUnit*, que vem se tornando um padrão para desenvolvimento dirigido por testes em Java.

O ciclo de vida de *Crystal Clear* é organizado em três níveis de ciclo (Figura 4-8):

a) A *iteração*, composta por estimação, desenvolvimento e celebração, que usualmente dura poucas semanas.
b) A *entrega*, formada por várias iterações, que no espaço máximo de dois meses vai entregar funcionalidades úteis ao cliente.
c) O *projeto*, formado pelo conjunto de todas as entregas.

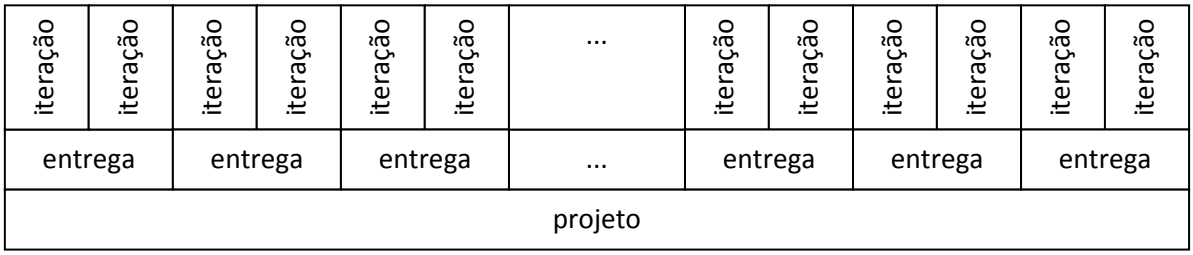


Figura 4-8: Estrutura do ciclo de vida do *Crystal clear*.

Segundo Cockburn[57] a família *Crystal* é centrada em pessoas (*human powered*), ultraleve e na medida (*stretch to fit*), onde:

a) *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.
b) *Ultraleve* significa que não importa o tamanho do projeto, a família *Crystal* fará o possível para reduzir a burocracia, papelada e *overhead*, que existirão na medida suficiente para as necessidades do projeto.
c) *Na medida* significa que o *design* começa com algo menor do que se pensa que seja necessário; depois isso 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 desnecessárias que já foram feitas.

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

a) *Entregas frequentes*. Entregas ao cliente devem acontecer até no máximo a cada dois meses, com versões intermediárias entre elas.
b) *Melhoria reflexiva*. Os membros da equipe discutem frequentemente se o projeto está no rumo certo e comunicam descobertas que possam impactar o projeto.
c) *Comunicação osmótica*. A equipe deve trabalhar em uma única sala para que uns possam ouvir a conversa dos outros e participar delas 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

[57] alistair.cockburn.us/Crystal+methodologies

em pares, pois cada um tem a sua atribuição, mas podem se auxiliar mutuamente com frequência.

d) *Segurança pessoal*. Os desenvolvedores devem ter a certeza de que poderão falar sem medo de repreensões, porque se as pessoas não falam, suas fraquezas viram fraquezas da equipe.
e) *Foco*. Espera-se que os membros da equipe tenham dois ou três tópicos de mais alta prioridade nos quais possam estar trabalhando tranquilamente, sem receber novas atribuições.
f) *Acesso fácil a especialistas*. Especialistas de domínio, usuários e cliente devem estar disponíveis para colaborar com a equipe de desenvolvimento.
g) *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 propriedades acima do que seguir procedimentos.

4.5.1 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 este princípio.

Porém, na maioria das vezes em que se fala em entregas frequentes, pensa-se em 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 este tipo de cliente. Muito software que é desenvolvido hoje em dia é distribuído via Internet e a comunidade de usuários pode não ser totalmente conhecida. Nestes casos, a tática de efetuar entregas (disponibilização de versões) com muita frequência (por exemplo, semanal), pode ser irritante para alguns usuários. Mas por outro lado, diminuir a frequência das entregas pode fazer com que a equipe de desenvolvimento perca um importante *feedback*. A solução neste caso é definir um conjunto de usuários amigáveis, que não se importem em receber versões com frequência maior do que os usuários normais.

Mesmo que usuários amigáveis não sejam encontrados, ainda assim, a sugestão é que o ciclo de desenvolvimento seja finalizado como se a entrega fosse ser feita, ou seja, uma falsa entrega é criada, com toda a formalidade, como se fosse uma entrega verdadeira. Mas, ela não é 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. Mas a iteração pressupõe o fim de um ciclo de atividades previamente definidas e controladas, incluindo várias integrações ao longo do ciclo. Para Crystal é importante que o fim de um ciclo seja marcado com uma celebração para que a equipe ganhe ritmo emocional com a sensação de etapa concluída, pois, afinal, são pessoas, e não máquinas.

Crystal Clear assume que uma iteração possa durar de uma hora a três meses. Mas, o mais comum é que as iterações durem de duas semanas a dois meses, como no Processo Unificado.

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 pode baixar.

Uma estratégia melhor é manter o prazo e deixar a equipe disponibilizar aquilo que for possível naquele intervalo de tempo, e depois replanejar, se necessário, as próximas iterações.

Algumas equipes poderão tentar usar a técnica de requisitos fixos (*requirements locking*), ou seja, é assumido que durante uma iteração os requisitos ou suas prioridades não poderão mudar. Isso dá à equipe a paz de espírito de saber que poderão 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 também que entregas frequentes têm a ver com entregar software ao cliente, e não simplesmente completar iterações. Algumas equipes poderão fazer com que cada iteração corresponda a uma entrega, outras entregarão software a cada 2, 3 ou 4 iterações, outras ainda definirão no calendário datas específicas para iterações que produzirão entregas. Em todos os casos, não basta que a equipe faça iterações rápidas, é necessário entregar software com frequência. Não adianta fazer 24 iterações ao longo de um ano, mas nenhuma entrega, pois neste meio tempo o cliente não terá dado nenhum *feedback* sobre o que foi desenvolvido.

4.5.2 Melhoria Reflexiva

Uma das coisas que pode fazer com que um projeto que está falhando dê a volta por cima é a prática de *melhoria reflexiva*. Esta 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, mais importante de tudo, colocar em prática estas mudanças.

O melhor é que não é necessário gastar muito tempo com essa atividade. Uma ou poucas horas por mês serão normalmente suficientes.

É interessante observar que muitos projetos enfrentam grandes dificuldades já nas primeiras iterações. O que poderia levar a uma catástrofe logo de início, porém, deve ser considerado como um ponto de partida para a reflexão e aprimoramento das praticas (refletir e melhorar). Por outro lado, caso estes problemas não sejam seriamente abordados logo no início, poderão minar o projeto rapidamente e desmoralizar tanto a equipe que se torne 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 outras vezes ainda as práticas de projeto da equipe.

A sugestão de *Crystal clear* é que a cada poucas semanas, ou mensalmente, ou ainda 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 e não estão funcionando. Deve ser criada uma lista das coisas que serão mantidas e das coisas que devem mudar, e estas listas devem ser colocadas à vista de todos para que sejam gravadas e efetivamente mudadas nas próximas iterações.

4.5.3 Comunicação Osmótica

Comunicação osmótica significa que a informação deve fluir pelo ambiente, ou seja, as pessoas devem ser capazes de ouvir a conversa dos outros e intervir, se desejarem, ou continuar seu trabalho. Isso usualmente é obtido quando se coloca todos os desenvolvedores em uma mesma sala.

Além disso, é importante que as telas dos computadores sejam acessíveis. Em alguns casos é interessante que um pequeno grupo possa se reunir defronte a um computador para visualizar problemas e dar sugestões. Então devem ser evitados *designs* de sala que impeçam este 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 30 segundos para sua pergunta chegar aos olhos e ouvidos de alguém que possa responder?". Se a resposta for sim a esta pergunta, o projeto pode enfrentar dificuldades.

Comunicação osmótica tem custo baixo, mas é altamente eficiente em termos de *feedback*, de forma que erros são rapidamente 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 nestas situações. Pode-se tentar pelo menos deixar a equipe em salas próximas, mas ainda assim a comunicação osmótica só vai ocorrer entre as pessoas da mesma sala. Outra possibilidade, no caso de equipes grandes ou distribuídas, seria utilizar ferramentas de comunicação como videoconferência e *chat online* de forma que questões sejam colocadas de uma pessoa para outra, mas potencialmente visíveis 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 auto-regulando 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 então ele deve 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.

4.5.4 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 isso envolve,

entre outras coisas, dizer ao gerente que o cronograma não é realístico, que o *design* de um colega precisa melhorar ou mesmo que ele precisa tomar banho com mais frequência.

A segurança pessoal é importante porque com ela a equipe consegue descobrir e reparar suas fraquezas. 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*. Confiança é dar ao outro poder sobre si mesmo. Algumas pessoas confiam nos outros até que uma prova em contrário os faça rever esta confiança, outras evitam confiar até que tenham segurança de que o outro não os irá prejudicar.

Existem várias formas pelas quais uma pessoa pode prejudicar outras no ambiente de trabalho, ou mesmo prejudicar o trabalho. Há pessoas que mentem, pessoas que são incompetentes, há pessoas que sabotam o trabalho dos outros tanto ativamente quanto por não fornecer informações ou orientações importantes quando necessário.

Considerando essas formas de prejuízo, pode ser pedir demais a uma equipe que simplesmente confiem uns nos outros. Assim, é mais fácil iniciar com comunicação franca, onde cada um dirá o que o incomoda e o grupo vai regular ações e comportamentos a partir disso.

Segundo Cockburn, estabelecer confiança envolve expor as pessoas a situações onde outros poderiam prejudicá-lo e mostrar que isso não acontece. Por exemplo, um chefe pode expor um erro no *design* de um desenvolvedor e, ao invés de puni-lo, dar suporte para que ele corrija o erro, mostrando que isso é naturalmente parte do processo de autodesenvolvimento.

É importante mostrar que as pessoas não serão prejudicadas mesmo se demonstrarem ignorância sobre algum assunto pertinente à sua área de conhecimento. 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 contra as próprias convicções. As pessoas devem ouvir umas as outras de 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 os 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.

4.5.5 Foco

Foco implica em primeiro se saber em que se vai trabalhar e depois ter tempo, espaço e paz de espírito para fazer o trabalho. Saber quais são as prioridades é algo que usualmente é

determinado pela gerência superior. Ter tempo e paz de espírito vem de ter um ambiente de trabalho onde as pessoas não sejam arrancadas de suas atividades para realizar outras, muitas vezes não relacionadas.

Apenas definir prioridades para os desenvolvedores não é suficiente. Deve-se permitir a eles efetivamente concentrar-se nessas atividades. Um desenvolvedor que é 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 depois de resolvida a interrupção. Se essas interrupções acontecerem várias vezes ao dia não é incomum que o desenvolvedor passe a ficar ocioso nos intervalos, apenas esperando pela próxima interrupção, pois se sempre que está retomando o ritmo de trabalho for interrompido logo vai perceber a futilidade de tentar se concentrar.

Note-se, porém, que este princípio não contradiz o da comunicação osmótica. No caso da comunicação osmótica cada um resolve se quer parar o que está fazendo para responder alguma pergunta ou auxiliar alguém. Isso normalmente toma poucos segundos ou no máximo minutos. Mas uma tarefa de ultima 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 durante muitos minutos, ou mesmo horas.

De qualquer maneira, podem existir interrupções de alta prioridade, mas são poucas as tarefas que não possam esperar algumas horas ou mesmo dias para serem feitas. Ou seja, o importante é saber qual a verdadeira prioridade 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, ficando a nova tarefa na pilha de prioridades aguardando a finalização da tarefa atual para ser iniciada.

Outra coisa, pessoas que trabalham em vários projetos ao mesmo tempo dificilmente farão algum progresso em qualquer um deles. Segundo Cockburn 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 talvez no máximo dois projetos simultaneamente, mas quando se adiciona um terceiro projeto ela passa a ser não efetiva nos três.

Quando um desenvolvedor está atulhado com vários projetos e várias atividades simultâneas, a solução gerencial é definir a lista de prioridades e definir qual a atividade (ou, no máximo duas 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 funcionário que inicia em um projeto terá um tempo mínimo (por exemplo, dois dias) neste projeto antes de ser realocado para outro projeto. Isso dá ao funcionário a tranquilidade de saber que seu esforço inicial para entrar no ritmo do projeto não será bruscamente interrompido antes que tenha oportunidade de produzir algo de valor para o projeto.

4.5.6 Acesso Fácil a Especialistas

Não há dúvida de que acesso fácil a especialistas facilita muito o desenvolvimento de um projeto de software, uma vez que os desenvolvedores são especialistas em sistemas, mas não *no* sistema específico que estão desenvolvendo. Infelizmente essa característica não é das mais fáceis de obter em um projeto.

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

Cockburn afirma que no mínimo uma hora por semana seria essencial 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 do que isso poderá 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 poderá acontecer que os desenvolvedores incorporem no código sua melhor estimativa e depois esqueçam 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 para serem respondidas é importante que, mesmo o especialista não estando fisicamente presente no ambiente de desenvolvimento todas as horas da semana, exista uma forma de comunicação imediata com ele, como por exemplo, telefone.

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

a) *Reuniões uma ou duas vezes por semana com usuário e telefonemas adicionais*. O usuário dará muita informação importante à equipe nas primeiras semanas. Posteriormente, a necessidade que a equipe terá dele vai diminuindo gradativamente. Um ritmo natural vai se formando com o usuário informando requisitos e avaliando software desenvolvido a cada ciclo iterativo. Telefonemas adicionais durante a semana (poucos) ajudam a equipe a não investir tempo na direção errada,
b) *Um ou mais usuários especialistas permanentemente na equipe de desenvolvimento*. Esta é uma hipótese mais difícil de ser conseguida. Mas opções são colocar a equipe para trabalhar dentro da empresa cliente ou co-locada com algum usuário.
c) *Enviar desenvolvedores para trabalharem como trainees junto ao cliente por um tempo*. Por mais estranha que esta 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 eles vão desenvolver poderá ajudar a melhorar o modo de trabalho do cliente.

Uma coisa importante neste 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 nível, 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 suas necessidades.

4.5.7 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 realmente produtiva quando conseguir fazer integrações frequentes de versões automaticamente controladas e testadas. Desta forma, o processo de geração de novas versões de um sistema poderá levar poucos minutos e a confiabilidade nesta integração será bastante alta.

4.5.7.1 Teste Automatizado

O teste automatizado não pode ser considerado uma propriedade essencial de um processo de desenvolvimento porque 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.

Teste automatizado implica que a pessoa responsável pelo teste poderá, a qualquer momento, iniciar o teste e sair para fazer outra coisa enquanto o teste é realizado. Não deve haver necessidade de intervenção humana nos testes. Os resultados poderão 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 disto.

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, como forma de garantir que novos defeitos não foram introduzidos inadvertidamente.

Porém, é conveniente lembrar que este tipo de teste é unicamente para detectar defeitos no *design*. Para validar o sistema frente aos requisitos de usuário é necessário que uma avaliação seja feita por um ser humano. O mesmo também é necessário para avaliar a usabilidade de uma interface.

A automatização de testes de sistema a partir do uso de uma interface gráfica é usualmente uma atividade cara e consumidora de tempo. Alem disso, qualquer modificação na interface pode fazer com que a suíte de testes tenha que ser refeita. Assim, normalmente, tais testes não estão entre os mais recomendados. Para que testes de sistema sejam automatizados, então é necessário que a equipe invista em uma arquitetura do tipo *MVC* (*Model*/*View*/*Controller*) (Reenskaug, 2003)[58] onde os testes de sistema possam ser feitos pelo acesso sequencial de funções da controladora sem passar pela interface gráfica (*view*).

[58] heim.ifi.uio.no/~trygver/2003/javazone-jaoo/HM1A93.html

4.5.7.2 *Sistema de Gerenciamento de Configuração*

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

4.5.7.3 *Integrações Frequentes*

Quanto mais frequentemente a equipe fizer integração de partes do sistema, mais rapidamente erros serão detectados. Deixar de fazer integrações frequentes faz com que erros se acumulem e, desta forma, começam a se multiplicar, ou seja, um conjunto de erros é muito mais difícil de consertar do que cada um deles individualmente.

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

4.6 ASD - *Adaptive Software Development*

Adaptive Software Development (*ASD*)[59] é um método ágil criado por Jim Highsmith (2000) que aplica ideias oriundas da área de sistemas adaptativos complexos (*teoria do caos*[60]). Ele vê o processo de desenvolvimento de software como um sistema complexo com *agentes* (desenvolvedores, clientes e outros), *ambientes* (organizacional, tecnológico e de processo) e *saídas emergentes* (os produtos sendo desenvolvidos).

O modelo fundamenta-se em desenvolvimento cíclico iterativo baseado em três grandes fases: especular, colaborar e aprender.

4.6.1 Especular

Especular corresponde ao planejamento adaptativo de ciclo. Ao invés de planejar, o modelo assume que o mais provável nos momentos iniciais é que os interessados ainda não saibam exatamente o que vão fazer. Assim, eles especulam.

Nesta fase, porém, objetivos e prazos devem ser estabelecidos. O plano de desenvolvimento será baseado em componentes. A especulação implica na realização das seguintes atividades:

a) Determinar o tempo de duração do projeto, determinar o número de ciclos e sua duração ideal (Seção 6.4).
b) Descrever um grande objetivo para cada ciclo.
c) Definir componentes de software para serem desenvolvidos a cada ciclo.
d) Definir a tecnologia e suporte para os ciclos.
e) Desenvolver a lista de tarefas do projeto.

[59] www.adaptivesd.com/
[60] www.santafe.edu/

4.6.2 Colaborar

Colaborar corresponde à engenharia concorrente de componentes. A equipe deve então tentar equilibrar seus esforços para realizar as atividades que podem ser mais previsíveis e aquelas que são naturalmente mais incertas.

A partir dessa colaboração, vários componentes serão desenvolvidos de forma concorrente.

4.6.3 Aprender

Aprender corresponde à revisão de qualidade. Os ciclos de aprendizagem são baseados em pequenas interações de projeto, codificação e teste, onde os desenvolvedores, ao cometerem pequenos erros baseados em hipóteses incorretas, estão aprimorando seu conhecimento sobre o problema, até dominá-lo.

Esta fase exige repetidas revisões de qualidade e testes de aceitação com a presença do cliente e de especialistas do domínio. Três atividades de aprendizagem são recomendadas:

a) *Grupos de revisão com foco de usuário*. Um *workshop*, onde desenvolvedores apresentam o produto e usuários tentam usá-lo, apresentando suas observações.
b) *Inspeções de software.* Inspeções (Seção 11.4.2) e testes (Capítulo 13) que têm como objetivo detectar defeitos do software.
c) *Postmortems.* Onde a equipe avalia o que fez no ciclo e, se necessário, muda seu modo de trabalhar.

Essas três fases não são necessariamente sequenciais, mas podem ocorrer de forma simultânea e não linear.

4.6.4 Inicialização e Encerramento

Além dessas três fases iterativas, o modelo prevê uma fase inicial e uma fase final que ocorrem uma única vez cada:

a) *Inicialização de projeto*. A preparação para iniciar os ciclos iterativos. A inicialização deve ocorrer com um *workshop* de uns poucos dias, dependendo do tamanho do projeto, onde a equipe vai estabelecer as missões, ou objetivos, do projeto. Nesta fase também serão levantados requisitos iniciais, restrições e riscos, bem como feitas as estimativas iniciais de esforço.
b) *Garantia final de qualidade e disponibilização*. Que inclui os testes finais e a implantação do produto.

4.6.5 Juntando Tudo

O modelo ASD pode ser resumido pelo diagrama da Figura 4-9.

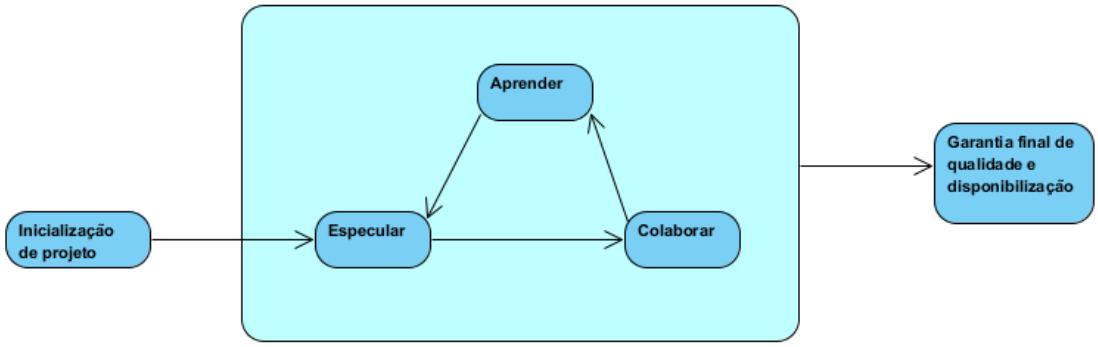


Figura 4-9: Modelo ASD.

O modelo ASD é semelhante aos outros modelos ágeis já vistos nos seguintes aspectos:

a) É baseado em ciclos iterativos de 4 a 8 semanas.
b) Prazos são pré-fixados (*timeboxing*).
c) É tolerante à mudança e adaptação.
d) É orientado a desenvolver primeiramente os elementos de maior risco.

O modelo ASD talvez não seja tão popular nem tão detalhado quanto XP e *Scrum*, mas ele tem o mérito de enfatizar a necessidade do desenvolvimento adaptativo. Nos modelos prescritivos, normalmente a adaptação é entendida como um desvio que é causado por um erro. Mas em ASD a adaptação é o caminho que leva os desenvolvedores na direção correta, a qual eles seriam incapazes de conhecer de antemão.