Especialização em Gestão Estratégica de Tecnologia da Informação

Métodos Ágeis

Visão Geral

- AVISO
  - Vamos iniciar com o olhar dos “Agilistas”!!!
  - A seguir: precisaremos ser crítios!!!

Novos ventos no mundo do desenvolvimento de software

- Sociedade demanda
  - grande quantidade de sistemas/aplicações
  - software complexo, sistemas distribuídos, heterogêneos
  - requisitos mutantes (todo ano, todo mês, todo dia)
- Mas, infelizmente,
  - não há gente suficiente para desenvolver tanto software com qualidade.

Problemas

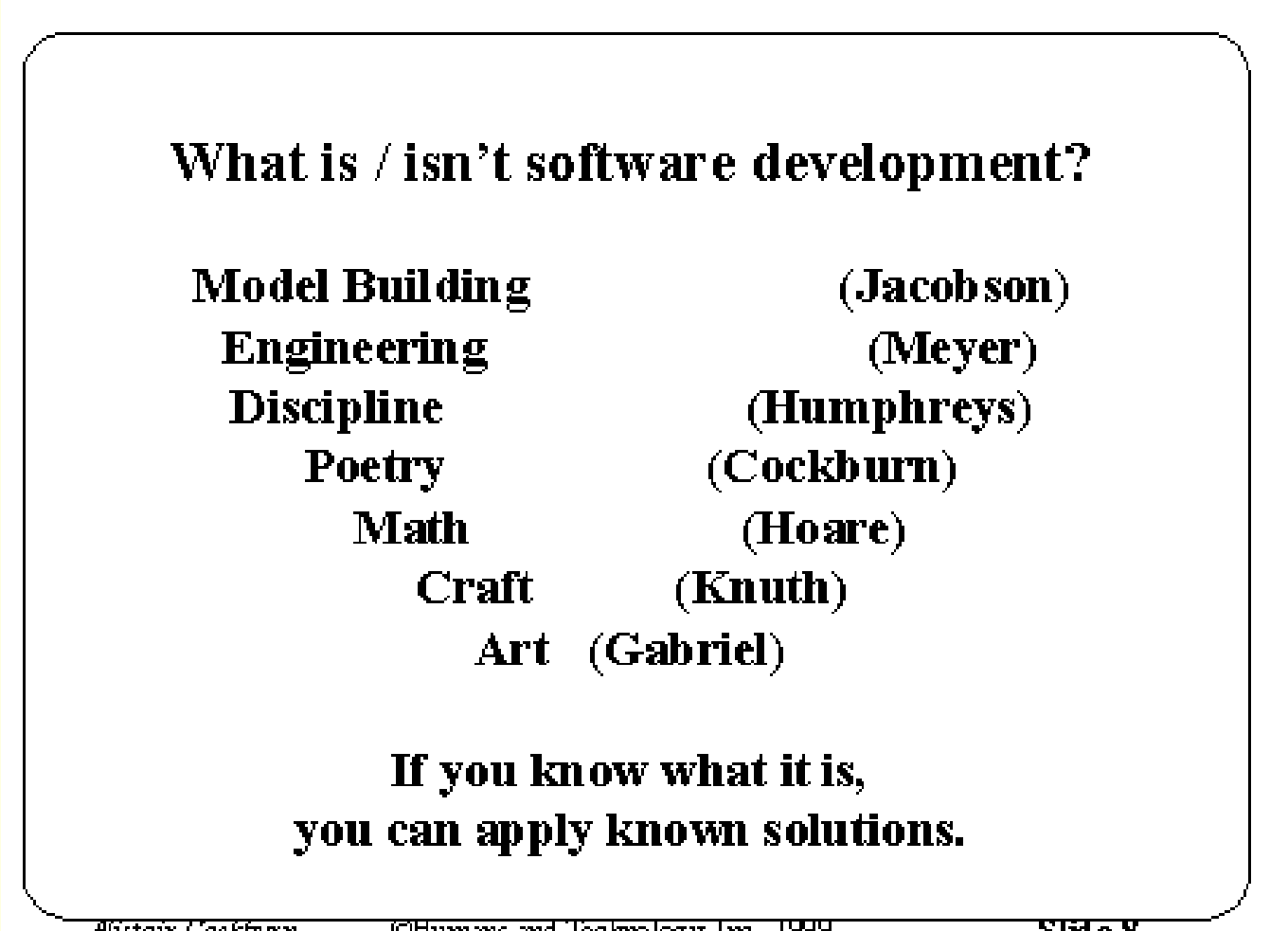
- Com metodologias de desenvolvimento
  - Supõem que é possível prever o futuro
  - Pouca interação com os clientes
  - Ênfase em burocracias (documentos, formulários, processos, controles rígidos, etc.)
  - Avaliação do progresso baseado na evolução da burocracia e não do código
- Com software
  - Grande quantidade de erros
  - Falta de flexibilidade

Como resolver esse impasse?

- Melhores Tecnologias
  - Componentes (melhoram a reutilização)
  - Middleware (aumenta a abstração)

- Melhores Metodologias
  - Métodos Ágeis (foco deste mini-curso)
  - outras... (não abordadas aqui)

O que é desenvolvimento de software (A. Cookburn)?



Processo Reprodutível ???

- Pode existir um processo pré-definido?
  - Alguns pensam que além de possível isto é necessário!
  - Abordagem CMM
  - 5 níveis (inicial, sistematizado, definido, gerencial e otimizado)
    - Os processos-chave são definidos em seguida
- Crítica ao “tradicional” -> leitura correta?

Processo Reprodutível ??? (extraído de artigo da PLoP'98)

- Suposições para "sistematizado/definido"
  - Problema. ∃ fase para definição de requisitos
    - Podem não ser bem definidos, ou mudar
  - Solução. Especificação completa da arquitetura
    - Além do problema acima, envolve processo criativo
  - Desenvolvedores.
    - São humanos (logo, diferentes)
  - Ambiente.
    - Pressão de cronograma, prioridades e clientes mudam

Métodos Ágeis

- Movimento iniciado por programadores experientes e consultores em desenvolvimento de software.
- Questionam e se opõe a uma série de mitos/práticas adotadas em abordagens tradicionais de Engenharia de Software e Gerência de Projetos.
- Manifesto Ágil:
  - Assinado por 17 desenvolvedores em Utah em fevereiro/2001.

O Manifesto do *Desenvolvimento Ágil de Software*

1. ***Indivíduos e interações*** são mais importantes que *processos e ferramentas.*
2. ***Software funcionando*** é mais importante do que *documentação completa e detalhada*.
3. ***Colaboração com o cliente*** é mais importante do que *negociação de contratos*.
4. **Adaptação a mudanças** é mais importante do que *seguir o plano inicial*.

Princípios do Manifesto Ágil

- Maior prioridade: satisfazer o cliente entregando, antes e sempre sistemas com valor.
- Estar preparado para requisitos mutantes. O quê fornece uma vantagem competitiva.
- Entregar versões funcionais em prazos curtos.
- Pessoal de negócios e desenvolvedores juntos.
- Construir projetos com indivíduos motivados, dando um ambiente de trabalho e confiando neles.
- Troca de informações através de conversas diretas.

Princípios do Manifesto Ágil

- Medir progresso através de software funcionando.
- Desenvolvimento sustentável.
- Cuidados com o nível técnico e bom desenho.
- Simplicidade é essencial.
- As melhores arquiteturas, requisitos e desenho aparecem de equipes que se auto organizam.
- De tempos em tempos, o time pensa em como se tornar mais efetivo, e ajusta o seu comportamento de acordo.

Uma Boa Metáfora para Desenvolvimento de Software

- “Um jogo Cooperativo de Invenção e Comunicação”. Alistair Cockburn.

- Como um grupo de pessoas escalando uma montanha em conjunto.
  - Jogo cooperativo, finito, onde se busca alcançar um objetivo.

Dois Objetivos Principais do Desenvolvimento de Software

1. Entregar software que funcione.

2. Preparar o terreno para o próximo projeto.

Epifanias Ágeis

- O gerente de projeto concorda em prosseguir sem que todos os requisitos estejam bem definidos.
- Os desenvolvedores concordam em prosseguir sem ter todos os requisitos documentados.
- O gerente de projeto vê uma novas funcionalidades a cada nova versão, e percebe que o sua participação foi importante com resultados visíveis. Também percebe que o projeto terá sucesso e será útil.

Mais Epifanias Ágeis

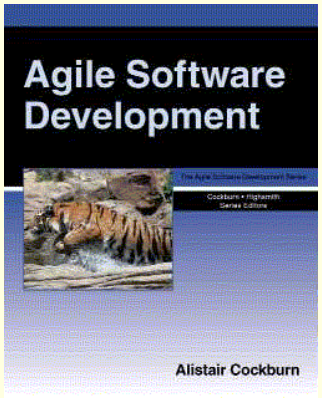
- Os membros da equipe sabem que alguém vai ajudar quando ocorrerem problemas.
- O gerente de TI sente o trabalho de equipe nas áreas de trabalho.
- Os gerentes percebem que não precisam dizer a equipe o que fazer, ou garantir o que vai ser feito.
- A equipe percebe que ninguém vai dizer o quê fazer, isto faz parte do trabalho da equipe.
- Não existem mais a impressão de divisão (testers and programmers), todos são desenvolvedores.

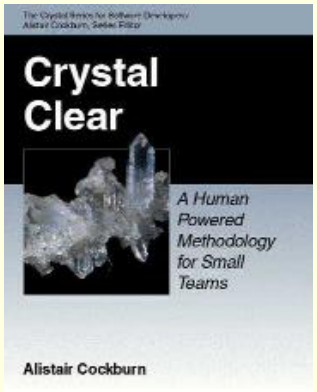
Principais Métodos Ágeis

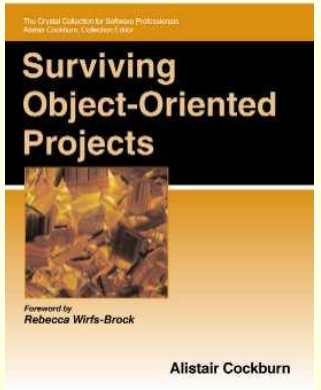
- Daremos uma visão geral
  - Crystal (uma família de métodos)
  - Scrum
- Abordaremos mais detalhadamente
  - Programação eXtrema (XP)
- Não abordaremos
  - *Adaptive Software Development*
  - *Feature Driven Development, etc.*

A família *Crystal* de Métodos

- Criada por Alistair Cockburn
- http://alistair.cockburn.us/crystal
- Editor da série *Agile Software Development* da Addison-Wesley.







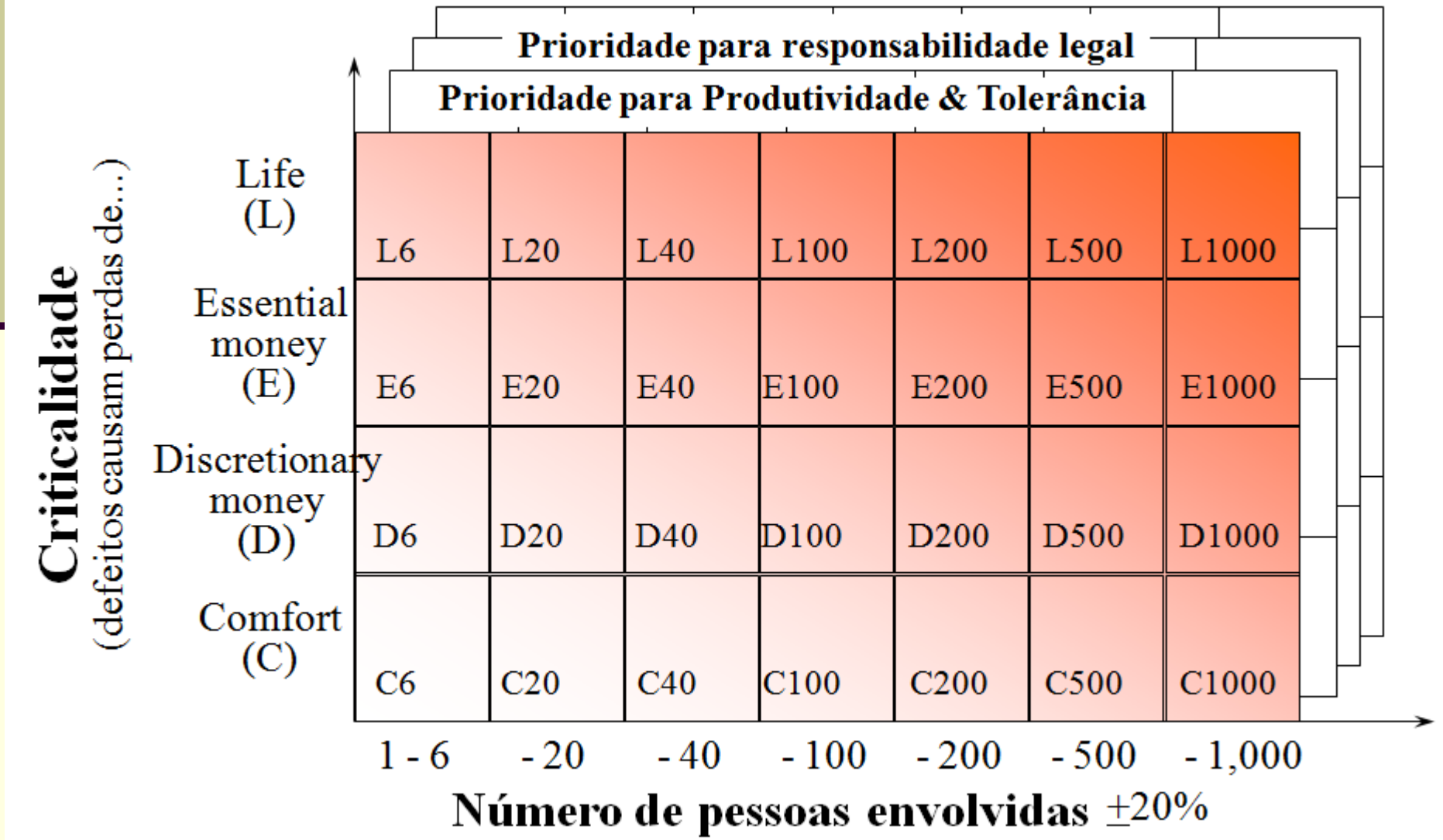
Pequeno histórico

- Foi contratado em 1990 pela IBM
  - documentar a metodologia de desenvolvimento
  - optou por entrevistar as diversas equipes
- Resultado:
  - Pedidos de desculpa por não utilizar ferramentas
    - e entregar software funcionando
  - Explicações sobre como metodologias foram seguidas a risca
    - e o projeto falhou

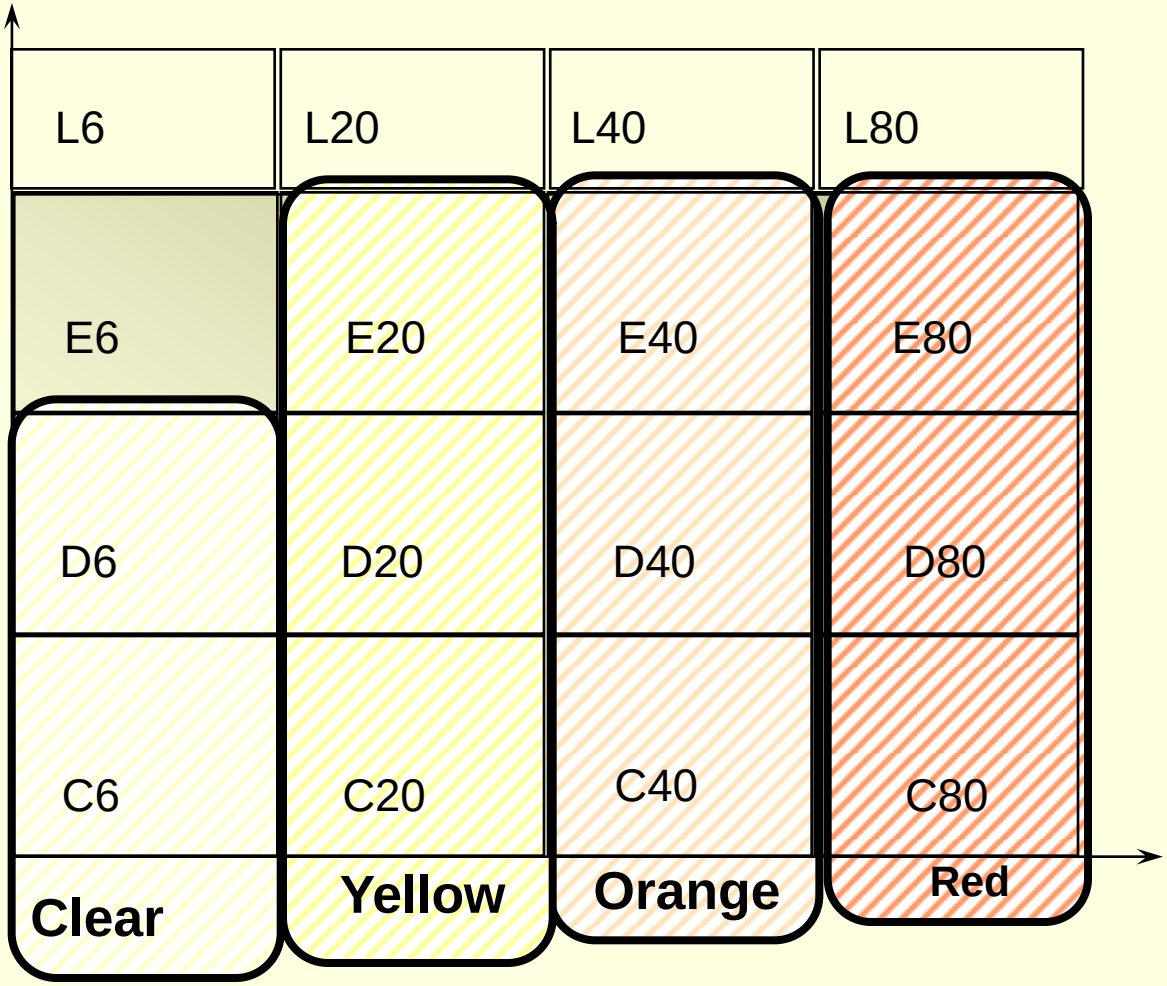
Conclusões (resumo)

- Pode-se trocar documentação escrita por conversas face a face
- Quanto mais se entrega software funcionando e testado, menos depende-se da documentação escrita (promissórias)

Classificação de Projetos de Desenvolvimento de Software



Escopo da Família Crystal



Principais Características da Família Crystal

- Filosofia básica:
  - ênfase em comunicação
  - leve nos produtos gerados (evitar “peso morto”)
- Princípios:
  - Precisamos de menos produtos intermediários se possuímos:
    1. **canais de comunicação informal** ricos e rápidos
    2. **entrega freqüente de código funcionando**
    3. **uso dos pontos fortes das pessoas** (conversas, olhar a sua volta, interagir com outros)
    4. **estar ciente dos pontos fracos das pessoas** (baixa disciplina, falta de cuidado)

Políticas Clear e Orange

- Desenvolvimento incremental
  - com freqüência regular
- Acompanhamento com marcos
  - início, revisão 1, revisão 2, teste, entrega
- Envolvimento direto do usuário
- Testes de regressão de funcionalidades automáticos
- Duas/três visões de usuários por versão
- Workshops para ajustes no produto e na metodologia no início e meio de cada incremento

Adaptação da Metodologia

- Em cada caso, escolha a metodologia mais leve possível que pode fazer o que você precisa.
- Quanto maior o projeto (número de pessoas), maior burocracia será necessária e pior será a produtividade.
- *Reflection Workshops* ao final de cada fase.

Oficinas de Reflexão (*Reflection Workshops*)

- Perguntas:
  - O que aprendemos na última fase (p.ex. mês)?
  - O que podemos fazer de uma forma melhor?
- Resultado:
  - pôster

Manter reuniões com cliente programação pareada	Tentar testes automatizados multas para interrupções escrita pareada de testes
Problemas muitas interrupções erros no código entregue	

Mais perguntas na reflexão

- O quê fizemos de bom e de ruim ?
- Quais são as nossas prioridades
- O quê mantivemos de mais importante
- O quê podemos mudar para a próxima vez ?
- O quê podemos adicionar/tirar ?

- Após 2 ou 3 versões incrementais, a metodologia deve começar a convergir para uma metodologia tolerável para o projeto.

Scrum

Definição informal:
Estratégia em um jogo de rugby onde jogadores colocam uma bola quase perdida novamente em jogo através de trabalho em equipe.

Origens de Scrum

- Jeff Suttherland
  - http://jeffsutherland.com
- Ken Schwaber
  - http://www.controlchaos.com
- Conferências
  - OOPSLA 96, PLoP 98
- Inspiração
  - Desenvolvimento Iterativo e Incremental em empresas (DuPont, Honda, etc) nos anos 80

Fundamentos de Scrum 1/2

- Desenvolvimento de software a partir de padrões de projeto (*design patterns*)
- Mas, o que é isto ???

O quê são padrões ?

- No final dos anos 70, o arquiteto Christopher Alexander escreveu dois livros com a idéia.
- Cada padrão descreve um problema recorrente no nosso ambiente e, em seguida, o princípio de sua solução.
- A solução pode ser aplicada diversas vezes, nunca da mesma maneira.
- Um exemplo: escritório com janela.

Fundamentos de Scrum

- Desenvolvimento de software depende muito de criatividade e de trabalho
- Logo, não é um bom candidato a processos pré-definidos
  - modelo de controle de processo empírico
- O desenvolvimento nem sempre será repetitivo e bem definido
- Mas existem padrões que podem ser usados

Ênfases

- Comunicação
- Trabalho em equipe
- Flexibilidade
- Fornecer Software funcionando
  - incrementalmente

Principais Padrões em Scrum

- *Backlog*
- Equipes
- *Sprints*
- Scrum *Meetings*
- Revisões Scrum/Demos

Backlog

- Lista de todas as funcionalidades desejadas
- É gerada incrementalmente
  - Começa pelo básico, o extra aparece com o tempo
- Pode conter
  - Tarefas diretas, casos de uso e histórias (a la XP)
- A lista é priorizada pelo Dono do projeto
  - Cliente, depto de marketing, ...

O Backlog inicial

- Deve conter características que agreguem algum valor de negócio ao produto.
- Novos requisitos aparecem quando o cliente vê o produto.
- A arquitetura do sistema surge enquanto o projeto surge e é refatorado.

Equipes

- Sem nível hierárquico nem papéis
  - Mas com várias especialidades
- Estão todos no mesmo barco
- Geralmente equipes pequenas (até 10)
  - Existem casos com equipes maiores (800 !)
  - Usa-se também Scrum hierárquico
- Comunicação é essencial
  - Encontro Scrum diário

Sprint

- Unidades básicas de tempo (até 30 dias)
- Começa com um encontro *Sprint*
  - Tarefas do *Backlog* são priorizadas
  - A equipe seleciona tarefas que podem ser completadas durante o próximo *Sprint*
  - As mesmas podem ser quebradas para o *Backlog* do *Sprint*
  - Cada tarefa recebe um responsável na equipe
  - Não há mudança nas tarefas durante o *Sprint*

Scrum Meetings 1/2

- Pequenos encontros diários da equipe
  - geralmente pela manhã
  - galinhas e porcos (só os porcos falam)
  - todos os porcos devem participar
- Questões que aparecem devem ser resolvidas durante o dia e não na reunião
- Os encontros iniciais são geralmente mais longos

Scrum Meetings 2/2

- Questões que devem ser respondida por cada porco:
  - 1) O quê você fez ontem?
  - 2) O quê você vai fazer hoje?
  - 3) Quais os problemas encontrados?
- Ajuda a manter as promessas
- Evita: Como um projeto atrasa um ano?
  - Um dia por vez ...
  - Qualquer deslize pode ser corrigido de imediato

Local do encontro

- Sempre o mesmo local e hora
- Pode ser o local de desenvolvimento
- Pessoas sentadas ao redor de uma mesa
- A sala já deve estar arrumada antes
- Punições (atrasos/faltas)
- Todos devem participar
- Galinhas ficam na periferia
- Pode ser em pé
- Sala bem equipada, quadro branco, etc.

Revisão do Sprint

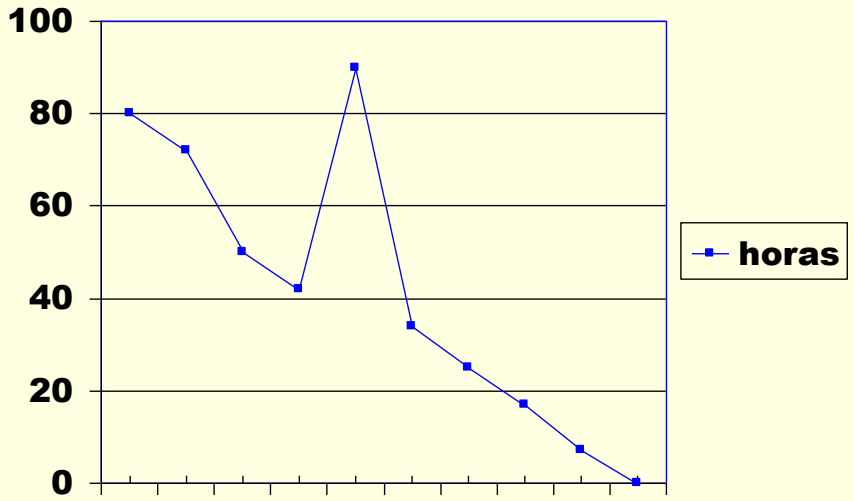
- No final de cada *Sprint* é feita uma reunião com todos os interessados
- Geralmente
  - Na forma de demonstração
  - Informal (preparação rápida, sem projetor,..)
  - Deve ser o resultado natural de um *Sprint*
- O projeto é comparado com os objetivos iniciais do *Sprint*

Scrum Master 1/2

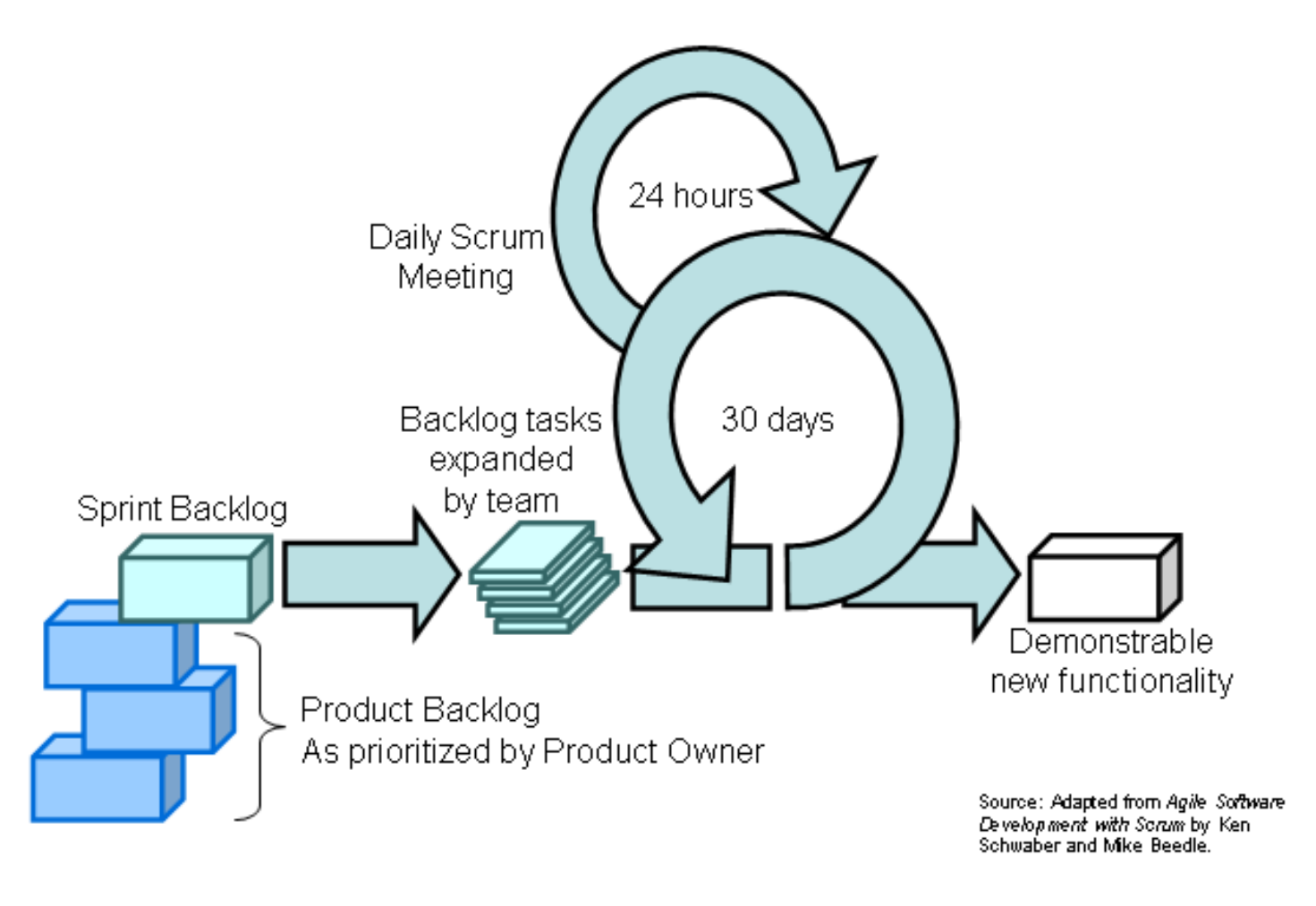
- Faz com que a equipe viva os valores e práticas de Scrum
- Protege a equipe de:
  - Riscos e interferências externos
  - Excesso de otimismo
- Resolve os problemas que aparecerem
  - logísticos
  - de conhecimento/habilidade

Scrum Master 2/2

- Mantém o *Backlog* do *Sprint*
  - Tarefas completadas
  - Identifica eventuais problemas
- Mantém um gráfico de “quanto falta”



Scrum (de forma gráfica)



Scrum final

- Um último *Sprint* para “fechar” o produto
- O objetivo é:
  - Preparar a versão de produção
  - O foco é a eliminação de erros
- Não faz parte do Scrum padrão
  - Mas é bem usado na prática

Programação eXtrema XP

- Metodologia de desenvolvimento de software aperfeiçoada nos últimos 5 anos.
- Ganhou notoriedade a partir da OOPSLA’2000.
- Nome principal: Kent Beck.

Reações a XP

- Alguns odeiam, outros amam.

- Quem gosta de programar ama!
- Deixa o bom programador livre para fazer o que ele faria se não houvesse regras.
- Força o [mau] programador a se comportar de uma forma similar ao bom programador.

Modelo Tradicional de Desenvolvimento de Software

0. Levantamento de Requisitos

1. Análise de Requisitos

2. Desenho da Arquitetura

3. Implementação

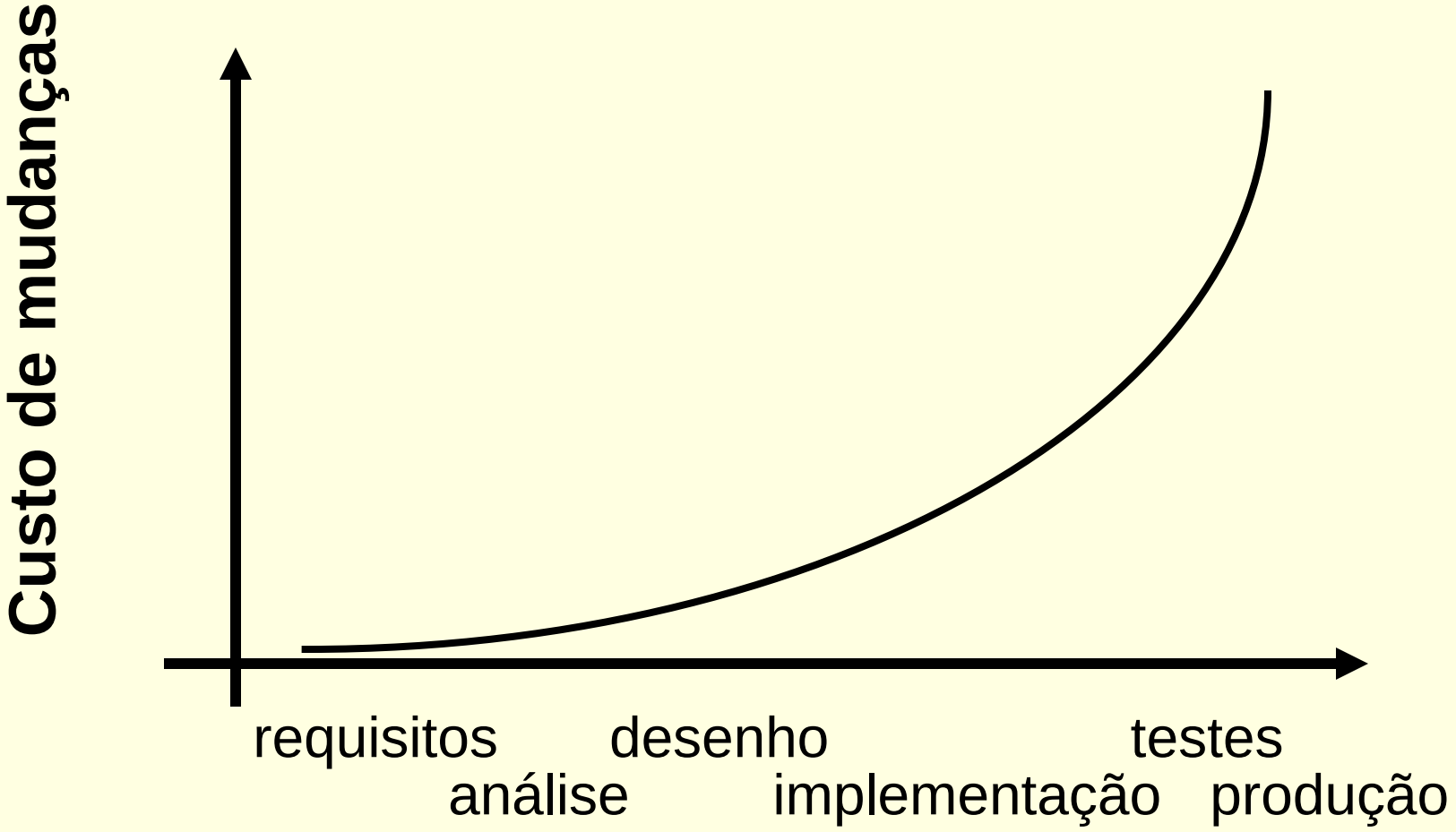
4. Testes

5. Produção / Manutenção

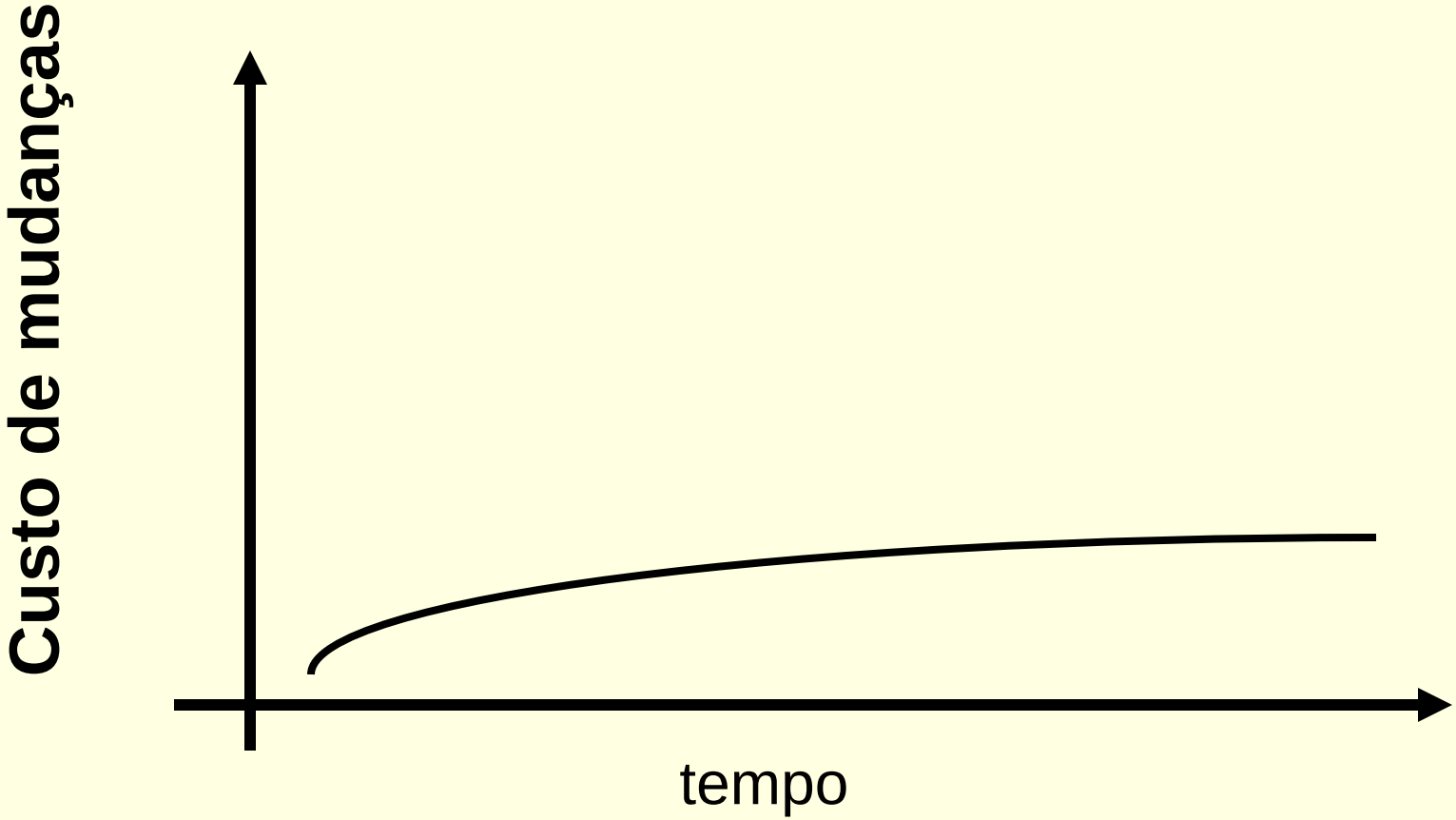
Premissas Básicas do Modelo Tradicional

- É necessário fazer uma análise de requisitos profunda e detalhada antes de projetar a arquitetura do sistema.
- É necessário fazer um estudo minucioso e elaborar uma descrição detalhada da arquitetura antes de começar a implementá-la.
- É necessário testar o sistema completamente antes de mandar a versão final para o cliente.

O que está por trás deste modelo?



E se a realidade hoje em dia fosse outra?



E se essa fosse a realidade?

- A atitude dos desenvolvedores de software seria completamente diferente:
  - Tomaríamos as grandes decisões o mais tarde possível.
  - Implementaríamos agora somente o que precisamos *agora.*
  - Não implementaríamos flexibilidade desnecessária (não anteciparíamos necessidades).

E essa é a nova realidade !
(pelo menos em muitos casos)

- **Orientação a Objetos**: facilita e cria oportunidades para mudanças.
- **Técnicas de Refatoração**.
- **Testes automatizados**: nos dão segurança quando fazemos mudanças.
- **Prática / cultura de mudanças**: aprendemos técnicas e adquirimos experiência em lidar com código mutante.

- Momento ... Really??

A Quem se Destina XP?

- Grupos de 2 a 10 programadores
- Projetos de 1 a 36 meses (calendário)
- De 1000 a 250 000 linhas de código
- Papéis:
  - Programadores (foco central)(sem hierarquia)
  - “Treinador” ou “Técnico” (*coach*)
  - “Acompanhador” (*tracker*)
  - Cliente

E Se Eu Não Me Encaixo Nesses Casos?

- Você ainda pode aprender muito sobre desenvolvimento de software.
- Terá elementos para repensar as suas práticas.
- No início se dizia:
  - “Ou você é 100% eXtremo ou não é eXtremo. Não dá prá ser 80% XP.”
  - *XP is now like teenage sex.*
- Hoje não é mais necessariamente assim.

As 12 Práticas de XP
(versão 2000)

1. Planejamento
2. Fases Pequenas
3. Metáfora
4. Design Simples
5. Testes
6. Refatoramento
7. Programação Pareada
8. Propriedade Coletiva
9. Integração Contínua
10. Semana de 40 horas
11. Cliente junto aos desenvolvedores
12. Padronização do código

Princípios Básicos de XP

- *Feedback* rápido
- Simplicidade é o melhor negócio
- Mudanças incrementais
- Carregue a bandeira das mudanças / não valorize o medo (*Embrace change*)
- Alta qualidade

As 4 Variáveis do Desenvolvimento de Software

- **Tempo**
- **Custo**
- **Qualidade**
- **Escopo (foco principal de XP)**

Um Projeto XP

- Fase de Exploração
  - duração: até 20% do tempo total.
  - termina quando a primeira versão do software é enviada ao cliente.
  - clientes escrevem “historias” (*story cards*).
  - programadores interagem com clientes e discutem tecnologias.
  - Não só discutem, **experimentam** diferentes tecnologias e arquiteturas para o sistema.
  - Planejamento: 1 a 2 dias.

Histórias X casos de uso

- Caso de uso: Autenticação caixa automático
  - Ator: Cliente
  - Descrição: Após inserir o cartão o sistema solicita a senha. O sistema verifica se o cartão é válido e se a senha confere. Se não o usuário tem mais três chances...
- Várias historinhas
  - O sistema mostra telas de boas vindas
  - O sistema verifica se o cartão é válido
  - Implementar sistema de leitura de senha
  - Verificação de senha
  - Releitura de senha se a mesma não confere
  - ...

Um Dia na Vida de um Programador XP

- Escolhe uma história do cliente.
- Procura um par livre.
- Escolhe um computador para programação pareada (*pair programming*).
- Seleciona uma tarefa claramente relacionada a uma característica (*feature*) desejada pelo cliente.

Um Dia na Vida de um Programador XP

- Discute modificações recentes no sistema
- Discute história do cliente
- Testes
- Implementação
- Desenho
- Integração

Um Dia na Vida de um Programador XP

- Atos constantes no desenvolvimento:
  - Executa testes antigos.
  - Busca oportunidades para simplificação.
  - Modifica desenho e implementação incrementalmente baseado na funcionalidade exigida no momento.
  - Escreve novos testes.
  - Enquanto todos os testes não rodam a 100%, o trabalho não está terminado.
  - Integra novo código ao repositório.

Testes

- Fundamento mais importante de XP.
- É o que dá segurança e coragem ao grupo.
- Testes de unidades *(Unit tests)*
  - escritos pelos programadores para testar cada elemento do sistema (e.g., cada método em cada caso).
- Testes de funcionalidades *(Functional tests)*
  - escritos pelos clientes para testar a funcionalidade do sistema.

Testes

- Testes das unidades do sistema
  - Tem que estar sempre funcionando a 100%.
  - São executados várias vezes por dia.
  - Executados à noite automaticamente.
- Testes das funcionalidades
  - Vão crescendo de 0% a 100%.
  - Quando chegam a 100% acabou o projeto.

O Código

- Padrões de estilo adotados pelo grupo inteiro.
- O mais claro possível.
  - XP não se baseia em documentações detalhadas e extensas (perde-se sincronia).
- Comentários sempre que necessários.
- Comentários padronizados.
- Programação Pareada ajuda muito!

Programação em Pares

- Erro de um detectado imediatamente pelo outro (grande economia de tempo).
- Maior diversidade de idéias, técnicas, algoritmos.
- Enquanto um escreve, o outro pensa em contra-exemplos, problemas de eficiência, etc.
- Vergonha de escrever código feio (*gambiarras*) na frente do seu par.
- Pareamento de acordo com especialidades.
  - Ex.: Sistema Multimídia Orientado a Objetos

Propriedade Coletiva do Código

- Modelo tradicional: só autor de uma função pode modificá-la.
- XP: o código pertence a todos.
- Se alguém identifica uma oportunidade para simplificar, consertar ou melhorar código escrito por outra pessoa, que o faça.
- Mas rode os testes!

Refatoração *(Refactoring)*

- Uma [pequena] modificação no sistema que não altera o seu comportamento funcional
- mas que melhora alguma qualidade não-funcional:
  - simplicidade
  - flexibilidade
  - clareza
  - desempenho

Exemplos de Refatoração

- Mudança do nome de variáveis
- Mudanças nas interfaces dos objetos
- Pequenas mudanças arquiteturais
- Encapsular código repetido em um novo método
- Generalização de métodos
  - **`raizQuadrada(float x)⇒ raiz(float x, int n)`**

Sobre Refatoração

- Catálogo de Refatorações de Martin Fowler:
  - *Refactoring: Improvng the Design of Existing Code*. Addison-Wesley, 2000.
- Ferramentas para refatoração automatizada embutidas no Eclipse, Smalltalk Browser, Emacs, Visual Studio, etc., etc.
- **`www.refactoring.com`**

Cliente

- Responsável por escrever “histórias”.
- Muitas vezes é um programador ou é representado por um programador do grupo.
- Trabalha no mesmo espaço físico do grupo.
- Novas versões são enviadas para produção todo mês (ou toda semana).
- *Feedback* do cliente é essencial.
- Requisitos mudam (e isso não é mau).

Coach (treinador)

- Em geral, o mais experiente do grupo.
- Identifica quem é bom no que.
- Lembra a todos as regras do jogo (XP).
- Eventualmente faz programação em pares.
- Não desenha arquitetura, apenas chama a atenção para oportunidades de melhorias.
- Seu papel diminui à medida em que o time fica mais maduro.

Tracker (Acompanhador)

- A “consciência” do time.
- Coleta estatísticas sobre o andamento do projeto. Alguns exemplos:
  - Número de histórias definidas e implementadas.
  - Número de *unit tests.*
  - Número de testes funcionais definidos e funcionando.
  - Número de classes, métodos, linhas de código
- Mantém histórico do progresso.
- Faz estimativas para o futuro.

Quando XP Não Deve Ser Experimentada?

- Quando o cliente não aceita as regras do jogo.
- Quando o cliente quer uma especificação detalhada do sistema antes de começar.
- Quando os programadores não estão dispostos a seguir (todas) as regras.
- Se (quase) todos os programadores do time são medíocres.

Quando XP Não Deve Ser Experimentada?

- Grupos grandes (>10 programadores).
- Quando *feedback* rápido não é possível:
  - sistema demora 6h para compilar.
  - testes demoram 12h para rodar.
  - exigência de certificação que demora meses.
- Quando o custo de mudanças é essencialmente exponencial.
- Quando não é possível realizar testes (muito raro).

Conclusão XP
Vencendo os Medos

- Escrever código.
- Mudar de idéia.
- Ir em frente sem saber tudo sobre o futuro.
- Confiar em outras pessoas.
- Mudar a arquitetura de um sistema em funcionamento.
- Escrever testes.

As 12 Práticas de XP
(versão 2000)

1. Planejamento
2. Fases Pequenas
3. Metáfora
4. Design Simples
5. Testes
6. Refatoramento
7. Programação Pareada
8. Propriedade Coletiva
9. Integração Contínua
10. Semana de 40 horas
11. Cliente junto aos desenvolvedores
12. Padronização do código

Práticas de XP

- As práticas foram refatoradas (veja www.extremeprogramming.org/rules.html)
- Mas a idéia é exatamente a mesma

- Novas práticas interessantes:
  - Conserte XP quando a metodologia quebrar.
  - Mova as pessoas.
  - Client Proxy (by Klaus)
  - Líbero (by Klaus)

(Ainda) Conclusão XP

- XP não é para todo mundo.
- Mas todo mundo pode aprender com ela.

Ex.: XP@Scrum™

- Aplicando XP com Scrum
  - Scrum: método de gerenciamento
  - XP: conjunto de práticas
- Vantagens
  - Objetivos por *Sprint* e não por história
  - Escalabilidade
  - Fácil implementação (alegadamente)

Feature Driven Development (FDD)

Introdução

- **Desenvolvimento Ágil**

(+)	(-)
Indivíduos e Interações	Processos e Ferramentas
Software Funcionando	Documentação Extensa
Colaboração do Cliente	Negociação de Contrato
Resposta às Mudanças	Seguir um Plano

Tabela 1 – Manifesto Desenvolvimento Ágil

Introdução

- **Feature Driven Development**
  - Criado por Jeff De Luca em 1999, baseado em 30 anos experiência na área de TI com envolvimento de Peter Coad e Steve Palmer;
  - Trata-se de uma compilação dos "*patterns of success*" percebidos;
  - Possui características de Desenvolvimento Ágil;
  - Iterações pequenas (no máximo 2 semanas)
  - Centrada em *Design*.

"We cannot predict where a chess game will go, but we can learn patterns of play that bring success." ***[Highsmith 2000].***

"Good habits are a wonderful thing. They allow the team to carry out the basic steps, focusing on content and results , rather than process steps . This is best achieved when process steps are logical and their worth immediately obvious to each member."

Coad, LeFebvre, De Luca [Coad 99]

Motivação

- **Clientes** têm resultados rápidos e relatório do status numa linguagem que eles entendem
- **Gerentes de projeto** têm uma visão completa e exata do status do projeto
- **Desenvolvedores** conseguem trabalhar em novas coisas em poucos dias e ficam mais envolvidos em análise, projeto e codificação

Melhores Práticas

- **Modelagem de Objeto do Domínio**
  - Diagramas de classes dos objetos do domínio e seus relacionamentos são construídos;
  - O suporte aos aspectos comportamentais é dado por diagramas de seqüência em alto nível;
  - Ênfase não é em cima de se determinar todos os atributos;
  - Evita as inferências isoladas dos analistas, por se tratar de um modelo global onde todos os envolvidos participam.

Modelagem De Objeto Do Domínio

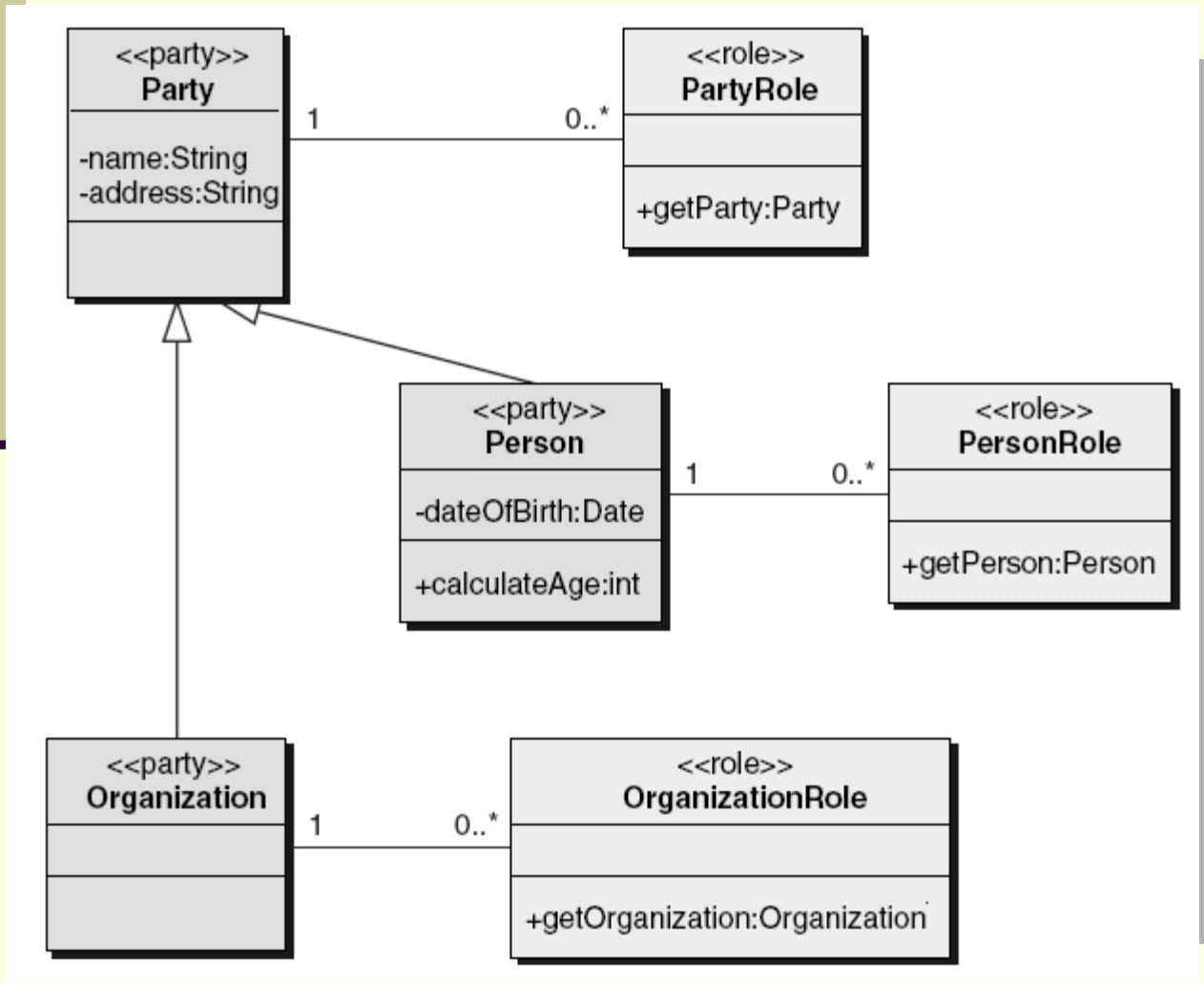


Figura 1 – Modelo de Objetos Do Domínio numa Fase Inicial

"Extending that analogy a bit further, a domain object model is like the road map that guides the journey; with it, you can reach your destination relatively quickly and easily without too many detours or a lot of backtracking; without it, you can very quickly end up lost or driving around in circles, continually reworking and refactoring the same pieces of code."

Stephen Palmer

Desenvolvimento Por *Feature*

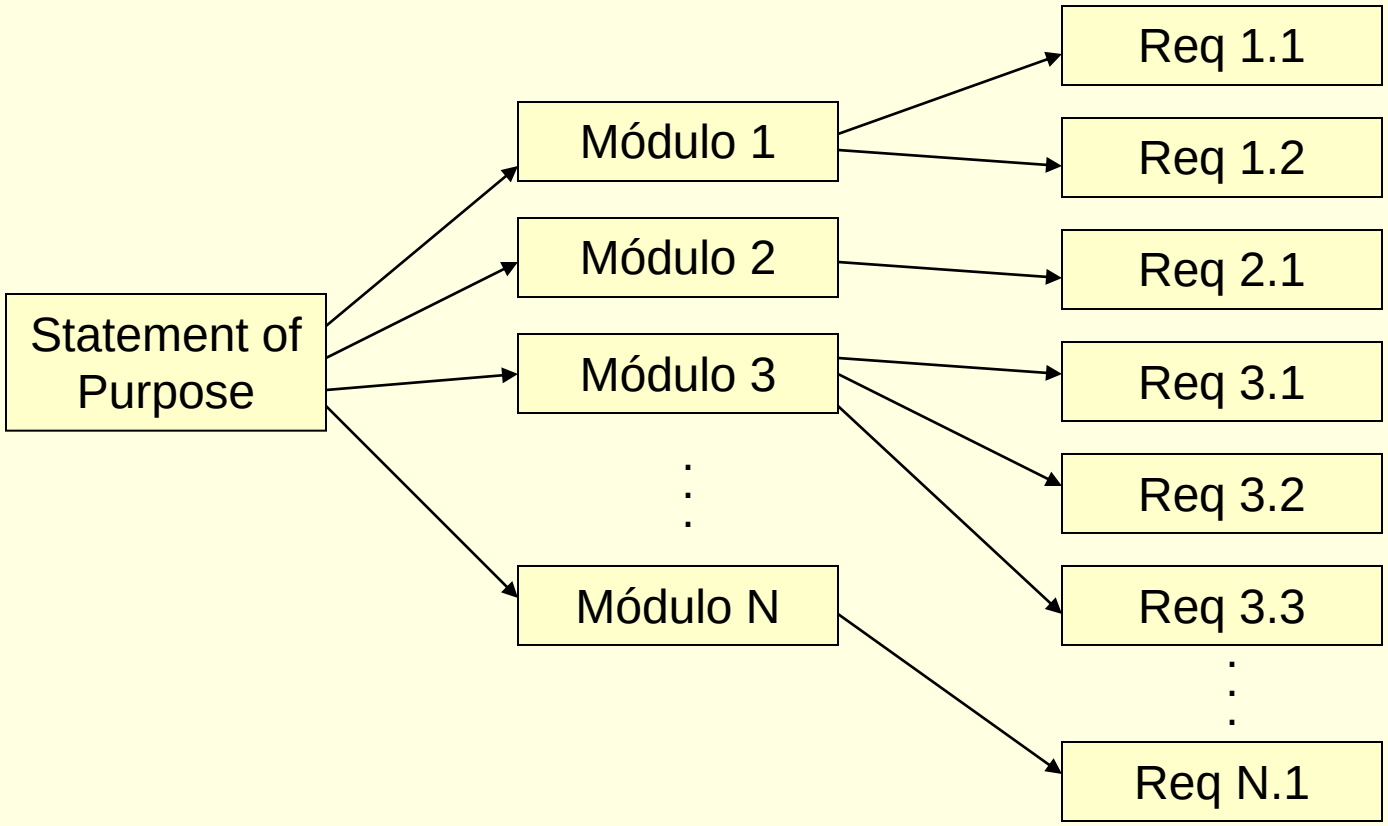


Figura 2 – Decomposição do *Statement Of Purpose* em Requisitos Funcionais

Feature

- *Features* são os requisitos funcionais com valor para o usuário, isto é, estão numa linguagem que o cliente ou usuário podem entender;
- Os requisitos funcionais tendem a misturar funções de interface, persistência e comunicação em rede com funções de negócio;
- Facilitam o acompanhamento da evolução do projeto (o quanto já foi feito) junto ao cliente.

"Once we have identified the classes in our domain object model, we can design and implement each one in turn. Then, once we have completed a set of classes, we integrate them and hey, presto! We have part of our system. Easy!...Well, it's a nice dream!"

"We can produce the most elegant domain object model possible, but if it does not help us to provide the system's clients with the functionality for which they have asked, we have failed. It would be like building a fantastic office skyscraper but either leaving each floor unfurnished, uncarpeted, and without staff, or furnishing it with ornamental but impractical furniture and untrained staff."

A Practical Guide to FDD, cap 3

Feature

- O termo *feature* em FDD é muito específico. Uma feature é uma função pequena, com valor no cliente expressa na seguinte forma:
  - *<action> <result> <object>*

  Com as proposições apropriadas entre ação, resultado e objeto.
  - Ex.: - Calcular o *total* de uma venda;
    - Avaliar o *desempenho* de um vendedor.
    - Validar a *senha* de um usuário.
- *Features* vs. Casos de Uso
  - Classes com funções pesadas constantemente acessando classes com dados pesados
  - Alto acoplamento
  - Baixa coesão
  - Encapsulamento pobre

Feature

- **Exemplo**
  - **Área de *Features* Principal (*Subject Domain Area*)**
    - Gerenciamento de venda de produtos
  - **Conjunto de *Features (Business Activity)***
    - Vender para um cliente
  - ***Features***
    - Calcular o total de vendas
    - Calcular o total de compras de um cliente
    - Estimar o tempo de entrega de uma venda
    - Calcular a taxa de uma venda

Class (Code) Ownership

- *Class (code) ownership* num processo de desenvolvimento denota quem é em última instância responsável pelo conteúdo de uma classe (parte de código);
- Pode ser:
  - **Individual** – a classe é atribuída somente a um único dono;
  - **Coletiva** – O time possui coletivamente a classe (código).

Individual Class Ownership

- **Vantagens**
  - Um guardião de integridade conceitual;
  - Um expert para explicar uma parte do código;
  - Um desenvolvedor mais rápido para implementar essa parte de código;
  - "*É minha classe, dela me orgulho*!!!"
- **Desvantagens**
  - Mudanças que dependem de outras classes;
  - Perda do conhecimento por ocasião da falta do *class owner.*

Collective Class Ownership

- **Vantagens**
  - Resolve o problema da espera por outro para solução de um problema;
- **Desvantagens**
  - Facilmente degenera em *não tem dono* ou classes governadas por uma “*elite*”.

Equipes por *Feature*

- Como se organiza os *class owners* para construir as *features*?

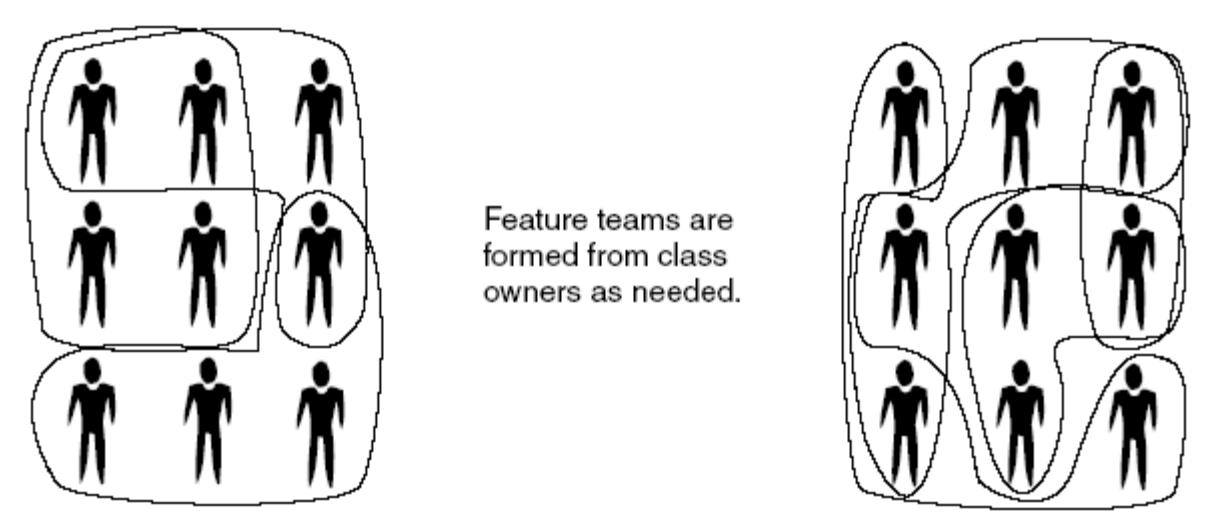


Figura 3 – Equipes por *Feature*

Equipes por *Feature*

- Normalmente pequena: de três a seis integrantes;
- Possui todo o código que necessita para mudar sua *feature;*
- Cada membro de uma *feature* contribui para o *design* e implementação de uma *feature* sob a orientação de um desenvolvedor mais experiente.

Inspeções

- Usadas para:
  - Detecção de Defeitos;
  - Transferência de Conhecimento;
  - Aderência à Padrões de Codificação;
  - Extrair Métricas para Melhoria do Processo.
- Cuidado: Não devem ser tomadas como uma avaliação pessoal de *performance*.

Builds Regulares

- Em intervalos regulares, o sistema completo é montado;
- Identifica os erros de integração;
- Garante um sistema "*up-to-date*" para demonstração ao cliente
- Um processo de geração de *builds* pode ser melhorado para:
  - Gerar documentação;
  - Rodar script de métricas e de auditoria;
  - Base para testes de regressão;
  - Construir novo *build* e *release notes*, listando novas features adicionadas, defeitos corrigidos, etc...

Gerência de Configuração

- Identificação do código para todas as *features* completadas;
- Manutenção de um histórico de mudanças das classes.

Comunicação/Visibilidade Dos Resultados

- **Progress Reporting**
  - Facilita o acompanhamento gerencial, através da coleta de informações confiáveis e precisas sobre o status do projeto;
  - Sugere um número de formatos de relatórios intuitivos e diretos para ilustrar o progresso.

Closely related to project control is the concept of "visibility," which refers to the ability to determine a project's true status....If the project team can't answer such questions, it doesn't have enough visibility to control its project.

The working software is a more accurate status report than any paper report could ever be.

Steve McConnell [McConnell 98]

Comunicação/Visibilidade Dos Resultados

Id	Description	Chief Prog.	Class Own.	Walkthrough		Design		Design Inspection		Code		Code Inspection		Promote to Build	
				Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual
MD125	validate *the transactional limits of a CAO against an* implementation instruction	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD126	set *the status of an* implementation instruction	CP	ABC	23/12/98		31/01/99		01/02/99		10/02/99		18/02/99		20/02/99	
MD127	specify *the authorising officer of an* implementation instruction	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD128	reject *an implementation instruction authorisation for a* set of lines	CP	ABC	STATUS: Inactive REMARKS: [added by CK: 3/2/99] No longer applicable											
MD129	confirm *an implementation instruction authorisation for a* set of lines	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD130	determine *if all documents are completed for a* borrower	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99		08/02/99		10/02/99	
				REMARKS: [added by SL: 3/2/99] Blocked on AS [added by SL: 3/2/99] Requirement clarification before upper management											
MD131	validate the *transactional limits of a CAO against a* disbursement instruction	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99		08/02/99		10/02/99	
				REMARKS: [added by SL: 3/2/99] Behind schedule - sick leave											
MD132	submit *for authorisation an* implementation instruction	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99	05/02/99	06/02/99	06/02/99	08/02/99	08/02/99
MD133	validate the *drawdown date of an* implementation instruction	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99	05/02/99	06/02/99	06/02/99	08/02/99	08/02/99

KEY: Work In Progress | Attention | Completed | Not Started

Comunicação/Visibilidade Dos Resultados

Implementation (MD)

Feature Sets	Total Features	Not Started	In Progress	Behind Schedule	Completed	Inactive	% Completed	Completion Date
Authorisation	15	0	0	13	2	2	51	Mar 1999
Change Current Account for OD Line	12	12	0	0	0	0	0	Mar 1999
Establish Disbursement Details	35	0	0	4	31	1	91	Mar 1999
Establish Implementation Instruction	27	1	0	12	14	1	70	Mar 1999
Establish Line Implementation Details	21	0	0	4	17	0	89	Mar 1999
Establish a Loans, HP, LeasingLoans, BlockDiskcouting Line Impl.	7	0	0	0	7	0	100	Mar 1999
Establish a NIF RL Disbursement	8	6	0	0	2	1	25	Mar 1999
Establish a Trade or Gtee Line Implementation	9	0	0	0	9	0	100	Mar 1999
Establish an FX Line Implementation	5	0	0	0	5	0	100	Mar 1999
Establish an OD Line Implementation	13	0	0	0	13	0	100	Mar 1999
New Feature	3	0	0	0	3	0	100	Ongoing
Subject Area Total	**155**	**19**	**0**	**33**	**103**	**5**	**74**	

Progress Summary for Problem Domain

Total Features	Not Started	In Progress	Behind Schedule	Completed	Inactive	% Completed
1013	**23**	**0**	**33**	**957**	**100**	**96**

Comunicação/Visibilidade Dos Resultados

Status Geral:

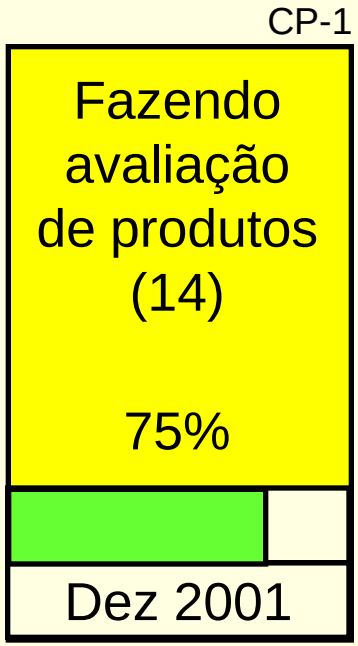
- **Trabalhos em progresso**
- **Atenção (ie, atrasado)**
- **Completo**
- **Não iniciado**

Porcentagem completa:

- **Barra de progresso**

Status Completo:

- **Completo**
- MY **Mês de conclusão**



Exemplo:

Conjunto de *Features*: Fazendo avaliação de produtos – Trabalho em progresso

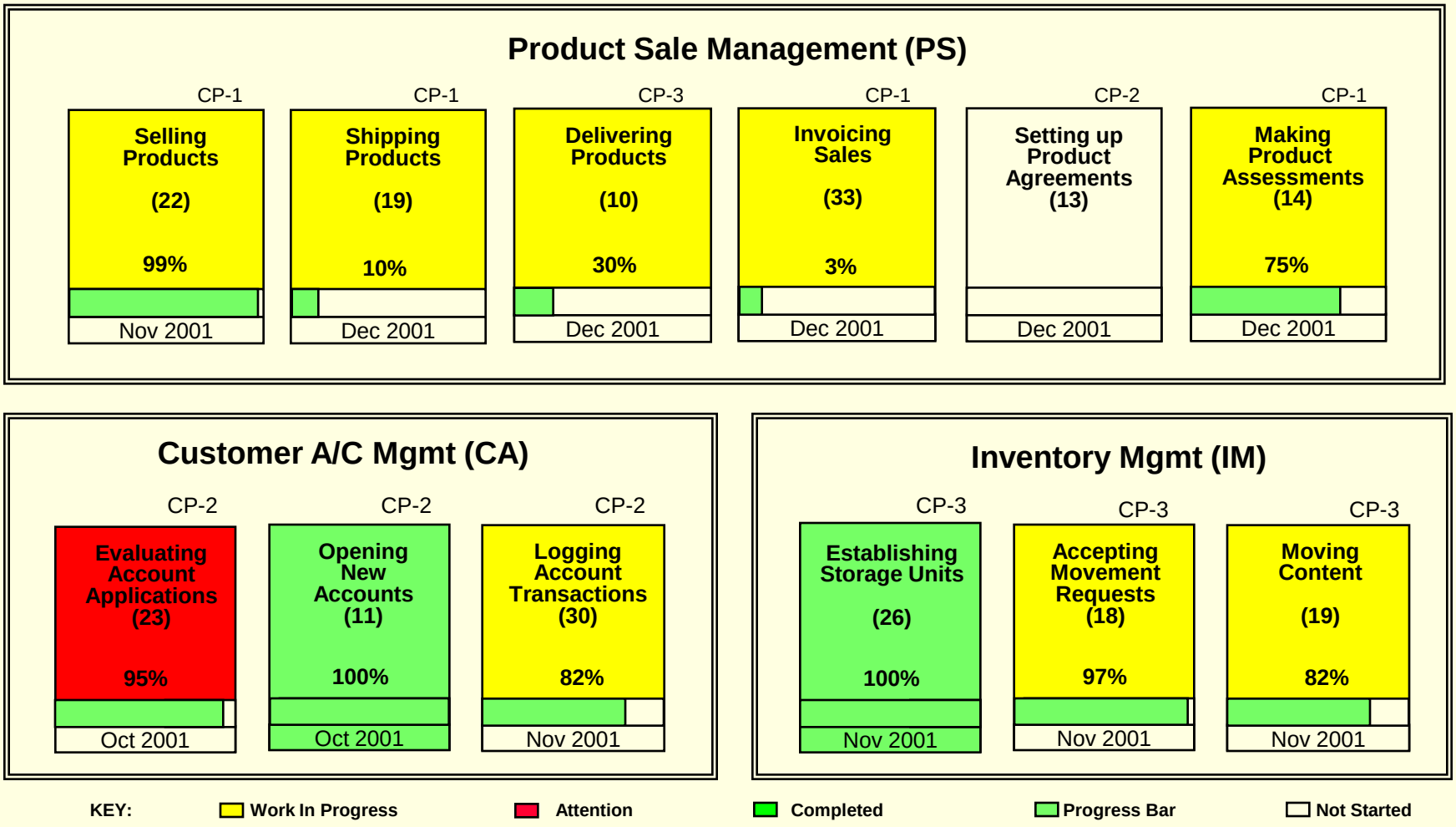
CP-1 é o programador chefe inicial

(14) esse conjunto de *Features* possui 14 características

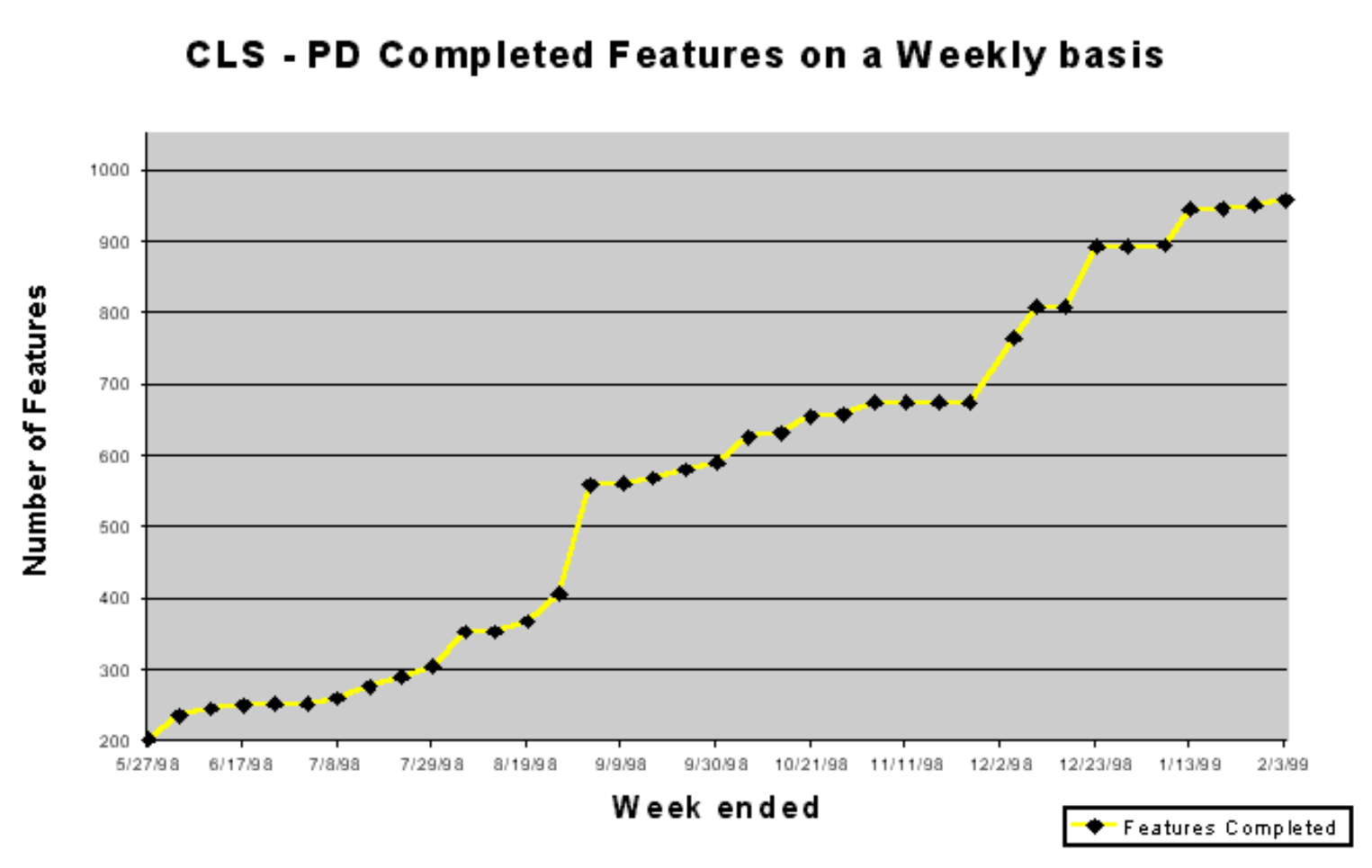
Conjunto de *Features/* está 75% completado

A conclusão é para dezembro de 2001

Comunicação/Visibilidade Dos Resultados



Comunicação/Visibilidade Dos Resultados



Processo

- **Descrição**
  - Cada processo é descrito em não mais do que duas páginas de papel tamanho carta, frente-e-verso;
  - Cada descrição do processo apresenta-se de acordo com a estrutura: **Entrada, Tarefas, Verificação e Saídas (ETVX).**

Processo

- **Papéis principais**
  - Gerente de projeto
  - Arquiteto chefe
  - Especialistas no domínio
  - Gerentes de desenvolvimento
  - Programadores chefes
  - Proprietários de classes

Processo

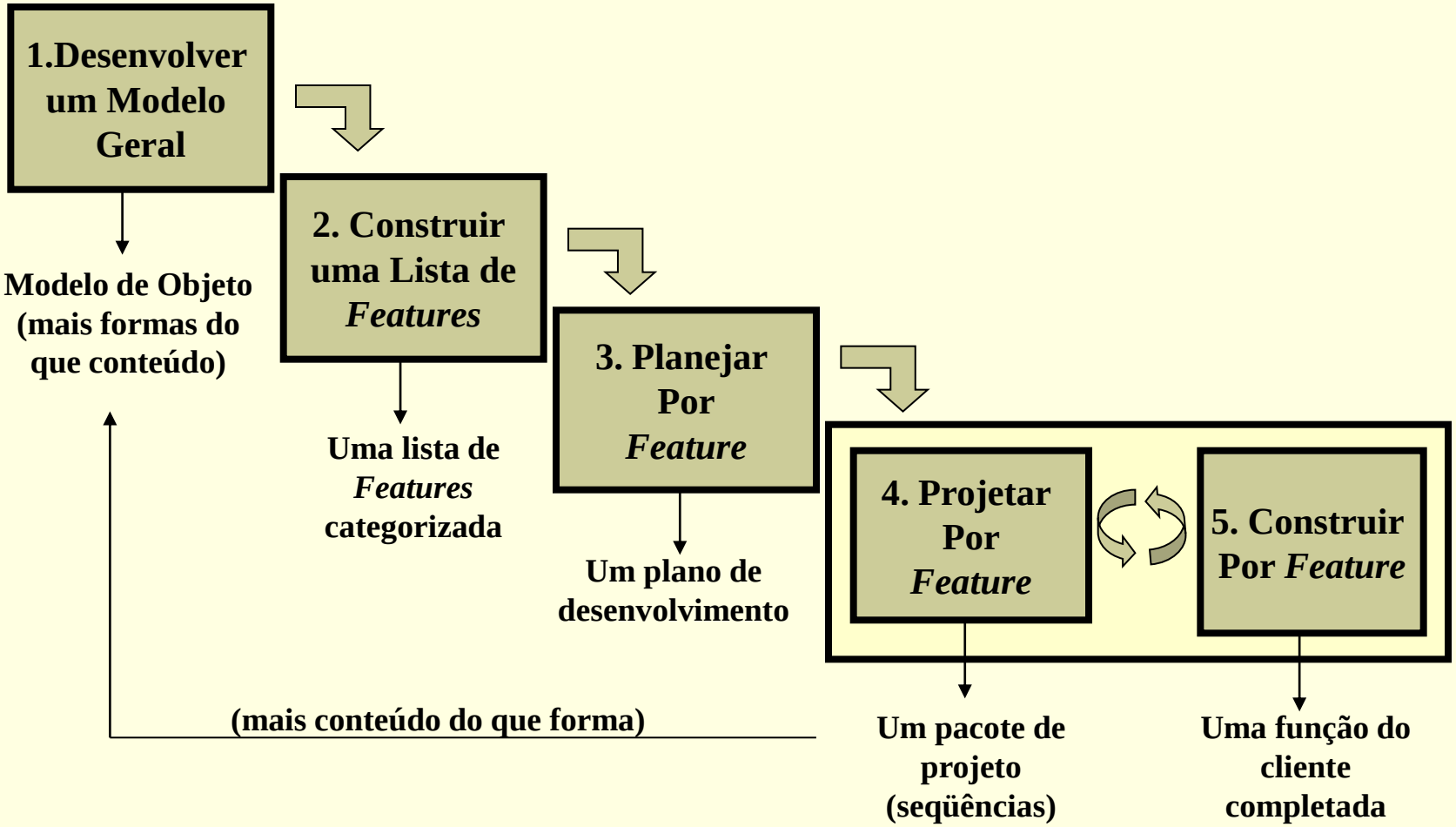
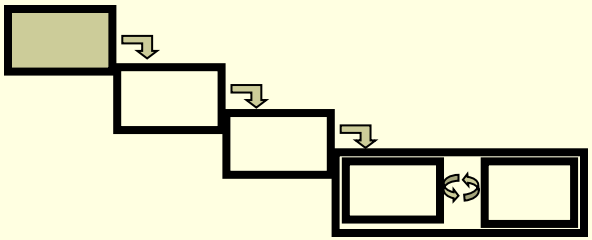


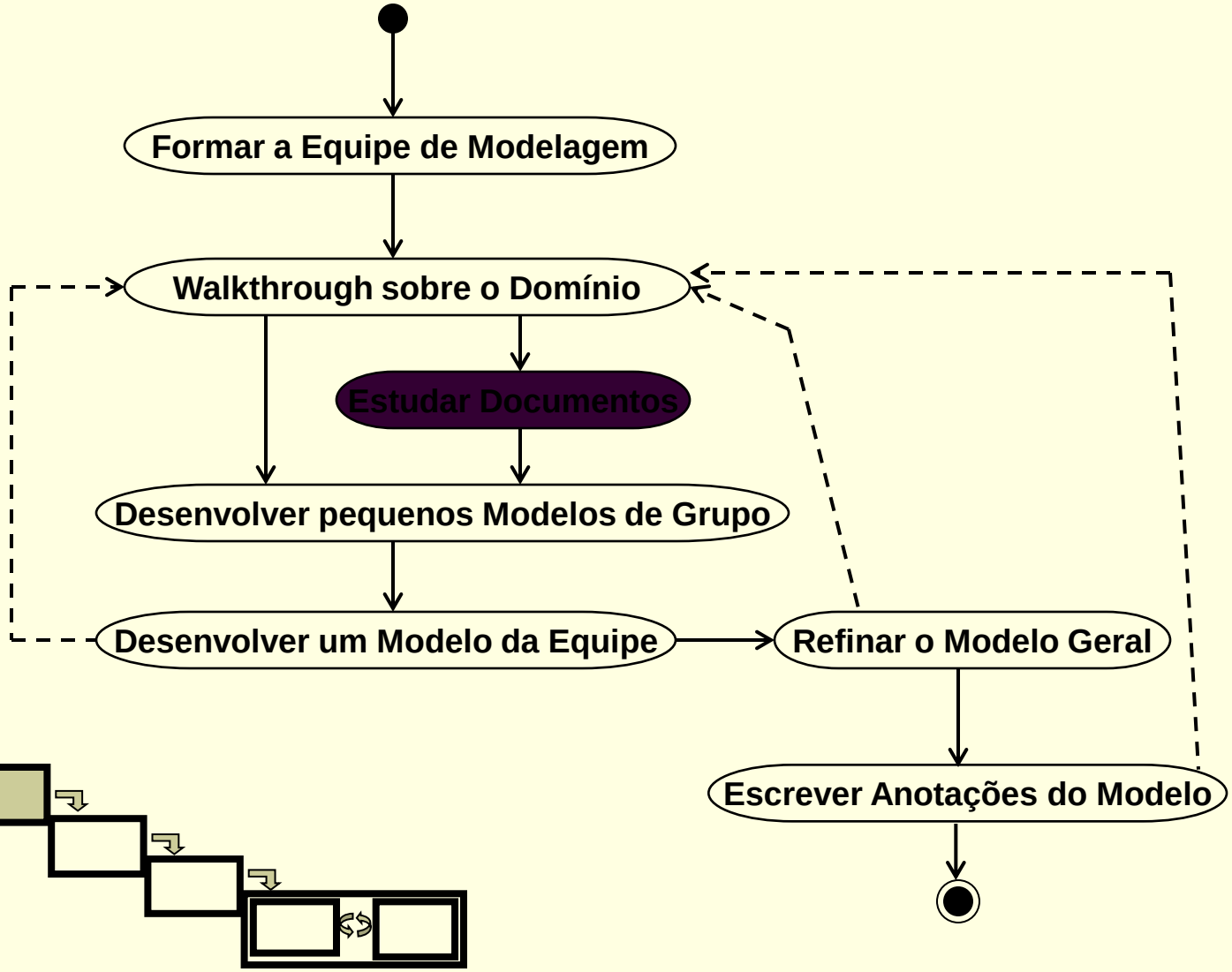
Figura 4 – Os 5 Processos FDD

Desenvolver um Modelo Geral

- Adquirir conhecimento do domínio e construir o modelo geral
  - Estabelecimento do “propósito de negócio” do novo sistema;
  - Construção de um “modelo conceitual” do sistema.

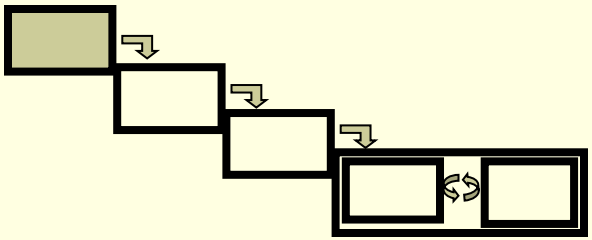


Atividades


Formar a Equipe de Modelagem
Walkthrough sobre o Domínio
Estudar Documentos
Desenvolver pequenos Modelos de Grupo
Desenvolver um Modelo da Equipe
Refinar o Modelo Geral
Escrever Anotações do Modelo

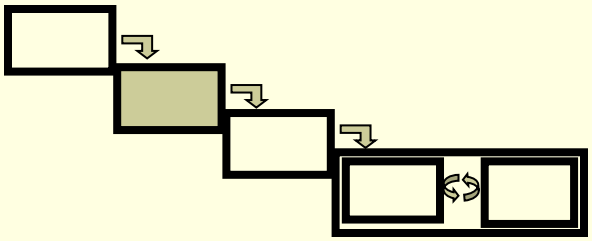
Entradas e Saídas

- **Entrada**
  - Especialistas no domínio, programadores e arquitetos chefes são selecionados.
- **Saídas**
  - Modelo geral do domínio;
  - Diagrama das classes principais com alguns métodos e atributos identificados;
  - Diagramas de seqüência de algumas funcionalidades mais complexas (se houver);
  - Comentário sobre o modelo.

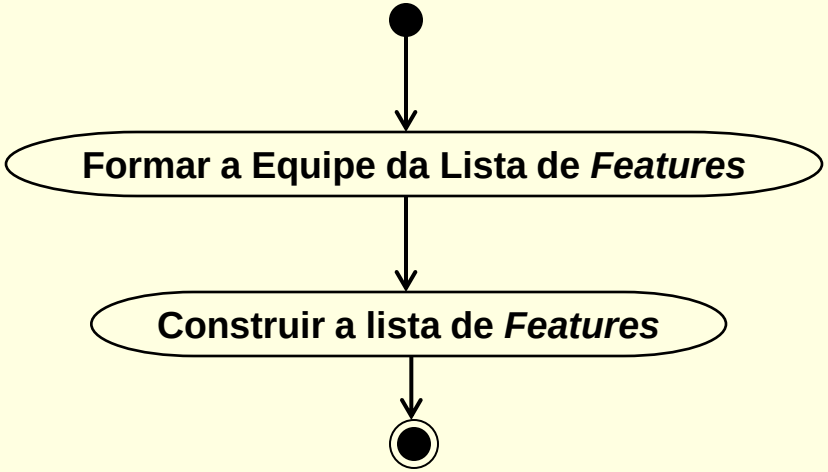


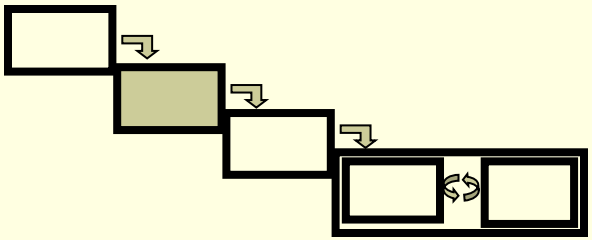
Construir lista de *Features*

- O domínio é decomposto até chegar nas *Features;*
- *Features* são agrupadas e categorizadas;
- *Features* são granuladas até ser necessário menos de 2 semanas pro seu desenvolvimento.



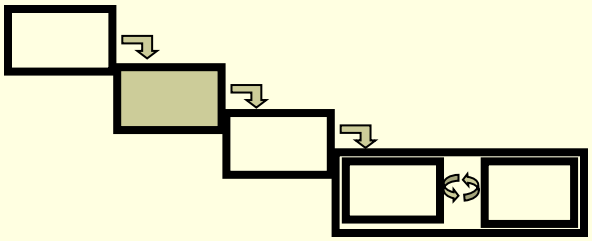
Atividades


Formar a Equipe da Lista de *Features*
Construir a lista de *Features*



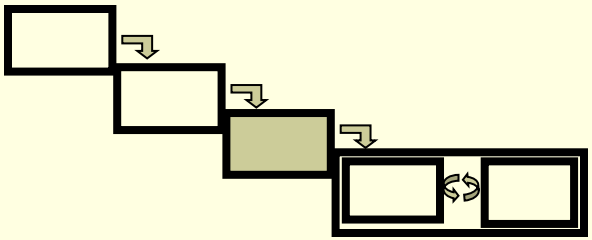
Entradas e Saídas

- **Entrada**
  - O processo de desenvolvimento do modelo geral ter sido concluído com sucesso.
- **Saídas**
  - Uma lista das áreas do domínio identificadas;
  - Para cada área, uma lista de atividades de negócio (conjunto de *Features*);
  - Para cada atividade, os passos a serem realizados (*Features*).

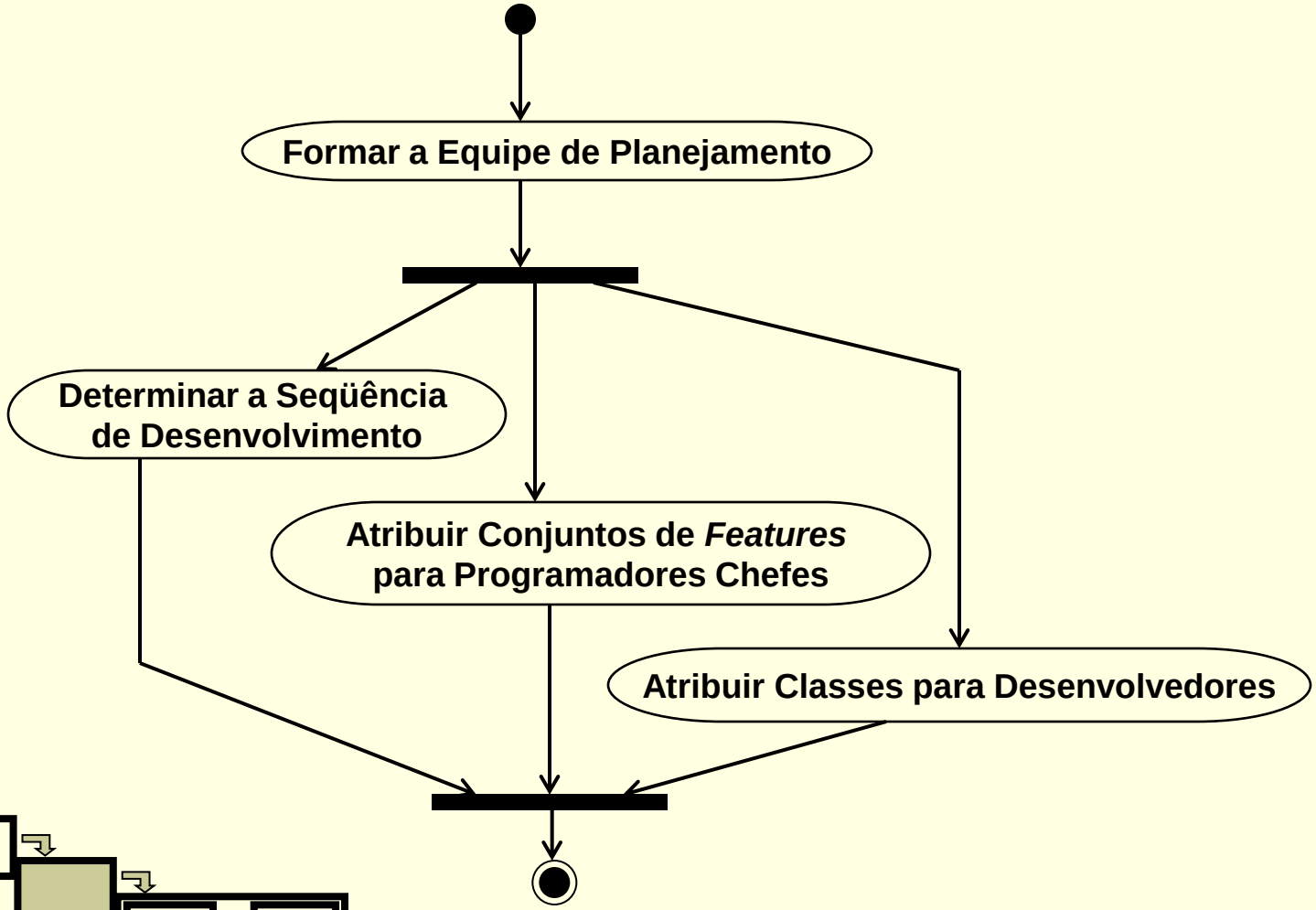


Planejar Por *Features*

- Uma data de lançamento é estabelecida para o release inicial;
- A lista de *Features* priorizadas é refinada;
  - Dependência entre funcionalidades
  - Carga de Trabalho
  - Complexidade ou Risco
  - Milestones
- O trabalho técnico é planejado e atribuído – plano de desenvolvimento

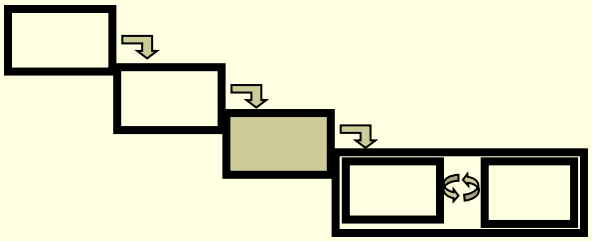


Atividade


Formar a Equipe de Planejamento
Determinar a Seqüência de Desenvolvimento
Atribuir Conjuntos de *Features* para Programadores Chefes
Atribuir Classes para Desenvolvedores

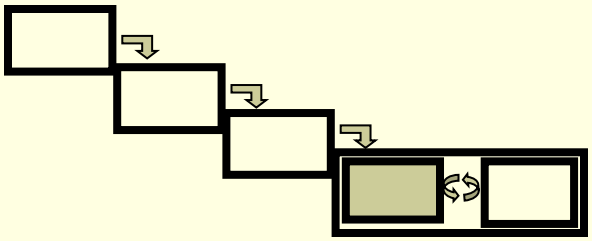
Entradas e Saídas

- **Entrada**
  - O processo de construir a lista de *Features* ter sido concluído com sucesso.
- **Saídas**
  - Áreas de Domínio com datas de término;
  - Atividades de negócio com datas de término;
  - Programadores-chefes atribuídos a atividades de negócio;
  - A lista de classes e seus donos (desenvolvedores).

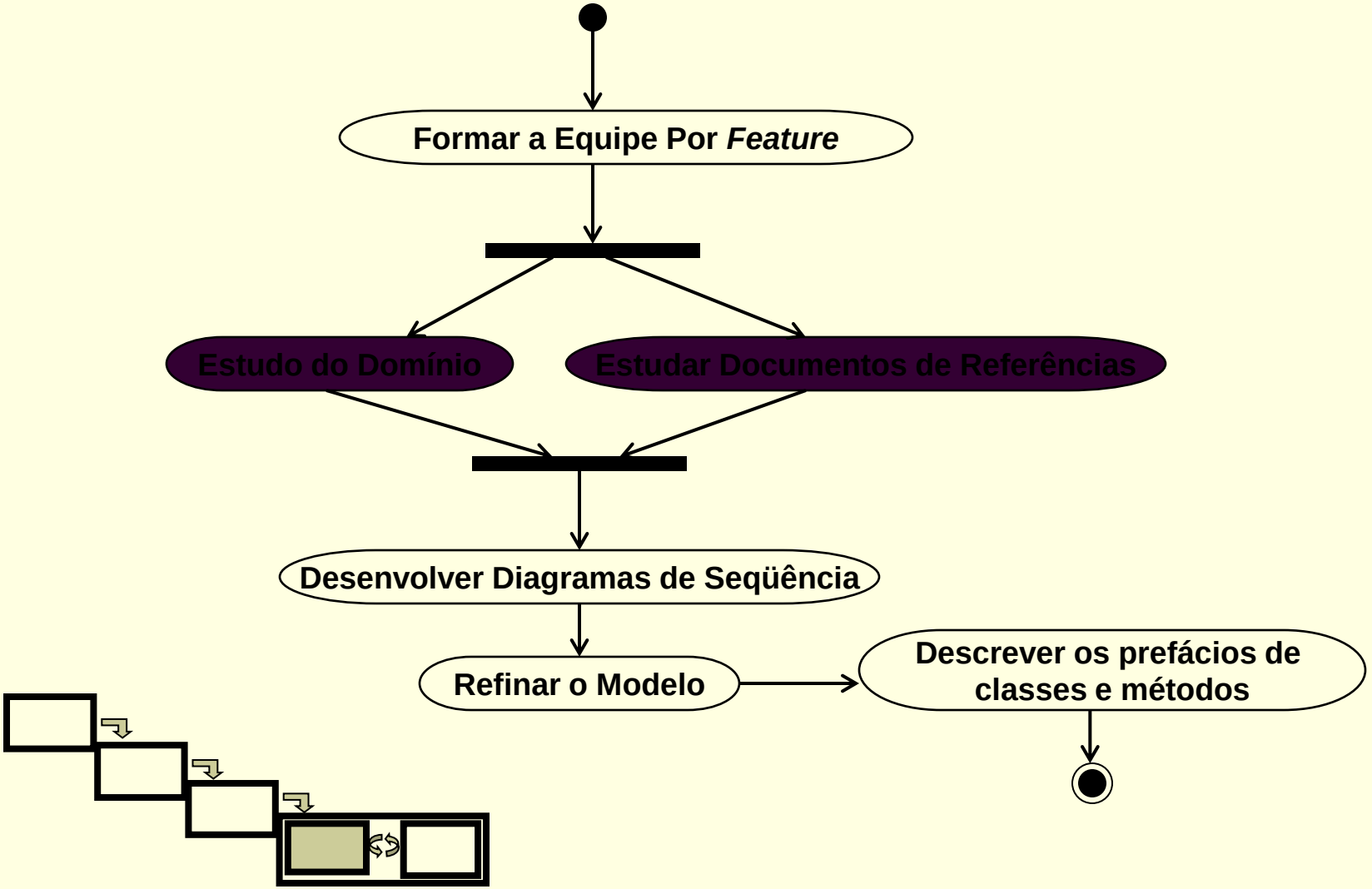


Projetar Por *Features*

- Regras e transações são identificadas
- O modelo da interface do usuário é esboçado
- Diagramas de seqüência mais detalhados são produzidos
- Especialistas são consultados para descobrir qualquer necessidade específica adicional

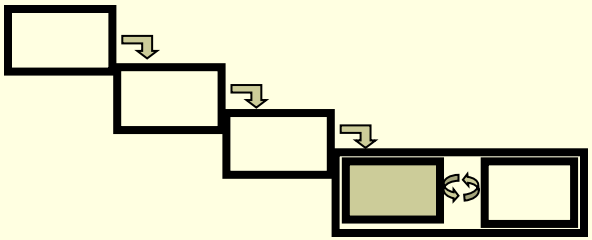


Atividades


Formar a Equipe Por *Feature*
Estudo do Domínio
Estudar Documentos de Referências
Desenvolver Diagramas de Seqüência
Refinar o Modelo
Descrever os prefácios de classes e métodos

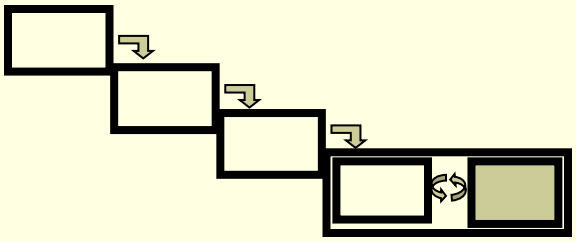
Entradas e Saídas

- **Entrada**
  - O processo de planejamento ter sido concluído com sucesso
- **Saídas**
  - Diagramas de seqüência
  - *Designs* alternativos (caso exista)
  - O modelo de objeto com classes, métodos e atributos novos ou atualizados
  - A documentação da API do sistema
  - Lista de tarefas (calendário/ To-Do)

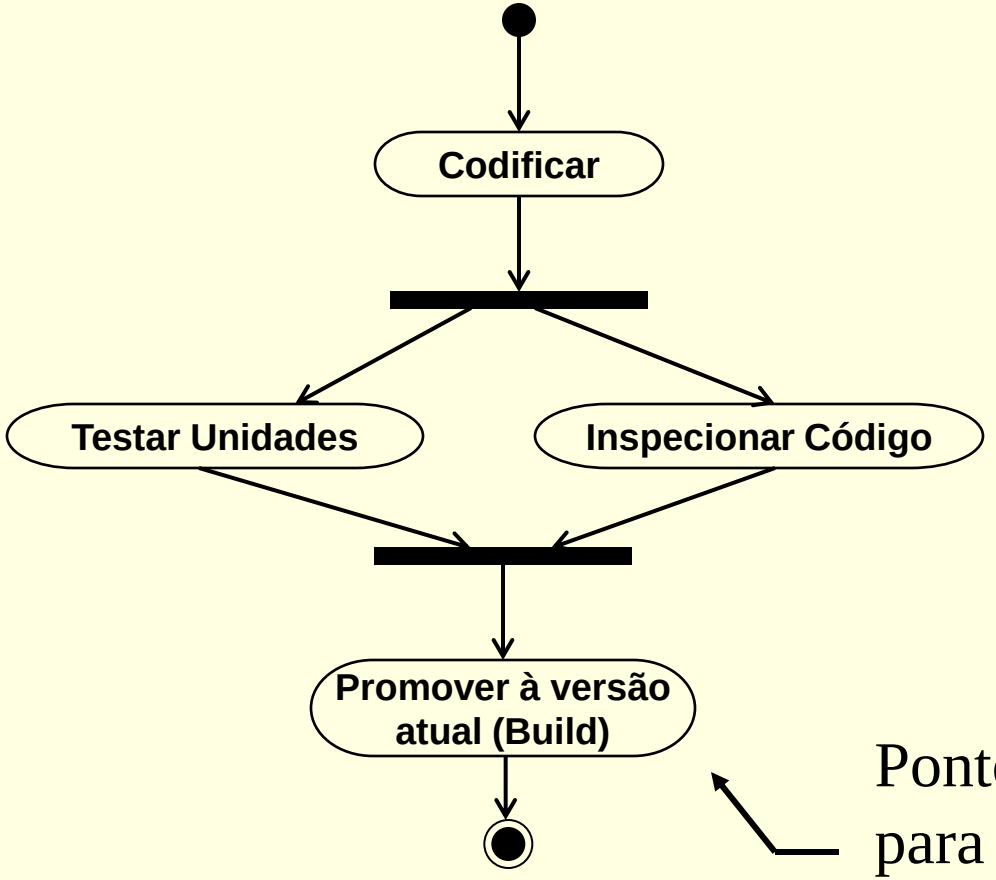


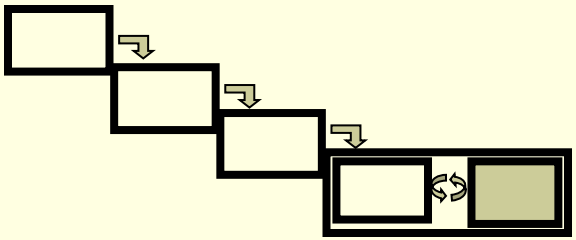
Construir Por *Feature*

- *Features* são construídas implementando todas as classes e métodos necessários
- Testes de unidades
- *Features* são inseridas no *build* quando o teste resulta em sucesso



Atividades

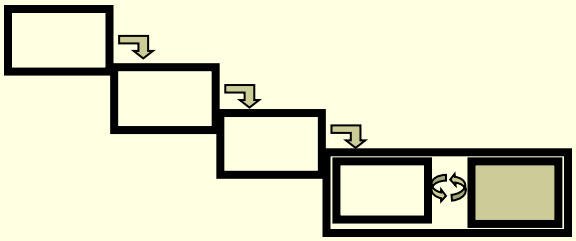




Ponto de integração para a funcionalidade inteira

Entradas e Saídas

- **Entrada**
  - O processo de construção por *Feature* ter sido concluído com sucesso
- **Saídas**
  - Classe(s) e/ou método(s) que passaram na inspeção de código com sucesso
  - Classes inseridas no *build*
  - A conclusão da funcionalidade do cliente



Conclusão dos FDDs

- Requer alguma arte na alocação de recursos;
- Permite um bom acompanhamento gerencial do status das atividades;
- Mais leve e simples de implementar;
- Favorece um bom *Design* do sistema.

Características Comuns dos Métodos Ágeis

- Coloca o foco
  - Na entrega freqüente de sub-versões do software funcionando para o cliente.
  - Nos seres humanos que desenvolvem o software.
- Retira o foco de
  - Processos rígidos e burocratizados.
  - Documentações e contratos detalhados.
  - Ferramentas que são usadas pelos seres humanos.