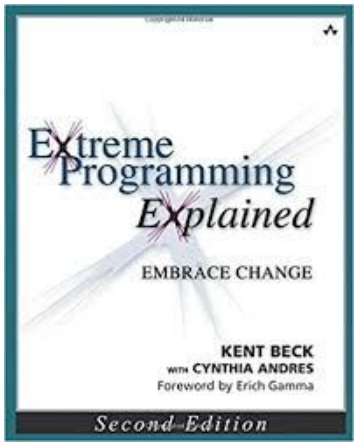
Extreme Programming (XP)

Prof. Marco Tulio Valente

mtov@dcc.ufmg.br

O que é XP?

- Um dos primeiros métodos ágeis, proposto por Kent Beck
  - Livro: Extreme Programming Explained
  - 1a edição: 1999; 2a edição: 2004 (com várias modificações)
  - Logo, a primeira versão foi escrita antes do "Manifesto Ágil" (2001)



O que é XP?

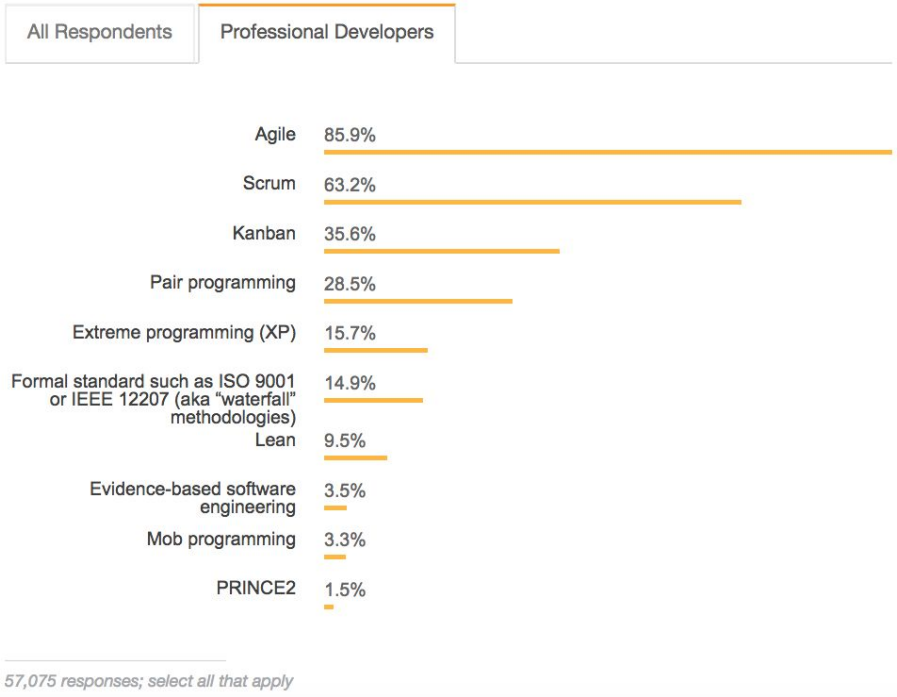
- Um método para desenvolvimento de software, baseada em conjunto de:
  - **Valores** (exemplos: comunicação, feedback, simplicidade, etc)
  - **Princípios** (exemplos: melhorias constantes, responsabilidade etc)
  - **Práticas** (exemplos: pair programming, user stories, iterações, integração contínua, build automatizados, test-first programming, design incremental)

O que é XP?

- Definição da 1a edição do livro de XP:
  - uma **metodologia leve**
  - recomendada para times pequenos e médios
  - que desenvolvem software que envolve **requisitos vagos ou sujeitos a mudanças frequentes** (isto é, sistemas do tipo B, na classificação do Bertrand Meyer, ver slides da Introdução)
- 2a edição do livro:
  - XP pode funcionar com times de qualquer tamanho

Stack Overflow Survey 2018

- Qual metodologia para desenvolvimento de software você costuma usar?



https://insights.stackoverflow.com/survey/2018/#career-values

Qual a diferença para Scrum?

- XP é um método ágil para desenvolvimento de software
- Scrum é um método ágil para gerenciamento de projetos, mas não necessariamente de software
- XP advoga o uso de diversas práticas específicas de desenvolvimento de software: pair programming, testes automatizados, user stories, integração contínua, test-first programming, etc
- Scrum não tem preocupação com práticas, pois seu objetivo é mais amplo.
- Mas ambos são ágeis, isto é, iterativos, incrementais, etc.
- Neste curso, ainda veremos um pouco de Scrum (após estudarmos XP).

XP vs outros métodos (tipicamente, plan-driven)

- Ciclos curtos de desenvolvimento (veja figura)
- Planejamentos incrementais
- Testes automatizados
- Design evolucionário
- Pouca documentação

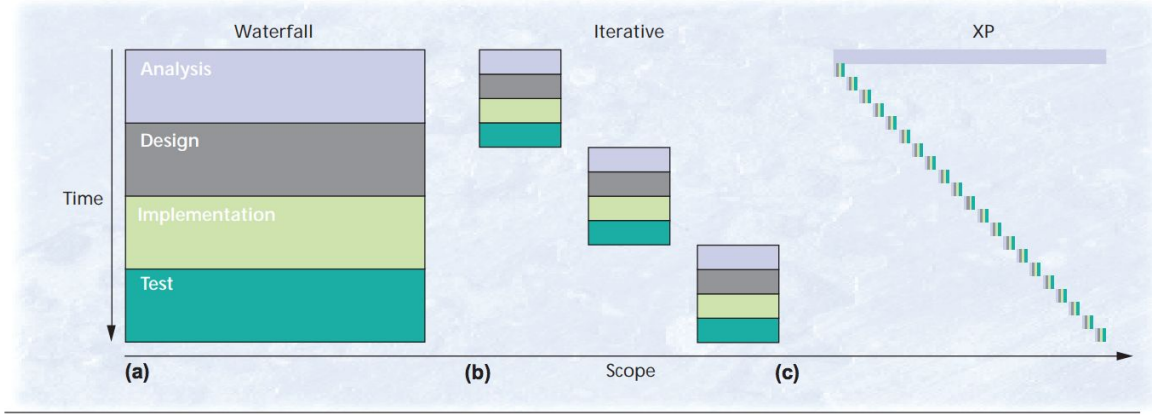


Figure 1. The evolution of the Waterfall Model (a) and its long development cycles (analysis, design, implementation, test) to the shorter, iterative development cycles within, for example, the Spiral Model (b) to Extreme Programming's (c) blending of all these activities, a little at a time, throughout the entire software development process.

Source: Embracing Change with Extreme Programming, K. Beck, IEEE Computer, 1999

Como XP controla riscos?

- Adotando ciclos curtos de desenvolvimento, pois clientes usualmente não sabem direito o que o sistema deve fazer
- Integrando pessoas das áreas de negócio aos times de desenvolvimento; essas pessoas escolhem e priorizam as features (stories) que devem ser implementadas primeiro
- Criando uma grande quantidade de testes; o que dá maior segurança aos times de desenvolvimento
- Garantindo que sistemas estão sempre prontos para serem liberados para uso (integração contínua)

Metáfora do Carro (famosa em XP)

- Desenvolver software é como dirigir um carro
- Pois dirigir um carro não é apontar em uma direção e seguir em frente, cegamente; em vez disso, dirigir exige várias correções de curso
- Na verdade, uma crítica a "plan-driven development", "big upfront design" etc

Observação Importante (citada no livro do K. Beck)

- Nenhum livro de jardinagem vai te transformar em um jardineiro:
  - Após ler um livro de jardinagem, você deve cuidar de seu próprio jardim, trocar ideias com uma comunidade de jardineiros, talvez ensinar outras pessoas a cuidar de jardins
  - Só então você pode se proclamar um jardineiro
- Neste curso, vamos estudar e discutir um "livro de jardinagem"
- E fazer alguns exercícios envolvendo a "criação de jardins", pequenos.
- Logo, vocês ainda não serão jardineiros; só após cuidarem de seu próprio jardim real, fazerem parte de uma comunidade de jardineiros, serem mentores de jardineiros, etc.

Observação Importante (2)

- Nenhuma organização usa 100% de um processo, tal como ele foi proposto
- Adaptações sempre vão ocorrer; com isso, não existem duas empresas diferentes, com o mesmo processo de desenvolvimento de software
- Ainda mais no caso de XP, que foi proposto há 20 anos

Observação Importante (3)

- Engenharia de Software, principalmente na parte de processos de software, não é uma "ciência exata"
- Isto é, não se deve esperar respostas e métodos objetivos, formais e quantificáveis, baseados em instrumentos matemáticos
- Mesmo comprovações empíricas são desafiadoras
  - Exemplo: vale a pena usar pair programming? Quais os custos? Quais as vantagens?

Observação Importante (4)

- Opiniões e divergências sobre processos de software tendem a ser pessoais, exageradas, algumas vezes até "religiosas"
    - Para isso, contribui o fato de o estudo de processos de software não ser uma "ciência exata, como afirmado no slide anterior
- Veja como Sommerville descreve o livro de XP:
    - "An **evangelical book** that describes the extreme programming process and extreme programming experiences. The author was the inventor of extreme programming and communicates his enthusiasm very well."

Valores, Princípios e Práticas

Valores

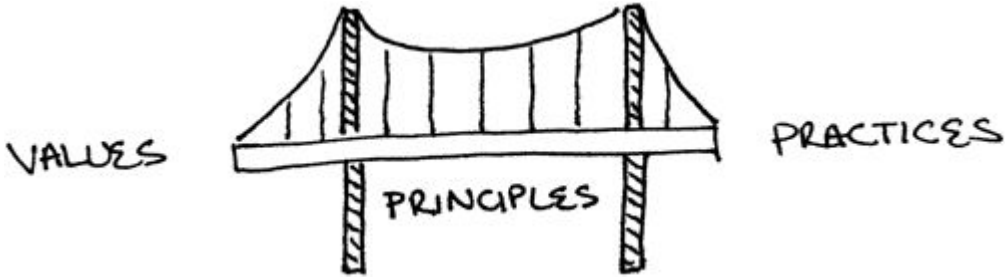
- **Importância de uma boa comunicação:** grande parte dos problemas de um projeto, são problemas de comunicação.
    - Quando você se deparar com um problema, verificar se ele foi causado por uma falha de comunicação? Qual estratégia de comunicação poderia ser usada para evitar esse problema no futuro?
- **Simplicidade**: qual é o sistema mais simples que funciona?
    - "Inside every large project, there is a small project struggling to get out"
- **Feedback constante**: pois em sistemas de software, dada à complexidade dos mesmos, pode ser difícil ter a solução "certa", logo de início;
    - Feedback é fundamental para se aproximar do sistema "certo"

Outros Valores

- Coragem, respeito
- E também qualidade de vida, previsibilidade etc
- Esses valores que justificam as práticas adotadas por XP (releases rápidos, integração contínua são práticas que fomentam "feedback constante")

Princípios

- São guidelines mais específicos e concretos, do que valores
- Princípios "conectam" valores às práticas
- Princípios de XP
  - Humanidade, Economics, Benefício Mútuo, Melhoria Contínua, Flow
  - Falhas, Baby Steps, Responsabilidade



Humanidade ("Humanity")

- Software é uma atividade 100% intelectual; desenvolvida em equipe
- XP é um método desenvolvido por "praticantes"; provavelmente, eles já sentiram "na pele" que o fator humano é fundamental em projetos de desenvolvimento de software

Peopleware

- Não é um princípio de XP, mas sim o título de um livro conhecido de Tom de Marco; bastante relacionado com o princípio de "humanidade"
- Maior causa de falhas de projetos de software é "política"
- Quando essas causas foram estudadas com mais detalhes, revelaram ser:
  - Problemas de comunicação, staffing
  - Desencantamento com o cliente, com o gerente, etc
  - Desencantamento com o salário, com a última promoção
  - Falta de motivação, excesso de trabalho, turnover alto etc.
- Em nenhum projeto, a causa do insucesso foi unicamente "técnica".

Economics

- Quem paga as contas tem que ser considerado
- Logo, todo o software tem que gerar valor para o seu "dono"

Benefício Mútuo

- Práticas tem que me beneficiar e também beneficiar outros
- Exemplo:
  - Testes, me ajudam a ter certeza de que estou implementando uma funcionalidade de forma correta (conforme especificado pelo usuário)
  - E ajudam outros desenvolvedores a terem segurança de que eles não estão inserindo "bugs" em códigos de terceiros
- Contra-exemplo:
  - Documentações completas, são um grande sacrifício para quem escreve; usuário ou beneficiário futuro são incertos.

Melhorias Contínuas

- No livro, chamadas de "improvements"
- Sistemas são construídos "iterativamente", release a release
- Cada release é validada pelos usuários

Flow

- Priorizar um fluxo contínuo e produtivo de atividades (vs fases discretas)
- Favorecer práticas que mantenham desenvolvedores no "fluxo" e produtivos
- Exemplo:
  - Integre o seu código continuamente; não fique muito tempo trabalhando em uma feature sem integrar
  - Integração contínua gera um sentimento "diário" de dever cumprido
  - Integração não-contínua, por sua vez, gera ansiedade (será que estou trabalhando certo?) e stress (uma lista imensa de conflitos para resolver, às vésperas de um deadline).

Falhas Acontecem

- Suponha que você tenha que implementar uma feature F
- Suponha que existam várias alternativas de implementação .... às vezes, é melhor escolher alguma alternativa e implementar, mesmo que ela falhe; e nova alternativa tenha que ser tentada
- É melhor que ficar semanas, meses, discutindo a melhor alternativa
- Às vezes, inclusive, dois times podem ser escalados para trabalhar em paralelo em duas features; depois se escolhe a melhor implementação

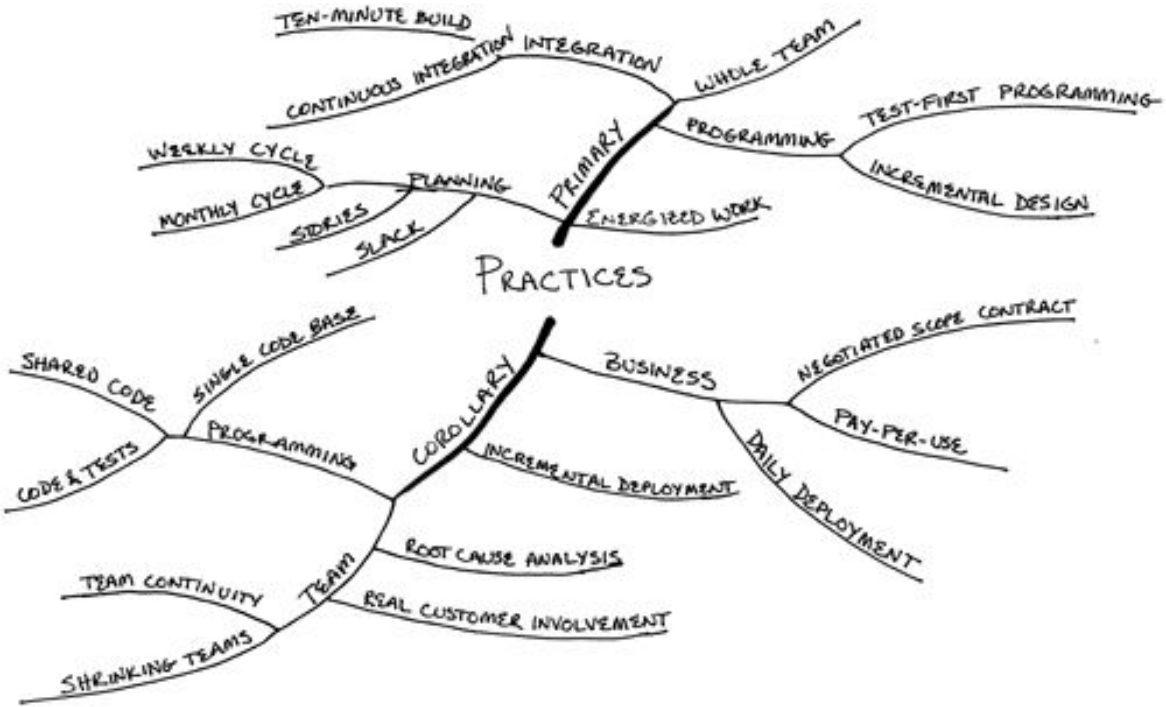
Baby Steps

- Uma feature de cada vez (a de maior prioridade para o cliente); uma sub-feature de cada vez etc.
- Qual a menor tarefa que você pode realizar e que depois pode ser validada como estando na direção certa?

Responsabilidade Pessoal

- No livro, chamada de "accepted responsibility"
- Responsabilidade é algo que se aceita e assume
- Exemplo: quem implementa, depois testa, assume a manutenção e até monitora em produção; logo, não existe "jogo-de-empurra"
- DevOps: engenheiro responsável por implementar um código ("dev") e mantê-lo funcionando, em operação ("ops")

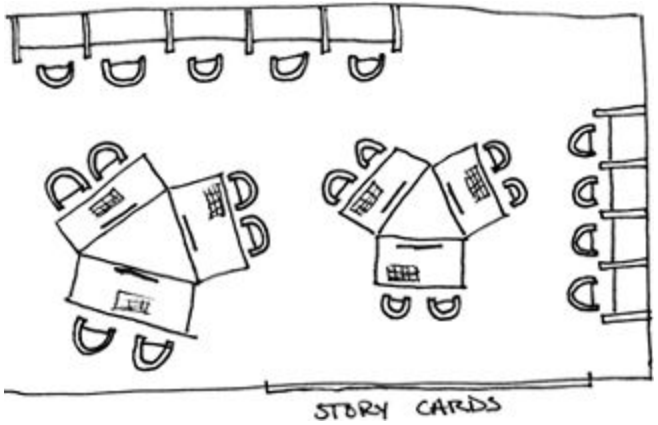
Práticas


TEN-MINUTE BUILD
CONTINUOUS INTEGRATION
INTEGRATION
WHOLE TEAM
PROGRAMMING
TEST-FIRST PROGRAMMING
INCREMENTAL DESIGN
PRIMARY
WEEKLY CYCLE
MONTHLY CYCLE
PLANNING
SLICES
SLACK
ENERGIZED WORK
PRACTICES
SHARED CODE
SINGLE CODE BASE
CODE & TESTS
PROGRAMMING
COROLLARY
BUSINESS
NEGOTIATED SCOPE CONTRACT
PAY-PER-USE
DAILY DEPLOYMENT
INCREMENTAL DEPLOYMENT
TEAM CONTINUITY
TEAM
ROOT CAUSE ANALYSIS
REAL CUSTOMER INVOLVEMENT
SHRINKING TEAMS

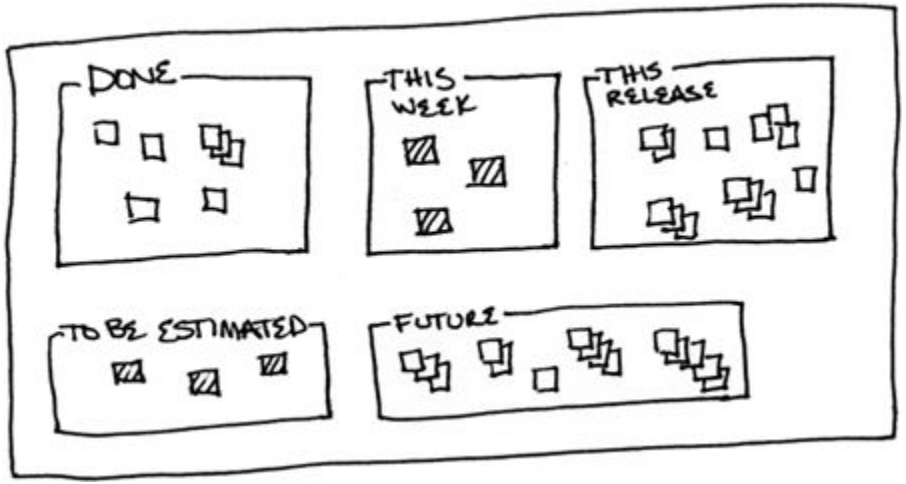
(1) Ambiente de Trabalho

- Favoreça ambientes abertos; que facilitem interações e comunicações
- (Chega-se a afirmar que times distribuídos não devem adotar XP)
- Tamanho dos times: pequenos; 10-12 desenvolvedores; ou número de devs que possam ser alimentados com "duas pizzas"; em Scrum: 7 +/- 2 devs
- Times dedicados a um único projeto/sistema (isto é, evitar "times fracionados"; 2a e 3a, eu trabalho em um projeto X; 4a-5a-6a feira, eu trabalho em um projeto Y).
- Espaço de trabalho deve ser "informativo"; com cartazes nas paredes
- Outra preocupação é com a jornada de trabalhos; recomenda-se algo próximo a 40 horas semanais; sem muitas horas-extras; sem trabalho no fim de semana; pois isso não é sustentável

Ambiente de Trabalho



Ambiente de trabalho



Cartaz que pode ser fixado em uma das paredes (objetivo: permite "visualizar o trabalho" que está em andamento)

(2) Pair Programming

- Talvez uma das práticas mais "polêmicas" de XP
- Ideia: **toda** tarefa de programação (design, codificação, testes) deve ser feita por dois programadores, em conjunto; compartilhando o mesmo computador
- Dois papéis: driver (teclado) e navegador (observador)

Pair Programming

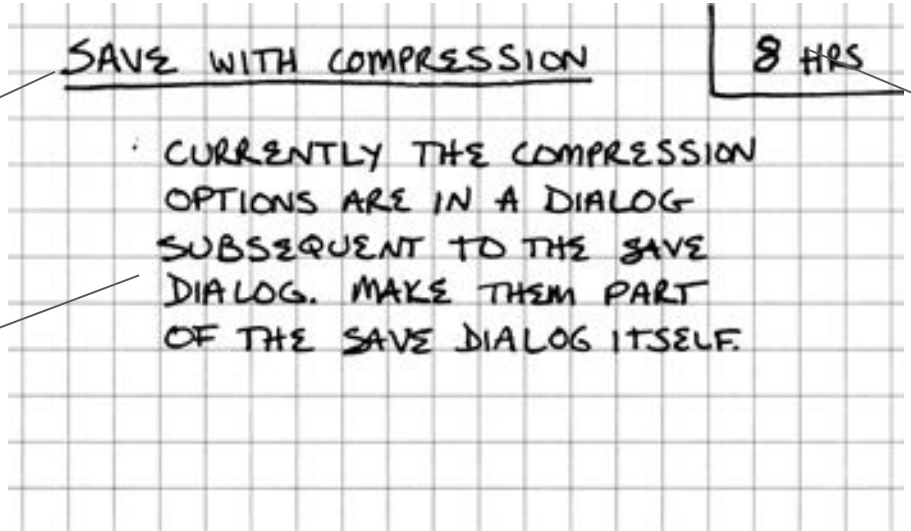
- Vantagens:
  - Incremento de qualidade
  - Transmissão e compartilhamento de conhecimento
- Desvantagem:
  - Custo ("duas pessoas implementando a mesma tarefa")
  - Alguns programadores não se sentem confortáveis.
- Atualmente, não é adotada na intensidade e frequência preconizadas por XP
- Em vez disso, diversas organizações adotam **revisões de código**, isto é, o código de um programador tem que ser obrigatoriamente revisado por outro

(3) Estórias (Stories)

- São unidades de funcionalidade visíveis para o usuário
- São features de um sistema, que tenham sentido para um usuário
- Instrumento usado em XP para especificação de requisitos
- XP evita o nome requisito, pois nem todas os "requisitos" inicialmente planejados são necessários na versão final do sistema; alguns são abandonados ao longo do desenvolvimento
    - Requisitos passa a ideia de algo mandatório.
- Estórias tem um nome curto e uma pequena descrição
- Recomenda-se documentá-las em um cartão de papel; que possa ser fixado em um mural (e permite a "visualização do trabalho")

Estórias

- São estimadas pelos programadores; que definem quanto tempo leva-se para implementar cada estória (porém, não podem ser muito complexas)
- Resumindo: usuário escreve; programadores estimam (accepted responsibility); finalmente, estórias devem ser "testáveis"



(4) Ciclo Semanal (ou Iteração, ou Sprint)

- Planeje seu trabalho semanalmente, de forma que ao fim de toda semana deve-se produzir "deployable software"
- No início da semana, escolha algumas estórias (mais especificamente, peça aos clientes para escolher as estórias da semana).
- Em seguida, os programadores quebram essas estórias em tarefas
- Por fim, eles escolhem algumas tarefas para implementar; e estimam o tempo que vão levar para concluí-las
- Ciclo semanal: chamado de **iteração**, na 1a edição do livro de XP
    - No caso de Scrum, recebe o nome de sprint; embora um sprint costume durar um pouco mais, entre 2-4 semanas.

(5) Ciclo Trimestral (ou Release)

- É importante ter uma visão com duração mais longa do que uma semana
- No início de um ciclo trimestral, define-se o **tema** das estórias do ciclo
    - Logo, em XP, um tema é um conjunto de estórias;
    - E uma estória é um conjunto de tarefas
- Bem como discutem-se possíveis gargalos que estão ocorrendo; discute-se o projeto com fornecedores e clientes externos
- Duração em si não é tão importante; pode ser um pouco mais ou menos
    - Na primeira edição do livro de XP, esses ciclos eram chamados de **releases**; uma release durava de 1 a 6 meses

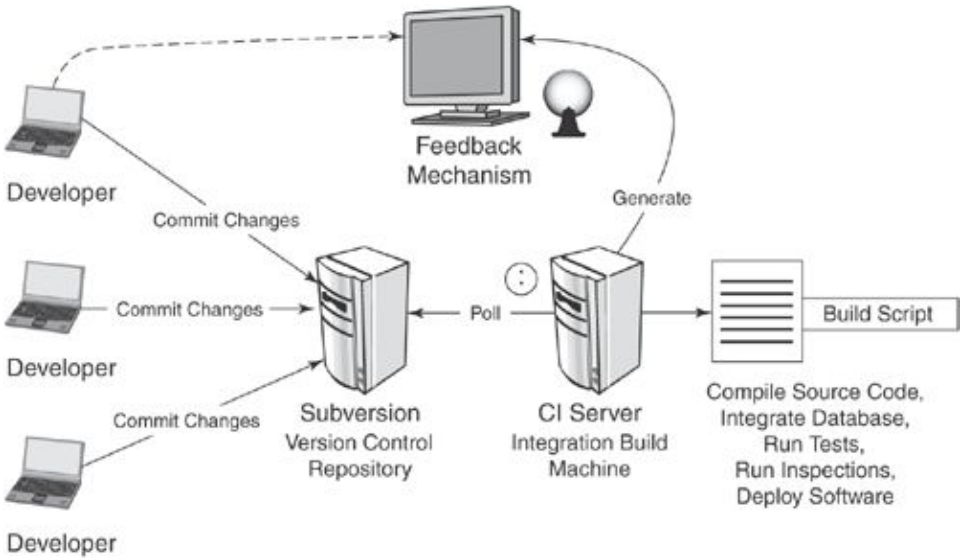
(6) Slack

- Ninguém é 100% produtivo, 100% do tempo
- Logo, introduza nos ciclos algumas "folgas", isto é, tarefas menos importantes, que possam ser descartadas, caso o projeto fique atrasado
    - Refatorações
    - Pesquisa e prospecção de novas tecnologias
    - Alguma documentação ou manual de uso
    - Seminários internos etc

(6) Builds Automatizados ("10-min build")

- Processo de build (compilação) e execução de testes deve ser automatizado
- Às vezes, inclui também a execução de linters e verificadores de estilo
- Build deve demandar o menor tempo possível (10 minutos, é a sugestão)
- Existem diversas ferramentas de build automatizado: Travis, Circle, Jenkins

Builds Automatizados



Source: Continuous Integration: Improving Software Quality and Reducing Risk, 2007

(8) Integração Contínua

- Antes, vamos explicar o que é "integração de código"
- O código fonte de um sistema é sempre armazenado em um serviço de controle de versões, chamado aqui de VCS, que executa em um servidor (git é um exemplo de VCS, mas existem outros, como Mercurial, SVN etc)
- Vantagem de um VCS: pode-se recuperar versões antigas do código
- Suponha então que você tenha que implementar uma feature F:
    - Primeiro, você faz check-out do código mais recente do sistema armazenado no VCS (isto é, executa um pull)
    - Realiza a alteração localmente
    - Submete o código de volta para o servidor (push), isto é, integra o código com a feature F de volta no código "master" do VCS

Integração Contínua

- Tudo seria muito bom, se você fosse o único desenvolvedor deste sistema
- Trabalhando sozinho, nunca haveria conflitos de integração
- O que é um conflito de integração?
  - Ao implementar F, você alterou um arquivo X
  - Ao mesmo tempo, outro desenvolvedor alterou as mesmas linhas de X e integrou o código dele! Antes de você integrar ….
  - O que ocorre quando você for integrar o seu código? Um conflito
- Como resolver um conflito? Não existe bala-de-prata, a solução tem que ser manual, conversada com o desenvolvedor que alterou X, enquanto você implementava F.

Integração Contínua (finalmente, a definição)

- Prática na qual membros do time integram seus trabalhos frequentemente:
  - Normalmente, cada desenvolvedor integra seu código pelo menos uma vez por dia; o que leva a várias integrações por dia.
  - Além disso, cada integração é verificada por um processo de build automatizado, que inclui testes, com o objetivo de detectar erros de integração tão rapidamente quanto possível.
  - Source: https://martinfowler.com/articles/continuousIntegration.html
- Objetivos: (1) feedback rápido; (2) evitar "integration-hell" (isto é, quando a integração é mais custosa que o desenvolvimento)
- Normalmente, implica em pequenas mudanças de código (em cada commit)

(9) Test-Driven Development (TDD)

- Talvez, a principal contribuição de XP!
- Proposta: não só ter muitos testes, mas escrever esses testes antes da fase de codificação (e claro, são sempre testes automatizados)
- Kent Beck é o autor das primeiras versões dos frameworks xUnit (para testes de unidades; em Smalltalk, Java etc); largamente usado hoje em dia.
- Testes são usados para explicitar e validar requisitos, antes da implementação propriamente dita de uma estória ou tarefa; servem como um "contrato" com os usuários
- Veremos mais sobre TDD na 3a parte deste curso

(10) Design Incremental

- Ideia básica: *no big design up front*
- Em vez disso: design incremental, à medida que o projeto vai evoluindo e vai-se conhecendo melhor suas funcionalidades, restrições etc.
- O melhor momento para se pensar em design é "no momento em que ele se faz necessário" (*in light of experience*)
- A principal questão não é se vamos ou não ter atividades de design, mas quando elas ocorrerão e com que intensidade.
- Objetivo: simplicidade (um dos valores de XP)
- (contra-argumento de métodos waterfall: o custo de uma mudança aumenta à medida que se avança em um projeto; na fase de requisitos, o custo é x, na fase de testes, esse custo aumenta exponencialmente)

Refactoring

- Atividade para melhoria de design (logo, é uma atividade fundamental em design incremental)
- Principal atenção deve ser em eliminar duplicações.
    - Princípio DRY (Don't Repeat Yourself): todo "conhecimento" deve ser representado de forma única e não-ambígua em um sistema
    - Designs sem duplicações tendem a ser mais fáceis de serem alterados
- Iremos nos aprofundar em Refactoring na 4a parte deste curso

Práticas Complementares
(Corollary Practices)

(são práticas mais "desafiadoras"; seu uso deve ser avaliado após o time estar adaptado às práticas principais, vistas anteriormente)

(1) Envolvimento de Clientes Reais

- Sempre integre clientes (pessoas das áreas de negócio, por exemplo) às equipes de desenvolvimento; em Scrum, são chamados de Product Owner
- Objetivo: feedback e comunicação
- O objetivo é garantir que o sistema está atendendo às necessidades dos clientes; e evitar que os problemas só apareçam no final do ciclo semanal ou do ciclo trimestral
- Tente chamar para o time um cliente que:
    - Entenda do domínio do sistema
    - Seja capaz de tomar decisões sobre o que é preciso agora; e o que pode esperar (isto é, possa priorizar a implementação de estórias)

(2) Continuous Deployment

- Estou aqui fundindo duas práticas: deployment incremental e deployment diário; e usando um nome mais moderno (continuous deployment)
- Continuous Deployment (CD) é uma evolução de Continuous Integration (CI)
- CD: logo após integrado, código é também liberado para uso, sempre automaticamente (veja que cada vez mais software é distribuído pela Web, sem necessidade de atualização nos clientes)

Source: https://travis-ci.org/

Exemplo: Release Engineering no Google

- Na maioria dos times, o trabalho de liberar novas releases (release engineering) é realizado pelos próprios Engenheiros de Software
- Novas releases são geradas com grande frequência
    - Na maioria dos sistemas, semanalmente ou a cada 15 dias; alguns times chegam a liberar novas releases diariamente
- Por que isso é possível? Porque todas as tarefas são automatizadas
- Benefícios: ajuda a manter os engenheiros motivados; é difícil se manter motivado se você só vai ver o resultado do seu trabalho após meses ou anos; velocidade do desenvolvimento aumenta, pois se tem mais iterações, logo mais feedback e mais chances de reagir a esse feedback.

Source: Software Engineering at Google, Fergus Henderson, https://arxiv.org/abs/1702.01715

Breve apresentação sobre Continuous Delivery

- Por Martin Fowler (ThoughtWorks XCONF 2014)

- Exercício: Qual a diferença entre Continuous Deployment e Continuous Delivery? Quais os benefícios de Continuous Delivery?

(3) Collective Code Ownership (Shared Code)

- Qualquer um do time pode realizar mudanças em qualquer parte do código
- Se você conclui que um certo código pode ser melhorado, refatorado ou possui um bug, vá em frente e resolva o problema.
- Veja como funciona no Google (segundo mesma ref do slide anterior):
  - Todo diretório de mais alto nível possui um arquivo chamado "owners"
  - Normalmente, todo o time (de um projeto) é incluído neste arquivo
  - Owners tem permissão de escrita direta nos arquivos deste diretório
  - Outros desenvolvedores também podem realizar mudanças neste diretório, porém elas devem ser revisadas e aprovadas por um dos "owners"

(3) Contratos com Escopo Negociado

- Contratos para desenvolvimento de software devem definir prazos, custos e qualidade
- Porém, eles não devem ter um "escopo rígido" (lembre-se: requisitos mudam ao longo do projeto; nem sempre, clientes conseguem ter uma visão clara do sistema que querem; nem sempre o que é prioridade no início, continua sendo prioridade no final do projeto)
- Por exemplo, tente uma sequência de contratos mais curtos, em vez de um longo contrato.
- De novo, o objetivo é priorizar feedback e comunicação; em vez de desenvolver algo "não-efetivo" apenas porque está no contrato.

(4) Pay-per-Use

- Esta prática advoga que clientes devem pagar pelo uso do sistema
  - Por exemplo, por compra realizada em um sistema de comércio eletrônico; por mensagem enviada em um sistema de messaging etc
- Outra possibilidade: clientes compram uma assinatura do sistema, pagando um valor mensal, por exemplo
- Por outro lado, pay-per-release deve ser evitado, pois opõe clientes e fornecedor do software
  - Fornecedor: quer sempre gerar novas releases, para lucrar mais
  - Clientes: tendem a postergar a migração para novas releases

Membros de Times XP

Membros de Times XP

- Programadores:
  - Estimam estórias, dividem estórias em tarefas, escrevem testes
  - Escrevem scripts para automatizar rotinas do desenvolvimento
  - Aperfeiçoam o design do software, de forma incremental
- Arquitetos (sim, podem existir arquitetos em times XP):
  - Mas "projeto arquitetural" não é uma tarefa "up-front" e "once-and-for-all"
  - Em vez de dividir-e-conquistar, a ideia é conquistar-e-dividir
  - Em outras palavras, a arquitetura também é definida de forma incremental; o sistema vai sendo particionado, à medida que evolui

Membros de Times XP

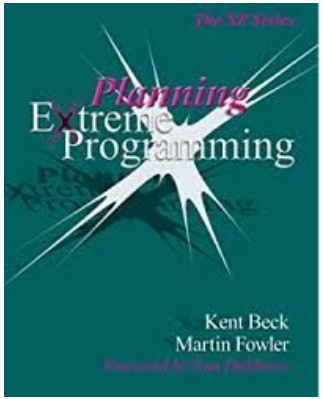
- Gerentes de Projeto:
  - Têm a função de manter os canais de comunicação funcionando, entre os membros do projeto e com os demais stakeholders
  - Acompanhar se o projeto está sincronizado com a realidade
- Gerentes de Produto:
  - Escrevem estórias
  - Em seguida, priorizam estórias, isto é, escolhem temas e estórias para o ciclo trimestral; escolhem estórias para o ciclo semanal
  - Esclarecem dúvidas dos programadores, durante um ciclo
  - (portanto, semelhante ao Product Owner, em Scrum)

Membros de Times XP

- Executivos:
  - Asseguram que o projeto está alinhado com os objetivos da organização
  - Podem usar duas métricas para acompanhar a "saúde" de projetos XP
  - Métrica #1: Número de bugs em produção
  - Métrica #2: intervalo de tempo entre o início do desenvolvimento e quando o projeto gera os primeiros lucros

Planning Extreme Programming

Kent Beck, Martin Fowler (2000)



(1) Planejamento Inicial

- Este projeto é viável?
  - Quais as suas principais funcionalidades? ("big stories")
  - Qual a sua duração? (uma primeira estimativa)
  - Qual o orçamento que temos? Quantas pessoas vão trabalhar?
  - Quais as restrições? Quais os requisitos não-funcionais?
  - Quais as tecnologias que vamos usar? (LP? BD? Frameworks?)
- Supondo que ele seja viável, antes de começar defina:
  - Duração de uma iteração (em semanas)
  - Duração de uma release (em meses)

(2) Escrita das Estórias

- Usuários se reúnem e escrevem algumas estórias
- Quais estórias? As principais, com as primeiras features que eles pretendem ver implementadas
- Quantas estórias? O suficiente para alguns meses de desenvolvimento; por exemplo: duas releases (logo, não precisam ser todas as estórias)
- Programadores analisam as estórias; tiram dúvidas; verificam se elas são testáveis e se não são muito complexas; eles podem, por exemplo, sugerir a divisão de uma estória; logo, o processo de escrever estórias é iterativo
- Por fim, os programadores estimam o esforço (tempo) para implementar cada estória (mais detalhes no próximo slide)

(3) Estimativa do Tamanho das Estórias

- Evidentemente, não é uma tarefa trivial; depende da experiência do time
  - Recomendação #1: não complique ...
  - Recomendação #2: qual a estória mais parecida que já implementou?
- Não estime sozinho; estimativas devem ser um trabalho em equipe
- Qual unidade de tempo/esforço usar? Não importa, desde que seja a mesma unidade em todas as estórias
  - Ideal weeks: quantas semanas uma pessoa vai levar para implementar essa estória, caso esteja 100% dedicada a essa tarefa
  - Outras unidades: homens-hora, homens-mês, story points etc.

Story Points e Velocidade

- **Story points**: unidade de tamanho de estórias, independente de tempo
- Por exemplo: estórias são estimadas como sendo de tamanho 1, 2, 4, 8
- O objetivo é estimar um "tamanho relativo" das estórias; evidentemente, estórias semelhantes devem possuir o mesmo valor de story points
    - "Ahhh, mais um relatório; eles sempre valem 2 story points"
- **Velocidade**: capacidade de trabalho de um time, em uma iteração
- Exemplo: a velocidade de um time = 26 (significa que ele é capaz de implementar 26 story points em uma iteração)
- Como estimar a velocidade de um time? Baseada na experiência anterior

Yesterday's Weather

- "Princípio" muito citado em livros de XP: um país decidiu criar um sofisticado sistema para previsão do tempo, usando algoritmos modernos de IA.
- Após estourar o orçamento diversas vezes, o sistema ficou pronto!
- 70% das suas previsões se revelaram corretas!
- Então, uma pessoa resolveu fazer um sistema alternativo e trivial: a previsão de amanhã será a mesma do "tempo" de hoje.
- Resultado: 69.5% das previsões deste sistema (chamado Yesterday's Weather) são corretas!!
- **Resumindo:** qual a velocidade do time? A mesma da iteração anterior!

(4) Release Planning

- Após definidas e estimadas, as estórias são organizadas em releases
  - Não se preocupe em planejar mais do que duas releases à frente
- Depois, repita o mesmo processo para definir as iterações
  - Isto é, quais estórias serão implementadas em cada iteração
- Qual o critério para ordenar as estórias? Valor para o negócio
- Quem define o valor para o negócio? O cliente
  - Isto é, programadores não devem interferir neste processo; por exemplo, sugerindo features que só "usuários avançados" dão valor para elas ….
  - Mas, programadores podem observar, "criticar" e refinar o planejamento

Release Planning (cont.)

- Resultado final do release planning é uma tabela, com 4 colunas:
  - Story, Estimativa (tempo/esforço), Release, Iteração
- Restrições:
  - **Alocação deve respeitar a velocidade do time**

(5) Iteration Planning

- No início de uma iteração, a equipe se reúne e divide as estórias em tarefas
  - Tarefas devem durar 2-3 dias
- Todo dia, o time realiza uma rápida "stand-up meeting", para monitorar e "socializar" o andamento do projeto. Cada participante responde 3 perguntas:
  - O que eu fiz ontem?
  - O que eu pretendo fazer hoje?
  - E, se for o caso, informa se está enfrentando algum obstáculo
- Quando uma tarefa é concluída? Após ser implementada e testada
- Quando uma estória é concluída? Após ser demonstrada para o cliente e ele aprovar sua conclusão.

Exemplo: Fórum de Q&A

(1) Planejamento Inicial

- Quais as "big stories"? Quais os principais requisitos não-funcionais?
- Qual linguagem? Qual framework web?
- Qual o tamanho da equipe?
- Quais profissionais precisamos (UI, Back-end, Databases, Deploy etc)?
- Qual a "arquitetura" vamos usar? Onde vamos fazer o deploy?
- Quanto tempo temos? Qual o cronograma?
- Quanto vamos cobrar? Como vai se o contrato? Escopo aberto ou fechado?
- Onde vamos trabalhar? Na própria empresa do cliente?
- Qual a duração de um sprint? Quantos sprints até ter uma primeira release?

(2) Estórias (foram escritas pelo usuário)

1. Perfil de usuário (com nickname, short desc, perguntas feitas e respondidas)
2. Postar pergunta
3. Postar resposta
4. Gamificação (upvotes, downvotes, aceitar etc)
5. Pesquisar posts e tela inicial
6. Tags
7. Comentários
8. Meta-QA (isto, é uma família de fóruns Q&A)
9. Jobs

(3) Estimativas (foram feitas pelo time; neste exemplo: usou-se story points)

1. Perfil de usuário (6)
2. Postar pergunta (6)
3. Postar resposta (4)
4. Gamificação (6)
5. Pesquisa (6)
6. Tags (4)
7. Comentários (4)
8. Meta-QA (8)
9. Jobs (8)

(4) Release Planning (feita pelo usuário ou product owner)

1. Postar pergunta (6) (release 1) (sprint 1) (veja que usuário que priorizou)
2. Postar resposta (4) (release 1) (sprint 1)
3. Perfil de usuário (6) (release 1) (sprint 1)
4. Gamificação (6) (release 1) (sprint 2)
5. Pesquisa (6) (release 1) (sprint 2)
6. Tags (4) (release 1) (sprint 2)

---

7. Comentários (4) (release 2) (como é mais à frente, não vamos dividir em sprints)
8. Meta-QA (8) (release 2)
9. Jobs (8) (release 2)

(4) Tarefas da Estória "Postar Perguntas" (Sprint 1)

1. Projetar interface, layout, CSS templates etc (UI)
2. Instalar banco de dados, criar tabelas
3. Instalar servidor web, testar framework
4. Implementar "data layer", com operações CRUD (dúvida: vai ser mesmo preciso editar perguntas? checar isso com o usuário ou "product owner")
5. Implementar interface
6. Testar

Duração do sprint: xx semanas

Leituras e Vídeos Complementares

- Vídeo: Extreme Programming 20 years later
  - Kent Back
  - https://www.youtube.com/watch?v=cGuTmOUdFbo&feature=youtu.be
- Software Engineering at Google
  - F. Henderson, 2017
  - https://arxiv.org/abs/1702.01715
- Development and Deployment at Facebook
  - D. Feitelson, E. Frachtenberg, K. Beck, 2013
  - https://research.fb.com/publications/development-and-deployment-at-facebook/
- The Winter Getaway That Turned the Software World Upside Down
  - The Atlantic Magazine
  - https://www.theatlantic.com/technology/archive/2017/12/agile-manifesto-a-history/547715/