



# MODELOS ÁGEIS

INE 5419 – Engenharia de Software II

Prof. Raul Sidnei Wazlawick

UFSC-CTC-INE

2012.1

## CONTEÚDO

- FDD – Feature Driven Development
- DSDM – Dynamic Systems Development Method
- Scrum
- XP – eXtreme Programming
- Crystal Clear
- ASD – Adaptive Software Development



## MANIFESTO ÁGIL

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



## PRINCÍPIOS ÁGEIS (1/2)

- Nossa maior prioridade é **satisfazer o cliente** através da entrega rápida e contínua de software com valor.
- **Mudanças nos requisitos são bem vindas**, mesmo nas etapas finais do projeto. Processos ágeis usam a mudança como um diferencial competitivo para o cliente.
- **Entregar software frequentemente**, com intervalos que variam de duas semanas a dois meses, preferindo o intervalo mais curto.
- Administradores (*business people*) e desenvolvedores devem **trabalhar juntos diariamente** durante o desenvolvimento do projeto.
- Construa projetos em torno de **indivíduos motivados**. Dê a eles o ambiente e o suporte e confie que eles farão o trabalho.



## PRINCÍPIOS ÁGEIS (2/2)

- O meio mais eficiente e efetivo de tratar a comunicação entre e para a equipe de desenvolvimento é a **conversa face a face**.
- **Software funcionando** é a medida primordial de progresso.
- Processos ágeis promovem **desenvolvimento sustentado**. Os financiadores, usuários e desenvolvedores devem ser capazes de manter o ritmo indefinidamente.
- Atenção contínua à **excelência técnica** e bom *design* melhora a agilidade.
- **Simplicidade** – a arte de maximizar a quantidade de trabalho não feito – é essencial.
- As melhores arquiteturas, requisitos e projetos emergem de **equipes auto-organizadas**.
- Em intervalos regulares a equipe **reflete** sobre como se tornar mais efetiva e então ajusta seu comportamento de acordo.



## FDD – FEATURE DRIVEN DEVELOPMENT

- 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.
- [www.featuredrivendevelopment.com/](http://www.featuredrivendevelopment.com/)



## FDD POSSUI DUAS GRANDES FASES

- *Concepção e planejamento:*

- que implica em pensar um pouco antes de começar a construir, tipicamente 1 a 2 semanas.

- *Construção:*

- desenvolvimento iterativo do produto em ciclos de 1 a 2 semanas.



## PROCESSOS DE CONCEPÇÃO E PLANEJAMENTO

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





## PROCESSOS DE CONSTRUÇÃO

- *DPF – Detalhar Por Funcionalidade*

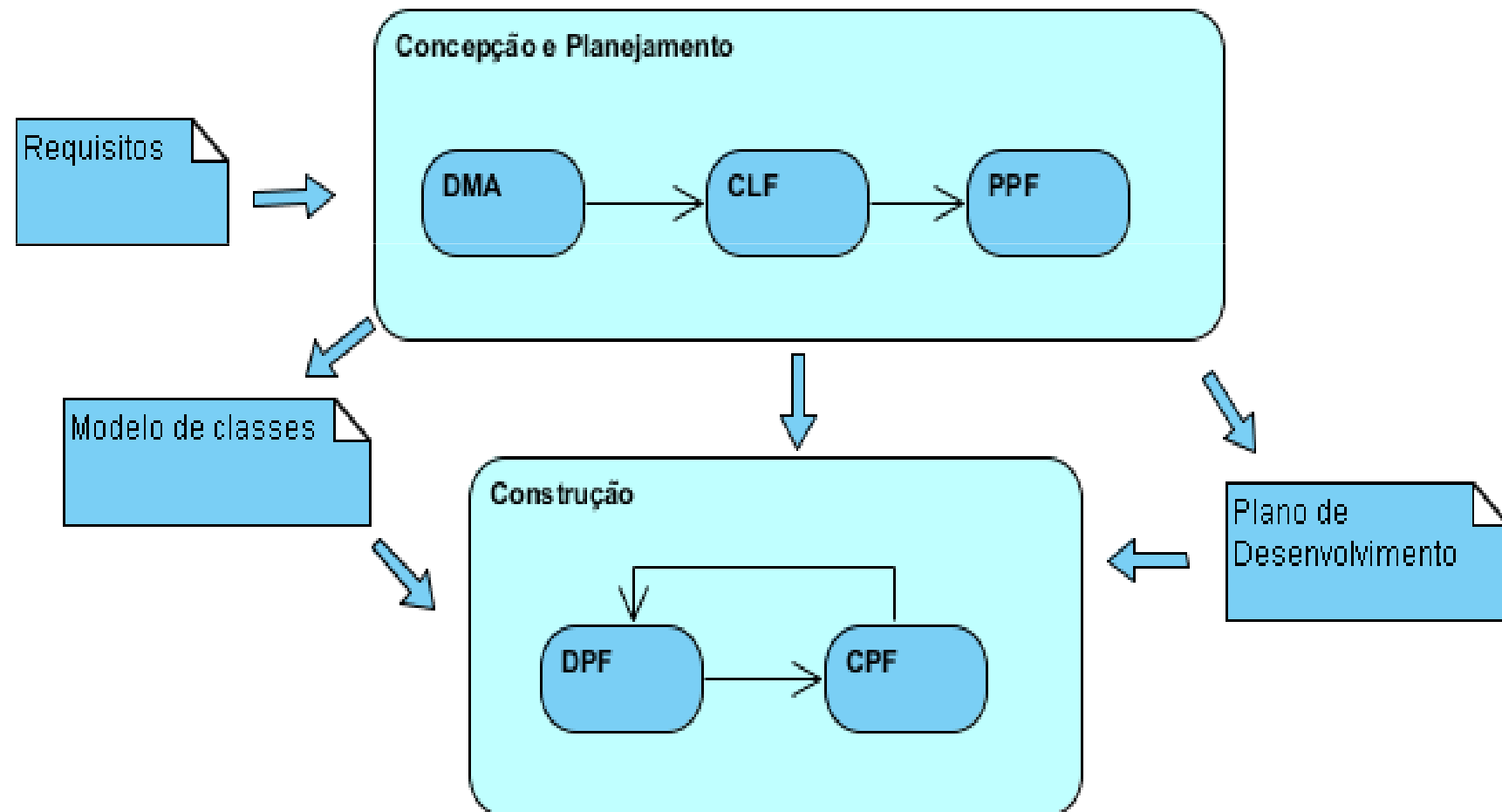
- que corresponde a realizar o *design* orientado a objetos do sistema.

- *CPF – Construir por Funcionalidade*

- que corresponde a construir e testar o software utilizando linguagem e técnica de teste orientadas a objetos.



## ESTRUTURA GERAL DO FDD



# DMA – DESENVOLVER MODELO ABRANGENTE

## (ATIVIDADES 1/2)

- *Formar a equipe de modelagem.*

- É uma atividade obrigatória sob a responsabilidade do gerente de projeto. As equipes devem ser montadas com especialistas de domínio, clientes e desenvolvedores. Deve haver rodízio entre os membros das equipes de forma que todos possam ver o processo de modelagem em ação.

- *Estudo dirigido sobre o domínio.*

- É uma atividade obrigatória sob responsabilidade da equipe de modelagem. Um especialista de domínio deve apresentar sua área de domínio para a equipe, especialmente os aspectos conceituais.

- *Estudar a documentação.*

- É uma atividade opcional sob responsabilidade da equipe de modelagem. A equipe estuda a documentação que eventualmente estiver disponível sobre o domínio do problema, inclusive sistemas legados.



# DMA – DESENVOLVER MODELO ABRANGENTE

## (ATIVIDADES 2/2)

### ○ *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.

### ○ *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.



## DMA (SAÍDAS)

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

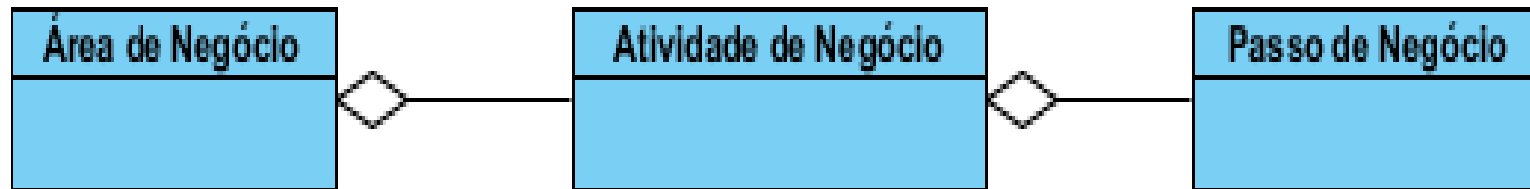


## CLF – CONSTRUIR LISTA DE FUNCIONALIDADES (ATIVIDADES)

- 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).



## ESTRUTURA DA LISTA DE FUNCIONALIDADES



Oriundas das atividades de DMA.  
Por exemplo, “vendas”.

São a decomposição funcional das áreas de negócio.  
Por exemplo, registrar pedido, faturar pedido, registrar pagamento de venda, etc.

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.

## CLF (SAÍDAS)

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






# PPF – PLANEJAR POR FUNCIONALIDADE (ATIVIDADES 1/2)

- *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.

- *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. A sequência de desenvolvimento deve ser construída levando em consideração os seguintes fatores:
    - Priorizar as atividades com funcionalidades mais complexas ou de alto risco.
    - Alocar juntas atividades ou funcionalidades dependentes umas das outras se possível.
    - Considerar marcos externos, quando for o caso, para criação de *releases*.
    - Considerar a distribuição de trabalho entre os proprietários das classes.
- 

# PPF – PLANEJAR POR FUNCIONALIDADE (ATIVIDADES 2/2)

- *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:
    - Dependência entre as funcionalidades e as classes das quais os programadores líder já são proprietários.
    - A sequência de desenvolvimento.
    - A complexidade das funcionalidades a serem implementadas em função da carga de trabalho alocada aos programadores líder.
- *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:
    - Distribuição de carga de trabalho entre os desenvolvedores.
    - Complexidade das classes (priorizar as mais complexas ou de maior risco).
    - Intensidade de uso das classes (priorizar as classes altamente usadas).
    - Sequência de desenvolvimento.



## PPF (SAÍDAS)

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



# DPF – DETALHAR POR FUNCIONALIDADE (ATIVIDADES 1/2)

- *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.
- *Estudo dirigido de domínio.*
  - É uma atividade opcional de responsabilidade do especialista de domínio. Se a funcionalidade for muito complexa, o especialista de domínio deve apresentar um estudo dirigido sobre a funcionalidade no domínio em que ela se encaixa. Por exemplo, uma funcionalidade como “calcular impostos” pode ser bastante complicada e merecerá uma apresentação detalhada por um especialista de domínio antes que os desenvolvedores comecem a projetá-la.
- *Estudar a documentação de referência.*
  - É uma atividade opcional de responsabilidade da equipe de funcionalidades. 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.



# DPF – DETALHAR POR FUNCIONALIDADE (ATIVIDADES 2/2)

- *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.).
- *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.
- *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.



## DPF (SAÍDAS)

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



## CPF – CONSTRUIR POR FUNCIONALIDADE (ATIVIDADES)

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



## CPF (SAÍDAS)

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





## FERRAMENTAS PARA FDD

### ○ *CaseSpec.*

- Uma ferramenta proprietária, mas com *free trial*, para gerenciamento de requisitos ao longo de todo o ciclo de vida. [www.casespec.net/](http://www.casespec.net/)

### ○ *TexExcel DevSuite.*

- Um conjunto de ferramentas específicas para aplicar FDD. [www.techexcel.com/solutions/alm/fdd.html](http://www.techexcel.com/solutions/alm/fdd.html)

### ○ *FDD Tools Project.*

- Um projeto que visa criar ferramentas gratuitas de código aberto para FDD. [fddtools.sourceforge.net/](http://fddtools.sourceforge.net/)



## DSDM – DYNAMIC SYSTEMS DEVELOPMENT METHOD

- 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*. fundamenta-se no *Princípio de Pareto*, ou Princípio 80/20: em geral, 80% do sistema advém de 20% dos requisitos.
- [www.dsdm.org/](http://www.dsdm.org/)



## FASES DO DSDM

- *Pré-projeto.*

- Nesta fase o projeto é identificado e negociado, seu orçamento é definido e um contrato assinado.

- *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.

- *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.



## ESTÁGIOS DA FASE DE CICLO DE VIDA

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



## SAÍDAS DA ANÁLISE DE VIABILIDADE

- Relatório de viabilidade.
- Protótipo de viabilidade.
- Plano de desenvolvimento.
- Controle de risco.



## ANÁLISE DE NEGÓCIO

- São estudadas as características do negócio e as possíveis tecnologias a serem utilizadas.
- Produz priorização dos requisitos e arquitetura inicial.
- Técnica Timeboxing.
- Técnica Moscow:
  - *Must:*
    - requisitos que devem estar necessariamente de acordo com as regras de negócio.
  - *Should:*
    - requisitos que devem ser considerados tanto quanto possível, mas que não tem um impacto decisivo no sucesso do projeto.
  - *Could:*
    - requisitos que podem ser incluídos desde que não afetem negativamente os objetivos de tempo e orçamento do projeto.
  - *Would:*
    - requisitos que podem ser incluídos se sobrar tempo.



## ITERAÇÃO DO MODELO FUNCIONAL (ATIVIDADES)

- *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.
- *Agenda*, que determina quando e como cada uma das funcionalidades será implementada.
- *Criação do protótipo funcional*, que consiste na implementação preliminar das funcionalidades definidas de acordo com a agenda.
- *Revisão do protótipo funcional*, que é feita a partir tanto de revisões da documentação quanto por avaliação do usuário final. Esta atividade deve produzir o *documento de revisão do protótipo funcional*.



## ITERAÇÃO DE DESIGN E CONSTRUÇÃO (ATIVIDADES)

- *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*.
- *Agenda.* A agenda define quando e como serão implementados os requisitos.
- *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.
- *Revisão do protótipo.* O protótipo assim construído deve ser testado e revisado gerando dois artefatos: *documentação para usuário* e *evidências de teste*.





## IMPLANTAÇÃO (ATIVIDADES)

- *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.
- *Treinamento.* Os usuários finais são treinados para o uso do sistema. *Usuários treinados* é considerado o artefato de saída desta atividade.
- *Implantação.* Quando o sistema for implantado e liberado para os usuários finais é obtido o artefato *sistema entregue*.
- *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.



# FILOSOFIA DSDM

- *Envolvimento* do usuário durante todo o tempo do projeto.
- *Autonomia* dos desenvolvedores para tomar decisões sem a necessidade de consultar comitês superiores.
- *Entregas frequentes* de *releases* 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.
- *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.
- *Feedback dos usuários* como forma de realimentar o processo de desenvolvimento iterativo e incremental.
- *Reversibilidade* de todas as ações realizadas durante o processo de desenvolvimento.
- *Previsibilidade* para que o escopo e objetivos das iterações sejam conhecidos antes de iniciar.
- *Ausência de testes no escopo*, já que o método considera a realização de testes como uma atividade fora do ciclo de vida.
- *Comunicação* de boa qualidade entre todos os envolvidos é fundamental para o sucesso de qualquer projeto.



## PAPÉIS NO DSDM (1/2)

- *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.

- *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.

- *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.

- *Anunciante.*

- É qualquer usuário que, por representar um importante ponto de vista, deve trazer informações frequentemente para a equipe, possivelmente diariamente.

- *Gerente de projeto.*

- É responsável por manter as atividades nos prazos, com o orçamento definido e com a qualidade necessária.



## PAPÉIS NO DSDM (2/2)

- *Coordenador técnico.*
  - É responsável pelo projeto da arquitetura do sistema e seus aspectos técnicos.
- *Líder de equipe.*
  - É um desenvolvedor com a função especial de motivar e manter a harmonia de seu grupo.
- *Desenvolvedor.*
  - Trabalha para transformar requisitos e modelos em código executável.
- *Testador.*
  - É o responsável pelos testes do sistema.
- *Escrivão.*
  - É responsável por tomar notas para registro de requisitos identificados, bem como de decisões de *design* tomadas ao longo do processo de desenvolvimento.
- *Facilitador.*
  - Disponibiliza o ambiente de trabalho e avalia o progresso das diversas equipes.



# SCRUM

- 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.



## PAPEIS

- 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 resolvidor de conflitos.
- 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.
- O *Scrum team*, que é a equipe de desenvolvimento. Essa equipe não é necessariamente dividida em papéis como analista, *designer* e programador, mas todos interagem para desenvolver o produto em conjunto. Usualmente são recomendadas equipes de 6 a 10 pessoas.



## PRODUCT BACKLOG

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

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. <i>Design</i> similar para visualizar página de usuário.

## 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.

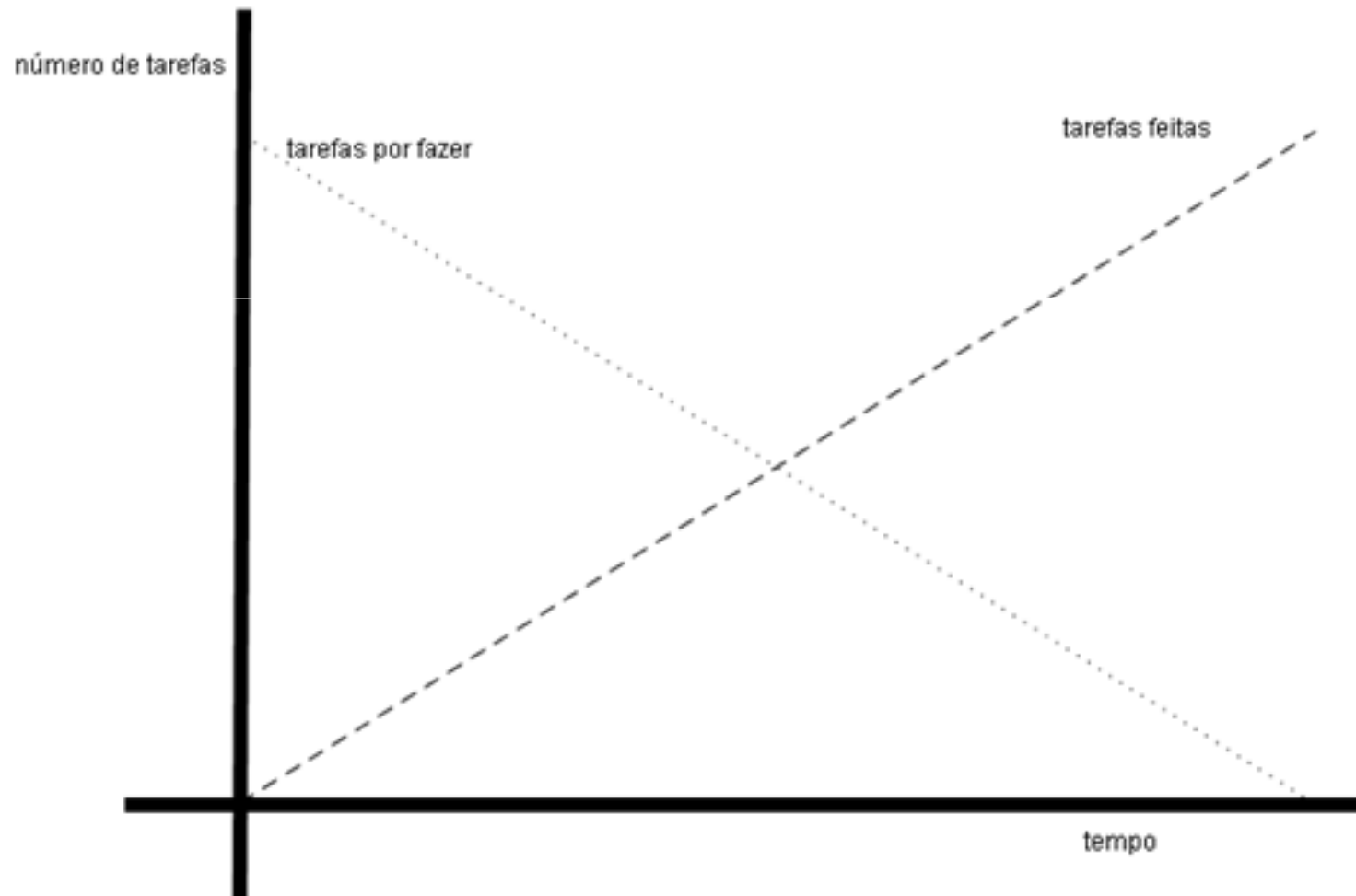




# SPRINT BACKLOG

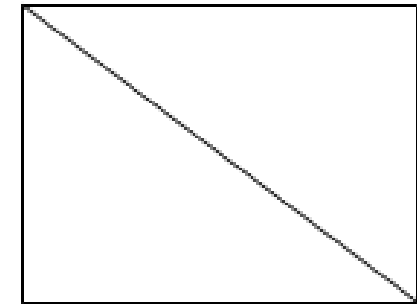
Histórias	Tarefas a fazer	Em andamento	Para verificar	Terminadas
Como usuário, eu ... 8 pontos	<div>Codificar ...</div> <div>Testar...</div> <div>Testar...</div>	<div>Codificar...</div> <div>Testar...</div>	<div>Codificar ...</div>	<div>Codificar ...</div> <div>Codificar ...</div> <div>Codificar ...</div> <div>Testar...</div>
Como usuário, eu... 5 pontos	<div>Codificar ...</div> <div>Codificar ...</div> <div>Testar...</div> <div>Testar...</div>	<div>Testar...</div>	<div>Codificar ...</div> <div>Testar...</div>	<div>Codificar ...</div> <div>Testar...</div> <div>Testar...</div>

# SPRINT BURNDOWN CHART

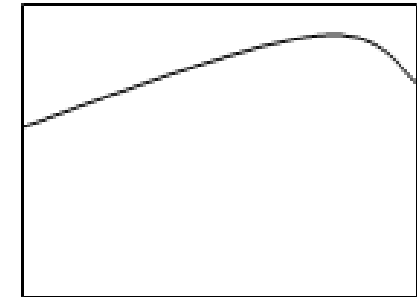


## PERFIS DE EQUIPE (1/3)

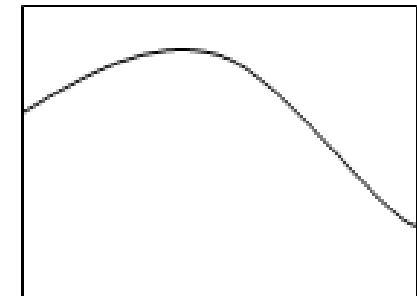
*Fakey-fakey*, caracterizado por uma linha reta e regular, indicando que provavelmente a equipe não está sendo muito honesta porque o mundo real é bem complexo e dificilmente projetos se comportam com tanta regularidade.



*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*.

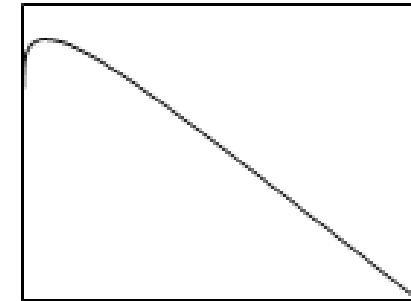


*Middle-learner*, indicando que a equipe pode estar amadurecendo e trazendo para mais cedo as atividades de descoberta e, especialmente, as necessidades de testes.

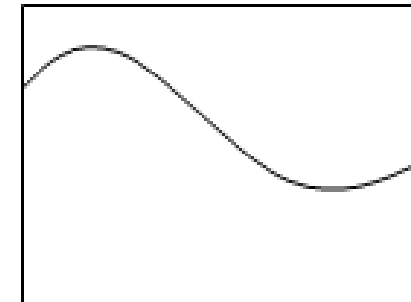


## PERFIS DE EQUIPE (2/3)

*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.

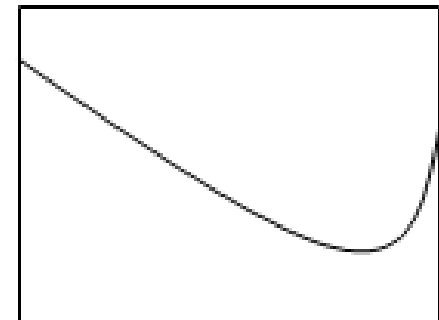


*Plateau*, indicando uma equipe que tenta balancear o aprendizado cedo e tardio, o que acaba levando o ritmo de desenvolvimento a um platô. Inicialmente fazem bom progresso, mas não conseguem manter o ritmo até o final do *sprint*.

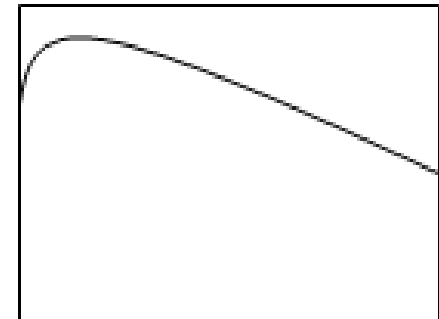


## PERFIS DE EQUIPE (3/3)

*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*.



## FINALIZAÇÃO DE SPRINT

- 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.).

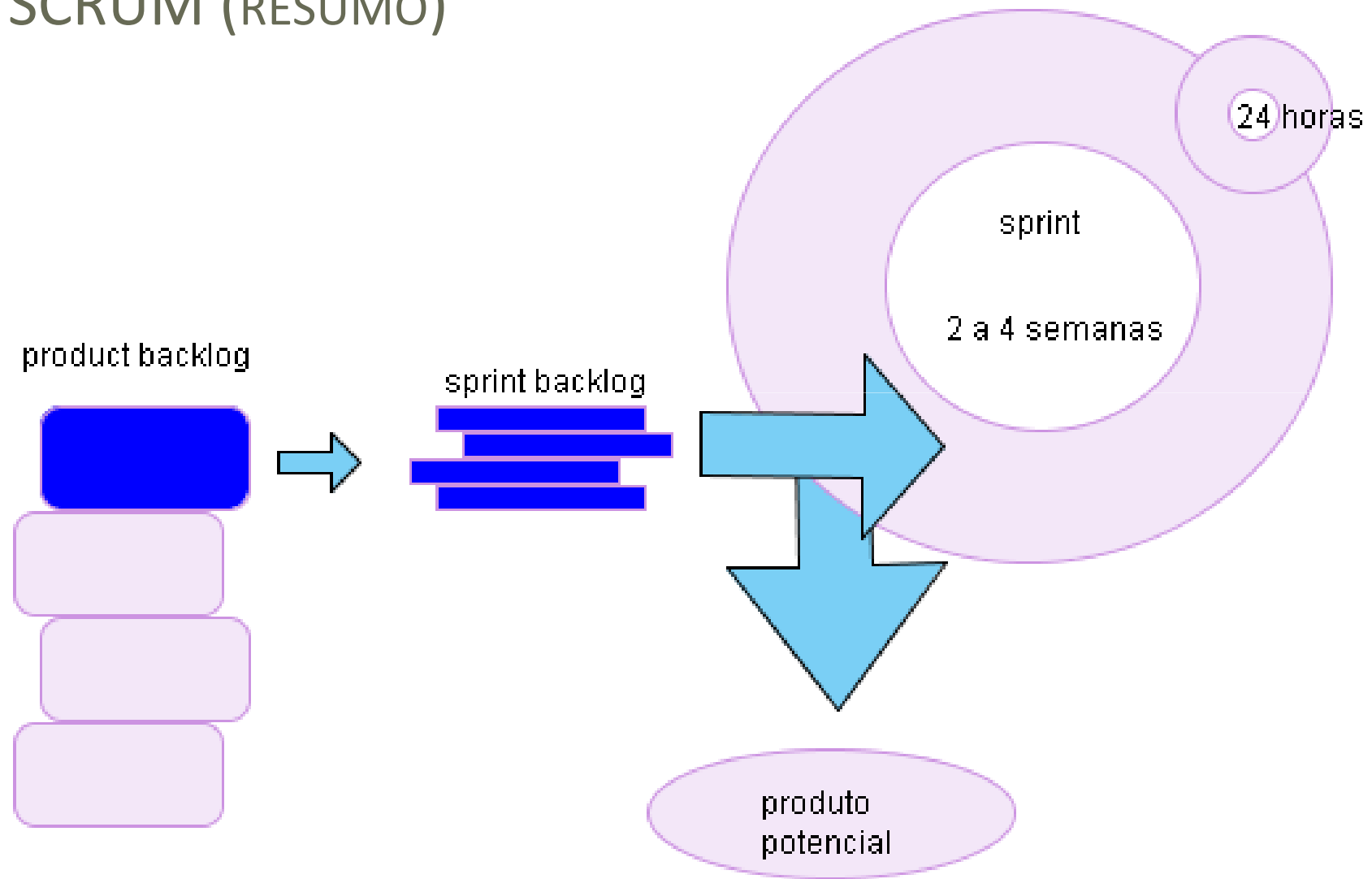


## DAILY SCRUM

- 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.



## SCRUM (RESUMO)





## 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, tendo sido criada por Kent Beck.



## VALORES XP

- Simplicidade.
- Respeito.
- Comunicação.
- Feedback.
- Coragem.



## PRINCÍPIOS XP

- *Feedback* rápido.
- Presumir simplicidade.
- Mudanças incrementais.
- Abraçar mudanças.
- Trabalho de alta qualidade.



# PRÁTICAS XP (1/3)

- *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.
- *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.
- *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.
- *Reuniões em pé (stand-up meeting).*
  - Como no caso de *Scrum*, reuniões em pé tendem a serem mais objetivas e efetivas.
- *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.



## PRÁTICAS XP (2/3)

- *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*.
- *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.
- *Posse coletiva (collective ownership).*
  - O código não tem dono e não é necessário pedir permissão a ninguém para modificá-lo.
- *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).



## PRÁTICAS XP (3/3)

- *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.
- *Testes de aceitação (customer tests).*
  - São testes planejados e conduzidos pela equipe em conjunto com o cliente para verificar se os requisitos foram atendidos.
- *Desenvolvimento orientado a testes (test driven development).*
  - Antes de programar uma unidade deve-se definir e implementar os testes pelos quais ela deverá passar.
- *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.
- *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.



## REGRAS DE PLANEJAMENTO

- *Escrever histórias de usuário.*
- *O planejamento de entregas cria o cronograma de entregas.*
- *Faça entregas pequenas frequentes.*
- *O projeto é dividido em iterações.*
- *O planejamento da iteração inicia cada iteração.*



## REGRAS DE GERENCIAMENTO

- *Dê à equipe um espaço de trabalho aberto e dedicado.*
- *Defina uma jornada sustentável.*
- *Inicie cada dia com uma reunião em pé.*
- *A velocidade do projeto é medida.*
- *Mova as pessoas.*
- *Conserte XP quando for inadequado.*





## REGRAS DE DESIGN

- *Simplicidade.*
- *Escolha uma metáfora de sistema.*
- *Use Cartões CRC durante reuniões de projeto.*
- *Crie soluções afiadas (spikes) para reduzir risco.*
- *Nenhuma funcionalidade é adicionada antes da hora.*
- *Use refatoração sempre e onde for possível.*



## REGRAS DE CODIFICAÇÃO

- *O cliente está sempre disponível.*
- *O código deve ser escrito de acordo com padrões aceitos.*
- *Escreva o teste de unidade primeiro.*
- *Todo o código é produzido por pares.*
- *Só um par faz integração de código de cada vez.*
- *Integração deve ser frequente.*
- *Defina um computador exclusivo para integração.*
- *A posse do código deve ser coletiva.*



## REGRAS DE TESTE

- *Todo o código deve ter testes de unidade.*
- *Todo código deve passar pelos testes de unidade antes de ser entregue.*
- *Quando um erro de funcionalidade é encontrado, testes são criados.*
- *Testes de aceitação são executados com frequência e os resultados são publicados.*



## 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.



## CICLODE VIDA CRYSTAL CLEAR

- *Iteração*, composta por estimação, desenvolvimento e celebração, que usualmente dura poucas semanas.
- *Entrega*, formada por várias iterações, que no espaço máximo de dois meses vai entregar funcionalidades úteis ao cliente.
- *Projeto*, formado pelo conjunto de todas as entregas.

iteração	iteração	iteração	iteração	iteração	iteração	:	iteração	iteração	iteração	iteração	iteração	iteração
entrega		entrega		entrega		..	entrega		entrega		entrega	
projeto												



## SETE PILARES (1/2 FUNDAMENTAIS)

### ○ *Entregas frequentes.*

- Entregas ao cliente devem acontecer até no máximo a cada dois meses, com versões intermediárias entre elas.

### ○ *Melhoria reflexiva.*

- Os membros da equipe discutem frequentemente se o projeto está no rumo certo e comunicam descobertas que possam impactar o projeto.

### ○ *Comunicação osmótica.*

- A equipe deve trabalhar em uma única sala para que uns possam ouvir a conversa dos outros e participar 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 em pares, pois cada um tem a sua atribuição, mas podem se auxiliar mutuamente com frequência.



## SETE PILARES (2/2 RECOMENDADOS)

- *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.

- *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.

- *Acesso fácil a especialistas.*

- Especialistas de domínio, usuários e cliente devem estar disponíveis para colaborar com a equipe de desenvolvimento.

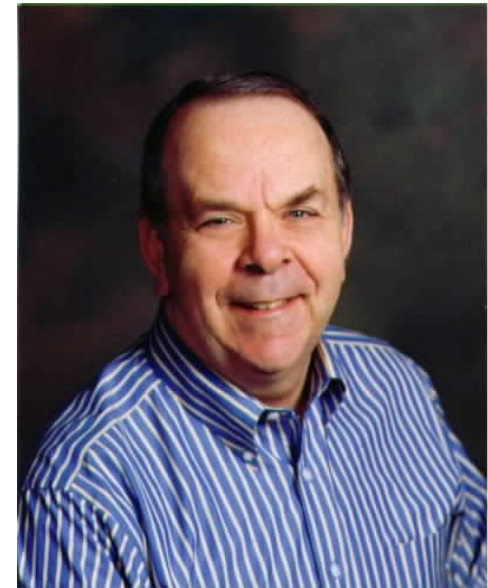
- *Ambiente tecnologicamente rico.*

- O ambiente de desenvolvimento deve permitir testes automáticos, gerenciamento de configuração e integração frequente.



## ASD - ADAPTIVE SOFTWARE DEVELOPMENT

- *Adaptive Software Development (ASD)* é um método ágil criado por Jim Highsmith (2000) que aplica ideias oriundas da área de sistemas adaptativos complexos (*teoria do caos*).
- 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).
- [www.adaptivesd.com/](http://www.adaptivesd.com/)





## CARACTERÍSTICAS

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



## FASES ASD

- Especular
- Colaborar
- Aprender



## 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:
  - Determinar o tempo de duração do projeto, determinar o número de ciclos e sua duração ideal.
  - Descrever um grande objetivo para cada ciclo.
  - Definir componentes de software para serem desenvolvidos a cada ciclo.
  - Definir a tecnologia e suporte para os ciclos.
  - Desenvolver a lista de tarefas do projeto.



## 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.



## 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:
  - *Grupos de revisão com foco de usuário.* Um *workshop*, onde desenvolvedores apresentam o produto e usuários tentam usá-lo, apresentando suas observações.
  - *Inspeções de software.* Inspeções (Seção 11.4.2) e testes (Capítulo 13) que têm como objetivo detectar defeitos do software.
  - *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.



## INICIALIZAÇÃO E ENCERRAMENTO

### ○ *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.

### ○ *Garantia final de qualidade e disponibilização.*

- Que inclui os testes finais e a implantação do produto.



## MODELO ASD (RESUMO)

