INFO1003

# *Engenharia de Software*

**Prof. Marcelo Soares Pimenta**
***mpimenta@inf.ufrgs.br***

Slides – arquivo 6



# como aumentar a produtividade no desenvolvimento de software?

- trabalhe *mais rápido*
  - automação, ambientes, ferramentas
  - substitua trabalho humano
- *Evite o trabalho!*
  - REUSE ARTEFATOS do CICLO DE VIDA
  - *evite/reduza o desenvolvimento de artefatos específicos para cada projeto…*
- trabalhe *mais inteligentemente*
  - melhore o(s) processo(s)
  - evite/reduza tarefas de pouco valor



# Métodos Ágeis

Uma Introdução !!

Marcelo S. Pimenta
*mpimenta@inf.ufrgs.br*

*Material adaptado de :(copyleft)*
*Prof. Guilherme Lacerda*

# Manifesto Ágil: Foco nos valores da organização

"Estamos evidenciando melhores maneiras de desenvolver software fazendo-o nós mesmos e ajudando outros a fazê-lo. Através desse trabalho, passamos a valorizar:

- Indivíduos e interações **MAIS QUE** processos e ferramentas
- Software em funcionamento **MAIS QUE** documentação abrangente
- Colaboração com o cliente **MAIS QUE** negociação de contratos
- Responder a mudanças **MAIS QUE** seguir um plano

Ou seja, mesmo que os itens da direita tenham um grande valor, valorizamos ainda mais os itens da esquerda!"

Utah, Fevereiro de 2001
Assinado por 17 gurus do desenvolvimento de software.

Fonte: **AgileAlliance** (http://www.agilealliance.com/)

## *Agile Software Development Manifesto* - a declaration of values

- *"We are uncovering better ways of developing software by doing it and helping others do it.*
- *Through this work we have come to value:*
  - Individuals and interactions over Processes and Tools.
  - Working software over Comprehensive documentation.
  - Customer collaboration over Contract negotiation.
  - Responding to change over Following a plan.
- *That is, while there is value in the items on the right, we value the items on the left more."*

- (Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn,
- Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith,
- Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert Martin,
- Stephen J. Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas )



## The Agile attitude focuses on:

- 1. Talent & Skill(fewer better people)
- 2. Proximity (developers - developers - users)
- 3. Communication (morale, daily standup)
- 4. Just-in-time requirements and design
- 5. Frequent Delivery (incremental development)
- 6. Reflection
- 7. Less paper, more tacit / verbal communication
- 8. Tools
- 9. Quality in work
- 10. Different strategies for different projects



## Princípios para esses valores

- ✓ A maior prioridade é satisfazer o cliente entregando, antes e sempre sistemas com valor agregado.
- ✓ 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.
- ✓ Medir progresso através de software funcionando.
- ✓ Desenvolvimento sustentável.
- ✓ Cuidados com o nível técnico e o "bom" projeto.
- ✓ 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.



## A revolução dos métodos ágeis

| | |
|---|---|
| **AD** (Agile Database Techniques) | - Foco em processos de realização |
| **AM** (Agile Modeling) | - Foco em processos de realização |
| **ASD** (Adaptive Software Development) | - **Foco em processos de gestão** |
| **Crystal** | - **Foco em processos de gestão** |
| **FDD** (Feature Driven Development) | - Foco em processos de realização |
| **DSDM** (Dynamic Systems Development Method) | - Foco em processos de realização |
| **Lean Software Development** | - **Foco em processos de gestão** |
| **Scrum** | - Foco em processos de realização |
| **TDD** (Test-Driven Design) | - Foco em processos de realização |
| **XBreed** | - Foco em processos de realização |
| **XP** (eXtreme Programming) | - Foco em processos de realização |

## Alguns exemplos ...

**Lean Software Development** (Produção Enxuta) Mary e Tom Poppendieck

*Lean não trata do que a equipe faz (práticas), mas sim de como a equipe decide o que fazer e quando fazê-lo! Ou seja, ele não é uma metodologia, mas sim um modo de pensar que pode ser aplicado sobre qualquer metodologia de desenvolvimento de software.*

- **Elimine o desperdício** (tudo que não agregar valor não deve entrar no processo)
- **Minimize o inventário** (documentar externamente o código fonte é desnecessário)
- **Agilize a entrega** (quanto mais rápido e mais consistente, melhor)
- **Desenvolva sob demanda** (postergue decisões próprias ou trabalho não solicitado)
- **Fortaleça seus colaboradores** (qualifique-os para exercerem suas atribuições)
- **Atenda aos requisitos do cliente** (envolva o cliente/usuário no processo de desenvolvimento)
- **Faça certo na primeira vez** (use ao máximo a realimentação do cliente)
- **Desconsidere otimizações locais** (foque nos resultados globais da organização)
- **Mantenha parceria com fornecedores** (não reinvente a roda, reutilize)
- **Crie uma cultura de melhoria contínua** (desenvolvimento iterativo + PDCA)



## Alguns exemplos ...

**SCRUM** Ken Schwaber, Jeff Sutherland e Mike Beedle

*Scrum é uma forma ágil de se gerenciar um projeto de software, confiando no auto-comprometimento e auto-organização ao invés de uso de medidas autoritárias.*

- **Crie um backlog do produto** (requisitos técnicos e não técnicos)
- **Implemente o produto em iterações de 30 dias** (cada iteração se denomina Sprint)
- **Planeje no início de cada iteração** (reunião de planejamento de Sprint)
- **Priorize um backlog para a iteração** (foco total nos objetivos do próximo Sprint)
- **Realize a iteração** (alvo nos objetivos do Sprint)
- **Acompanhe a iteração diariamente** (ciclos diários do Scrum)
- **Incremente o produto** (integre continuamente)
- **Avalie os resultados da iteração** (objetivos do Sprint e experiências anteriores)
- **Avalie o backlog do produto** (revisão dos requisitos técnicos e não técnicos)
- **Planeje no início da nova iteração** (reunião de planejamento de Sprint)
- ...



## Alguns exemplos ...

**eXtreme Programming** Kent Beck, Ward Cunningham e Ron Jeffries

*Xp é uma disciplina de desenvolvimento de software baseada em valores como simplicidade, comunicação, realimentação, realimentação e coragem. Ela funciona agregando a equipe de desenvolvedores à de usuários com práticas simples que provêem uma grande visibilidade do projeto e permitem ajustes para cada situação organizacional específica.*

- **Faça uso de metáforas para entender o problema**
- **Mantenha o cliente sempre presente no projeto**
- **Transforme o planejamento num jogo de prioridades**
- **Projete de forma simplificada**
- **Programe em pares**
- **Adote padrões de programação**
- **Mantenha coletiva a propriedade do código fonte**
- **Refatore diariamente os componentes do sistema**
- **Teste continuamente os componentes do sistema**
- **Integre continuamente os componentes do sistema**
- **Entregue com freqüência produtos utilizáveis**
- **Mantenha sempre uma semana de 40 horas**



# XP: Introdução

- XP é uma disciplina de desenvolvimento de software baseada nos valores de:
  - simplicidade,
  - comunicação,
  - feedback e
  - coragem.
- Processo suposto ser leve, eficiente, de baixo risco e flexível.

## Direitos do Cliente *Por Ron Jeffries*

- Planejamento Geral, definindo o que pode ser realizado, quando e a que custo
- Ver e acompanhar o andamento do projeto e, principalmente, o progresso do SW, passando por testes definidos em conjunto com a equipe
- Mudar de idéia, substituir funcionalidades, sem pagar custos exorbitantes
- Ser informado de mudanças no cronograma, em tempo de escolher e reduzir o escopo
- Poder cancelar o projeto a qualquer tempo e ainda assim ter um sistema funcionando, refletindo o investimento realizado até o momento



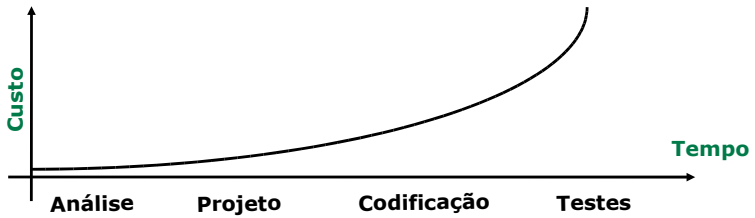
## Direitos do Desenvolvedor *Por Ron Jeffries*

- Saber o que é necessário, com declarações claras de prioridade
- Produzir trabalho de qualidade o tempo todo
- Pedir e receber ajuda da equipe, superiores e clientes
- Fazer e atualizar suas próprias estimativas
- Aceitar as suas responsabilidades, ao invés de tê-las impostas



## Desenvolvimento de SW no passado

- Processos tradicionais
  ***Analisar, projetar e só depois iniciar codificação***
- Prever o futuro
  ***Ter certeza do que se sabe hoje será exatamente o que se quer amanhã***
- Temores
  ***Mudança de requisitos depois de concluída a fase de análise e projeto***






## Desenvolvimento de SW hoje

- Processos modernos incentivam pequenas iterações
  ***Mudanças em etapas posteriores se tornam mais baratas***
- Adotar a mudança
  ***- A Engenharia de SW é diferente das outras engenharias***
  ***- Componentes, Frameworks, BDs podem absorver o alto custo da mudança***

## *eXtreme Programming*

- Disciplina de desenvolvimento de SW, voltada para equipes pequenas e médias, com requisitos vagos ou que mudam freqüentemente
- Criado por *Kent Beck*, *Ron Jeffries* e *Ward Cunningham*
- Principal tarefa é a codificação
- Ênfase
  - menor em processos formais de desenvolvimento
  - maior em disciplina rigorosa
- Aborda
  TDD, participação do cliente como membro da equipe, revisão permanente do código, *refactoring*, integração contínua, refinamento contínuo da arquitetura, planejamento



## Manifesto Ágil

**http://www.agilemanifesto.org**

*"Estamos evidenciando maneiras melhores de desenvolver software fazendo-o nós mesmos e ajudando outros a fazê-lo. Através desse trabalho, passamos a valorizar:*

- Indivíduos e interação **MAIS QUE** processos e ferramentas;
- Software em funcionamento **MAIS QUE** documentação abrangente;
- Colaboração com o cliente **MAIS QUE** negociação de contratos;
- Responder a mudanças **MAIS QUE** seguir um plano.

*Ou seja, mesmo tendo valor os itens à direita, valorizamos mais os itens à esquerda."*

*Kent Beck, Robert C. Martin, Scott Ambler, Alistair Cockburn, Ward Cunningham, Ron Jeffries, Steve Mellor, Mike Beedle, Arie van Bennekum, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Brian Marick, Ken Schwaber, Jeff Shuterland, Dave Thomas*



## Valores do XP

- Comunicação
  ***Práticas valorizam a comunicação, não limitada por procedimentos formais***
- Simplicidade
  ***Incentiva ao extremo práticas que reduzam a complexidade***
- *Feedback*
  ***Práticas garantem um rápido feedback sobre várias partes do projeto***
- Coragem
  ***Práticas aumentam a confiança do desenvolvedor***



## Variáveis de um Projeto

Processos Tradicionais

- Tempo
- Escopo
- Custo

Manipula-se a **Qualidade**

---

*eXtreme Programming*

- Tempo
- Qualidade
- Custo

Manipula-se o **Escopo**

## Ciclo de Vida



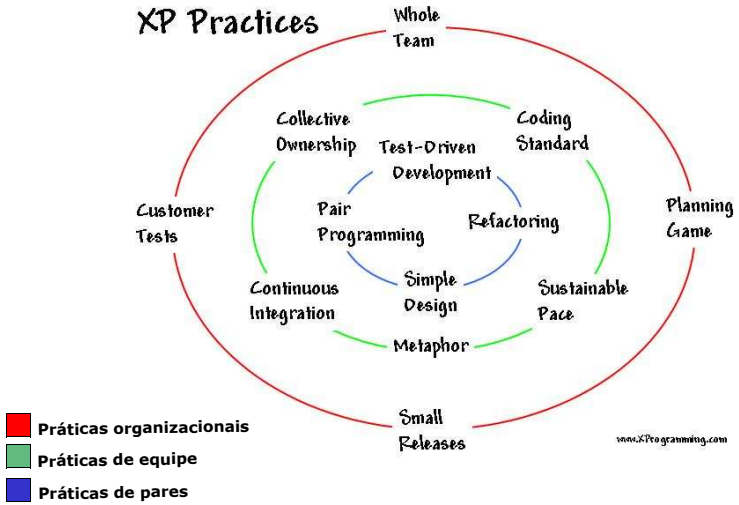

## Atividades de Projeto X Atividades Extremas

- Revisão de código
  ***XP: Programação em pares***
- Testes freqüentes
  ***XP: TDD, automatizados***
- Simplicidade
  ***XP: Recursos não prioritários são descartados***
- Projeto
  ***XP: É realizado a qualquer hora***
- Estimativas
  ***XP: São revistas a cada iteração, apoiada pelas metáforas***
- Versões
  ***XP: Pequenos lançamentos***

**. . .**



## Práticas XP






## *Whole Team*

Todos os colaboradores de um projeto XP formam **um** time.

O programador é o coração de XP
→ ele sabe como programar;

O cliente é a outra metade essencial
→ ele sabe o que programar;

# *Whole Team*

- Existem as figuras opcionais do técnico(coach)
  → notifica quando as pessoas estão se desviando do processo do time e chamam atenção disso;
- e do gerente
  → função é mais voltada a comunicação externa e aquisição de recursos;

- Os papéis não são necessariamente propriedade exclusiva de um indivíduo.



# Equipe (*Whole Team*)

**Equipe XP = Cliente + Time de Desenvolvimento**

- As funções do cliente englobam:
  - ***Definição dos requisitos do projeto***
  - ***Definição de prioridades***
  - ***Controle do rumo do projeto***
  - ***Definir os testes de aceitação do SW***

- Os papéis do time de desenvolvimento englobam:
  - ***desenvolvedores***
  - ***testadores (ajudam o cliente com os testes de aceitação)***
  - ***analistas/projetistas (ajudam o cliente a definir requisitos)***
  - ***gerente (garante os recursos necessários)***
  - ***coach (orienta a equipe, controlando a aplicação do XP)***
  - ***tracker (coleta as métricas do projeto)***



# *Coding Standard*

- Os times em XP seguem uma codificação padrão;

- O código do sistema deve parecer ter sido escrito por apenas uma pessoa;

- A padronização deve dar ênfase a comunicação, sendo adotada voluntariamente pelo time todo.



# Padrões de Codificação (*Coding Standards*)

- Os padrões de codificação são definidos pela equipe
  - ***Organização de código***
  - ***Nomenclaturas***

- Código com aspecto de escrito por um único desenvolvedor
  - ***Competência***
  - ***Organização***

- Práticas e valores favorecidos
  - ***Posse coletiva***
  - ***Comunicação***
  - ***Programação em Pares***
  - ***Refactoring***
  - ***Projeto Simples***

# *Metaphor*

- Uma metáfora é uma descrição textual simples que tem como objetivo ajudar todos os participantes a entender os elementos básicos do sistema e suas relações.

*"O programa funciona como uma colméia de abelhas, onde cada uma sai para adquirir pólen, trazendo-o de volta para a colméia".*

- Apesar de toda a poesia, existe um sistema de substantivos para garantir o entendimento do time.



# Metáfora (*Metaphor*)

- Equipes XP mantém uma visão compartilhada do sistema
  ***Analogia com outros sistemas (natural, computacional, abstrato)***

- Ótima fonte de comunicação entre a equipe, facilitando inclusive as estimativas

- *Design Pattern* de alto nível



# *Planning Game*

O planejamento em XP possui dois aspectos chave no desenvolvimento do sistema:

- o primeiro tenta predizer o tempo utilizado em uma etapa;
- o segundo determina o que fazer na etapa imediatamente posterior.

As considerações comerciais ou técnicas não devem ser primordiais. O desenvolvimento de software é um diálogo em evolução entre o possível e o desejável.

- Área de negócio decide escopo, prioridade e datas de entrega,
- Técnicos decidem estimativas, conseqüências, processo e agenda detalhada.



# *Planning Game*

- *Release Planning:* prática onde o Cliente apresenta as funcionalidades desejadas para os programadores e estes estimam sua dificuldade e definem o escopo da etapa. Ao final de cada etapa é entregue uma versão funcional do sistema ***(Small Releases)***.

- *Iteration Planning:* prática que indica a direção para o time a cada intervalo de tempo definido. Ao final de uma iteração, o Cliente define as novas funcionalidades desejadas para a próxima. Baseado na experiência das iterações anteriores o time pode sinalizar o que e como vão ser realizados seus objetivos na próxima iteração.

## Jogo de Planejamento (*Planning Game*)

- Principais definições
  - ***Estimativas de cada tarefa***
  - ***Prioridades (tarefas + importantes)***
- Próximos passos
  - ***Planejamento de release (cliente propõe as funcionalidades e desenvolvedores avaliam)***
  - ***Planejamento da iteração (define as funcionalidades da iteração e os desenvolvedores estimam)***
- Ótimo *feedback* para o cliente, que dirige o projeto
  - ***Idéia clara do avanço do projeto***
  - ***Clareza: Redução de Riscos, aumentando a chance de sucesso***



## *Small Releases*

- Cada versão deve ser tão pequena quanto possível;
- deve conter todos os requisitos definidos pelo Cliente;
- deve ser coerente como todo o projeto.



## Pequenos Lançamentos (*Small Releases*)

- Disponibiliza a cada iteração SW 100% funcional
  - ***Versão disponibilizada imediatamente***
  - ***Redução de riscos (se o projeto terminar, parte existe e funciona)***
  - ***Detecção prévia de problemas***
  - ***Comunicação entre cliente e desenvolvedor***
- Cada lançamento possui funcionalidades prioritárias para a iteração
- Lançamento pode ser destinado a
  - ***usuário/cliente (testa, avalia e oferece feedback)***
  - ***usuário/final (sempre que possível)***
- *Design* simples e integração contínua são práticas essenciais



## *Collective Code Ownership*

- Qualquer dupla de programadores pode trabalhar em qualquer trecho de código a qualquer hora.
- Quando usuário é único dono de um trecho de código, causa um conjunto de qualidades indesejado: difícil manutenção, duplicação de código e baixa coesão.
- Em XP todos tem responsabilidade total sobre todos os trechos de códigos do sistema.

## Posse Coletiva (*Collective Ownership*)

- O código tem um único dono: a equipe
  - ***Qualquer par de programadores pode melhorar o código***
  - ***Revisão constante do código***
  - ***Aumenta a comunicação***
  - ***Aumenta a qualidade (menos duplicação, maior coesão) e diminui os riscos de dependência de indivíduos***
- Todos compartilham a responsabilidade pelas alterações
- Ideal que se troque os pares periodicamente
  - ***"Todos conhecem tudo"***
  - ***Testes e integração contínua são essenciais e dão segurança***



## *Pair Programming*

*Pair programming* é um estilo de programação onde dois programadores trabalham lado a lado em um computador, colaborando no mesmo projeto, algoritmo, código ou teste.



## *Pair Programming*

- Como todas as metodologias, ela deve ser treinada por algumas vezes para que se aprenda como a usá-la efetivamente;
- Adicionalmente, 90% dos programadores que aprendem preferem *pair programming*.



## *Pair Programming*

- Provê melhores artefatos, e auxilia na comunicação entre os projetistas e no compartilhamento de conhecimento.
- A formação de pares deve ser dinâmica. Uma dupla que trabalhou pela parte da manhã forma novas duplas a tarde.

## Programação em Pares (*Pair Programming*)

- SW é desenvolvido em pares
  - ***- "2 cabeças juntas são melhores que duas cabeças separadas"***
  - ***- 1 piloto e 1 co-piloto***
  - ***- Papéis são alternados freqüentemente***
- Benefícios
  - ***- Melhor qualidade de código (refactoring, testes)***
  - ***- Revisão constante do código***
  - ***- Nivelamento da equipe***
  - ***- Maior comunicação***
- "Um" pelo preço de "Dois"??

  ***Pesquisas demonstram que duplas produzem código de melhor qualidade em aproximadamente o mesmo tempo que programadores que trabalham sozinhos***



## *Simple Design*

- Os times XP constróem *software* com projetos simples. Projetos que começam simples e através de testes (*Customer Tests* e *Test-Driven Development*) e de *Design Improvement* mantêm esse objetivo sempre válido.
- Cada pedaço do projeto deve estar apto a justificar sua existência no todo.
- É uma prática incremental e cotidiana a todos os membros da equipe.



## Projeto Simples (*Simple Design*)

- Projeto está presente em todas as etapas XP

  ***Projeto começa simples e se mantém assim através de testes e refinamento de código (refactoring)***
- Em XP, é levado ao extremo

  ***Não é permitido implementar qualquer funcionalidade adicional que não será usada na iteração atual***
- Implementação ideal é aquela que
  - ***- Passa por todos os testes***
  - ***- Expressa todas as idéias definidas no planejamento***
  - ***- Não contém código duplicado ou que "cheire"***
- Prever o futuro é impossível e é "anti-XP" (Só a Mãe Dináh consegue isso!)



## Testes

- XP tem como um dos pontos de grande impacto o *feedback* de informações;
- Customer test = cliente tem papel fundamental
- *Test-driven development*

# *Customer Tests*

- O Cliente define testes de aceitação quando apresenta o conjunto de funcionalidades desejadas;
- Os times constróem esses testes;
- Os testes são executados de forma automática;



# Testes de Aceitação (*Customer Tests*)

- São elaborados pelo cliente em conjunto com analistas e testadores
  - ***São automatizados***
  - ***Quando rodarem com sucesso, funcionalidade foi implementada***
  - ***Devem ser rodados a cada iteração***
- Ótimo *feedback* para o cliente
  - ***Pode se saber, a qualquer momento, o % de implementação do SW e quanto falta***



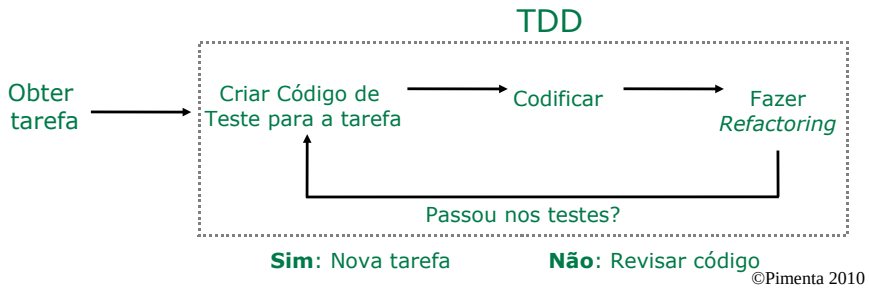
# *Test-Driven Development*

- Desenvolvimento em pequenos ciclos de desenvolvimento, sempre adicionando um teste e validando-o;
- Cada vez que um código é armazenado no repositório cada teste de cada programador deve rodar corretamente;

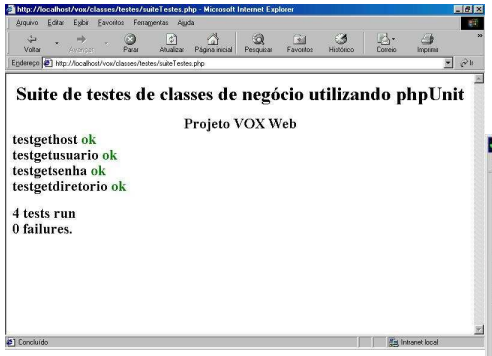


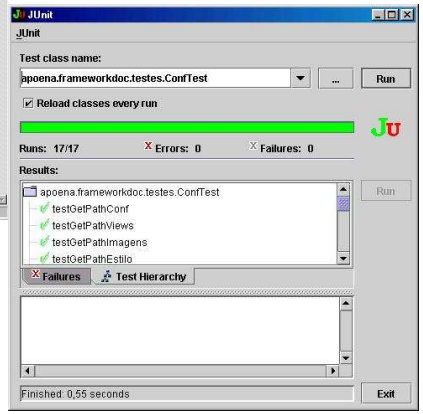
# Desenvolvimento dirigido por Testes (*Test-driven Development*)

- XP valoriza o desenvolvimento dirigido por testes
  - ***São automatizados, executados o tempo todo***
- Testes "puxam" o desenvolvimento
  - ***Cada unidade de código só tem valor se o teste funcionar 100%***
- Testes dão coragem para mudar
  - ***De que adianta a OO isolar a interface da implementação se o desenvolvedor tem medo de mudar a implementação?***



## Desenvolvimento dirigido por Testes (*Test-driven Development*)






## *Design Improvement (Refactoring)*

- *Refactoring* é um processo contínuo de melhoramento de projeto;
- *Refactoring* é focado na remoção de duplicação, um claro sinal de pobreza de código;
- A prática de *Refactoring* é fortemente atrelada pela execução de testes que garantem a evolução acertada do projeto.



## *Continuous Integration*

Em geral, fonte de problemas:

- Integração é uma atividade crítica;
- Integração é delegada para pessoas que não estarão familiarizadas com o código;



## *Continuous Integration*

Como XP ataca o problema de Integração:

- Os times XP mantêm o sistema totalmente integrado todo o tempo → Integração diária;
- Máquina com dedicação exclusiva para integração;
- Integração aceita apenas quando os testes derem 100% resultado.

## Integração Contínua (*Continuous Integration*)

- XP mantém o SW integrado o tempo todo
  - ***Realizada pelo menos uma vez por dia***
  - ***Para integrar, deve-se realizar os testes primeiro***

- "Reduz o tempo passado no inferno da integração" - *Martin Fowler*

- Benefícios
  - ***Expõe o estado atual de desenvolvimento***
  - ***Oferece feedback***
  - ***Estimula agilidade e versões freqüentes do SW***



## Refinamento de Código (*Refactoring*)

- *Design* é melhorado continuamente através de refinamento
  - ***Mudança proposital no código que está funcionando***
  - ***Melhorar sua estrutura interna sem alterar a funcionalidade***
  - ***Visa simplificar o código, remover o código duplicado***

- Principal referência é o catálogo de *refactorings* de *Martin Fowler*

  ***Existência prévia de testes é fundamental para utilização desta prática (elimina o medo dos desenvolvedores de adotar a mudança)***



## *Ritmo Sustentável*

- Os participantes estão em um projeto XP por um prazo longo;
- Fazer hora extra quando é realmente efetivo, em situações onde possam maximizar a sua produtividade;
- Horas extra não são vilões no desenvolvimento de software, mas nunca devem se tornar uma tarefa normal.



## Ritmo Saudável (*Sustainable Pace*)

- XP está na arena para ganhar

- Projetos vampiros não são projetos XP
  - ***Semanas de 80 horas***
  - ***Código ruim, relaxamento da disciplina, stress da equipe***
  - ***Tempo ganho será perdido depois***

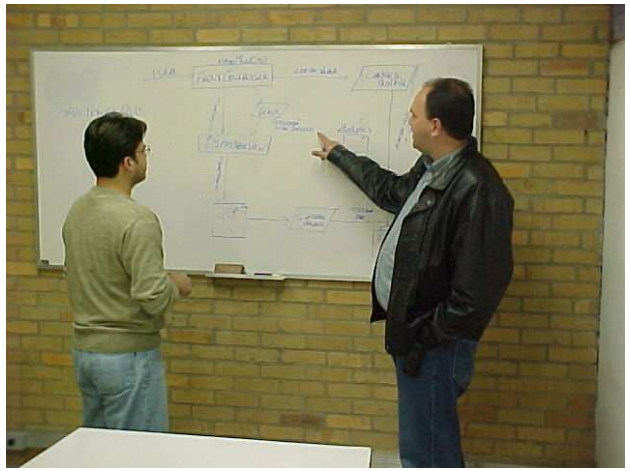
- São aceitáveis eventuais horas extras quando a produtividade é maximizada

## Reuniões em Pé (*Stand-up Meeting*)

- A maioria dos desenvolvedores odeiam reuniões
  - ***Assuntos demorados e desgastantes***
  - ***Impedem os desenvolvedores de codificar***
- As reuniões são valiosas quando
  - ***Não perdem o foco***
  - ***São breves***
- São abordadas tarefas realizadas e a realizar



## Reuniões em Pé (*Stand-up Meeting*)





## *Spikes* de Planejamento (*Spike Solution*)

- São investigações de tecnologias que podem ser utilizadas no projeto
  - ***Auxiliam nas estimativas e na especificação do projeto***
  - ***Podem durar horas ou dias, porém devem ser curtos***
- Englobam
  - ***Avaliações de arquiteturas***
  - ***Algoritmos***
  - ***componentes e frameworks***
  - ***BDs***
  - ***Servidores de aplicação, Web***
  - ***Utilização de artefatos e ferramentas***



## Algumas instituições que usam XP

- HP
- Ford
- Symantec
- Motorola
- Chrysler
- BMW
- Borland
- IBM
- First National Bank
- Thought Works
- CC Pace Systems
- Industrial Logic
- Moore
- Workshare
- Xerox
- Siemens
- Object Mentor
- Objective Solutions
- ImproveIt
- Brasil Telecom
- Embrapa
- Qualiti
- APOENA Software Livre
- Argonavis
- CETIP
- Secretaria da Fazenda SP
- Smart Tech Consulting
- iQualy
- IME-USP
- EverSystems
- PowerLogic Consultoria
- UFRJ
- PUC-Rio

## Pesquisas na *Web*




## Adotando e Escalando XP

- Adote as práticas em doses homeopáticas
  - ***Não seja afobado, seja ninja!***
  - ***Enfatize o problema mais importante***
- Dificuldades Culturais
  - ***Deixar alguém mexer no seu código***
  - ***Pedir ajuda***
  - ***ânsia de tentar prever o futuro (Lembre-se da mãe Dináh...)***
  - ***Escrever testes antes de codificar***
  - ***Refatorar com freqüência (superar o medo)***
- Situações difíceis de aplicar XP

***- Equipes grandes e espalhadas geograficamente (dificulta a comunicação)***
***- Perda do controle sobre o código (código legado que não pode ser alterado)***
***- Feedback demorado***



## Adotando e Escalando XP

- XP e os processos ágeis valorizam as pessoas
  ***Bons desenvolvedores criam bons softwares***
- Processos ágeis são suplementos aos outros métodos
  - ***São atitudes***
  - ***São efetivos***
  - ***Não é um ataque à documentação, e sim a criação de documentos que tem valor***
  - ***Não são para todos***
- O segredo está na sinergia de suas práticas
  ***Menos formalidade, mais diversão***



## Is / Isn't: Misconstruing the message

- 1. Agile SD is cheating
- 2. Agile SD requires the best developers
- 3. Agile SD is hacking
- 4. Agile SD won't work for all projects

# 1. Agile techniques are “cheating”.

- Hire good people;
- Seat them close together to help each other out;
- Get them close to the customers and users;
- Arrange for rapid feedback on decisions;
- Let them find fast ways to document their work;
- Cut out the bureaucracy.

- This is:
- cheating
- stacking the deck
- a good idea
- the heart of agile software development



# 2. Agile only works with the best developers.

- Every project needs at least one experienced and competent lead person. (Critical Success Factor)
- Each experienced and competent person on the team permits the presence of 4-5 “average” or learning people.
- With that skill mix, agile techniques have been shown to work many times.



# 3. Agile is hacking. (Hacker interpretations are available & inevitable.)

- *Hackers: “...spend all their time coding”*
- Agilists: ...test according to project priorities,
- recheck results with users often.

- *Hackers: “...talk to each other when they are stuck”*
- Agilists: ...talk to each other and customers as
- a matter of practice.

- *Hackers: “...avoid planning”*
- Agilists: ...plan regularly

- *Hackers: “...management caves in out of fear”*
- Agilists: ...expect management to provide priorities,
- & participate jointly project adjustments.



# 4. Agile won’t work for all projects.

- *Right. (Business isn’t fair).*

- Agile is an attitude prioritizing:
- Project evaluation based on delivered code
- Rapid feedback
- People as a value center
- Creativity in overcoming obstacles

- Not every team
- ... values the Agile value set.
- ... can set up the needed trust and communication

## Considerações Finais

- XP é uma disciplina de desenvolvimento ágil de SW baseada em comunicação, *feedback*, simplicidade e coragem
- Para usar XP, é preciso fazer com que a equipe se una em torno de práticas simples, obtendo *feedback* e reajustando estas práticas para a situação particular
- XP pode ser implementada aos poucos, porém a maior parte das práticas são essenciais
- Nem todos os projetos são bons candidatos para XP
- XP não é para todo mundo, mas todo mundo pode aprender com ela



## Principais fontes

- http://www.agilemanifesto.org
- http://www.agilemodeling.com
- http://www.extremeprogramming.org
- http://www.xprogramming.com
- http://www.pairprogramming.com
- http://www.martinfowler.com
- http://www.objectmentor.com
- http://www.c2.com
- http://www.xispe.com.br



## Conclusões

- Os valores que XP considera essencial são reflexo do pensamento que o estilo de programação deve servir tanto a necessidade humana quanto a comercial (Agile Alliance manifesto):
  - **Individuals and interactions over processes and tools**
    **Working software over comprehensive documentation**
    **Customer collaboration over contract negotiation**
    **Responding to change over following a plan**
- A incessante comunicação entre os membros da equipe acaba fortalecendo as relações entre si e entre os códigos, gerando respeito entre os participantes.
- Necessita interesse, curiosidade e disposição !!



## Bibliografia


- Programação eXtrema Explicada, K.Beck, Bookman, 2004.
  - *A Bíblia. Deve ser lido com fervor e devoção.*
- Modelagem Ágil, Scott Ambler, Bookman, 2003.
  - *Entre XP e métodos tradicionais...fique com os 2*
- Agile Software Development, Alistair Cockburn, Addison-Wesley, 2002.
  - *Conceitos, metodologias ágeis, discussões...*

# Links

- www.agilealliance.org
- www.extremmeprogramming.org
- www.xp123.com
- www.xprogramming.com

- No Brasil:
  - www.xispe.com.br



# Refatoração

Melhorando a Qualidade de
Código Pré-Existente

Marcelo Pimenta

Material adaptado de :(copyleft)
Prof. Dr. Fabio Kon
Prof. Dr. Alfredo Goldman
Laboratório de Programação eXtrema
Departamento de Ciência da Computação IME / USP



# 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)`**

# Aplicações

1. Melhorar código antigo e/ou feito por outros programadores.
2. Desenvolvimento incremental *à la* XP.

- Em geral, um *passo de refatoração* é tão simples que parece que ele não vai ajudar muito.
- Mas quando se juntam 50 passos, bem escolhidos, em seqüência, o código melhora radicalmente.



# Passos de Refatoração

- Cada passo é trivial.
- Demora alguns segundos ou alguns poucos minutos para ser realizado.
- É uma operação sistemática e óbvia (ovo de Colombo).
- O segredo está em ter um bom vocabulário de *refatorações* e saber aplicá-las criteriosamente e sistematicamente.



# Refatoração Sempre Existiu

- Mas não tinha um nome.
- Estava implícito, *ad hoc*.
- A novidade está em criar um vocabulário comum e em catalogá-las.
- Assim podemos utilizá-las mais sistematicamente.
- Podemos aprender novas técnicas, ensinar uns aos outros.



# Quando Usar Refatoração

- Sempre há duas possibilidades:
  1. Melhorar o código existente.
  2. Jogar fora e começar do 0.
- É sua responsabilidade avaliar a situação e decidir quando é a hora de optar por um ou por outro.

# Origens

- Surgiu na comunidade de Smalltalk nos anos 80/90.
- Desenvolveu-se formalmente na Universidade de Illinois em Urbana-Champaign.
- Grupo do Prof. Ralph Johnson.
  - Tese de PhD de William Opdyke (1992).
  - John Brant e Don Roberts:
    - ***The Refactoring Browser Tool***
- Kent Beck (XP) na indústria.



# Estado Atual

- Hoje em dia é um dos preceitos básicos de Programação eXtrema (XP).
- Mas não está limitado a XP, qualquer um pode (e deve) usar em qualquer contexto.
- Não é limitado a Smalltalk.
- Pode ser usado em qualquer linguagem [orientada a objetos].



# Catálogo de Refatorações

- [Fowler, 2000] contém 72 refatorações.
- Análogo aos padrões de desenho orientado a objetos [Gamma et al. 1995] (GoF).
- Vale a pena gastar algumas horas com [Fowler, 2000].
- (GoF é obrigatório, não tem opção).



# Dica #1

*Quando você tem que adicionar uma funcionalidade a um programa e o código do programa não está estruturado de uma forma que torne a implementação desta funcionalidade conveniente, primeiro refatore de modo a facilitar a implementação da funcionalidade e, só depois, implemente-a.*

# O Primeiro Passo

- Antes de começar a refatoração, verifique se você tem um conjunto sólido de testes para verificar a funcionalidade do código a ser refatorado.
- Refatorações podem adicionar erros.
- Os testes vão ajudá-lo a detectar erros se eles forem criados.

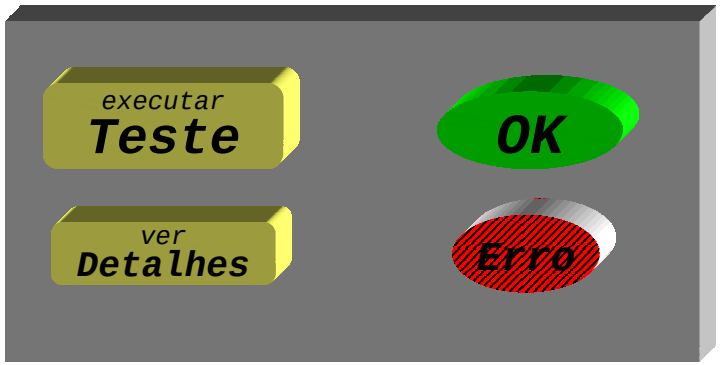


# Testes Automáticos (*self-checking*)

- Os testes devem verificar a si mesmos.
- A saída deve ser
  - "OK" ou
  - lista precisa das coisas que deram errado.
- Quando os testes funcionam, sua saída deve ser apenas uma lista enxuta de "Oks".
- Ou um botão e uma luz verde e outra vermelha.



# O Testador Ideal






# Formato de Cada Entrada no Catálogo

- **Nome** da refatoração.
- **Resumo** da situação na qual ela é necessária e o que ela faz.
- **Motivação** para usá-la (e quando não usá-la).
- **Mecânica**, i.e., descrição passo a passo.
- **Exemplos** para ilustrar o uso.

# *Extract Method (110)*

- **Nome**: *Extract Method*
- **Resumo:** *Você tem um fragmento de código que poderia ser agrupado. Mude o fragmento para um novo método e escolha um nome que explique o que ele faz.*
- **Motivação:** *é uma das refatorações mais comuns. Se um método é longo demais ou difícil de entender e exige muitos comentários, extraia trechos do método e crie novos métodos para eles. Isso vai melhorar as chances de reutilização do código e vai fazer com que os métodos que o chamam fiquem mais fáceis de entender. O código fica parecendo comentário.*



# *Extract Method (110)*

**Mecânica:**

- Crie um novo método e escolha um nome que explicite a sua intenção (o nome deve dizer o que ele faz, não como ele faz).
- Copie o código do método original para o novo.
- Procure por variáveis locais e parâmetros utilizados pelo código extraído.
  - Se variáveis locais forem usados apenas pelo código extraído, passe-as para o novo método.
  - Caso contrário, veja se o seu valor é apenas atualizado pelo código. Neste caso substitua o código por uma atribuição.
  - Se é tanto lido quando atualizado, passe-a como parâmetro.
- Compile e teste.



# *Extract Method (110)*
## Exemplo Sem Variáveis Locais

```
void imprimeDivida () {
   Enumerate e = _pedidos.elementos ();
   double divida = 0.0;
   // imprime cabeçalho
   System.out.println (“***************************”);
   System.out.println (“*** Dívidas do Cliente ****”);
   System.out.println (“***************************”);
   // calcula dívidas
   while (e.temMaisElementos ()){
    Pedido cada = (Pedido) e.proximoElemento ();
    divida += cada.valor ();
   }
   // imprime detalhes
   System.out.println (“nome: ” + _nome);
   System.out.println (“divida total: ” + divida);
}
```



# *Extract Method (110)*
## Exemplo Com Variáveis Locais

```
void imprimeDivida () {
   Enumerate e = _pedidos.elementos ();
   double divida = 0.0;
   imprimeCabecalho ();
   // calcula dívidas
   while (e.temMaisElementos ()){
    Pedido cada = (Pedido) e.proximoElemento ();
    divida += cada.valor ();
   }
   //imprime detalhes
   System.out.println(“nome: ” + _nome);
   System.out.println(“divida total: ” + divida);
}
void imprimeCabecalho () {
   System.out.println (“***************************”);
   System.out.println (“*** Dívidas do Cliente ****”);
   System.out.println (“***************************”);
}
```

# *Extract Method (110)*
## Exemplo **COM** Variáveis Locais

```
void imprimeDivida () {
   Enumerate e = _pedidos.elementos ();
   double divida = 0.0;
   imprimeCabecalho ();
   // calcula dívidas
   while (e.temMaisElementos ()){
    Pedido cada = (Pedido) e.proximoElemento ();
    divida += cada.valor ();
   }
   imprimeDetalhes (divida);
}

void imprimeDetalhes (double divida)
{
   System.out.println("nome: " + _nome);
   System.out.println("divida total: " + divida);
}
```



# *Extract Method (110)*
## com atribuição

```
void imprimeDivida () {
   imprimeCabecalho ();
   double divida = calculaDivida ();
   imprimeDetalhes (divida);
}

double calculaDivida ()
{
   Enumerate e = _pedidos.elementos ();
   double divida = 0.0;
   while (e.temMaisElementos ()){
    Pedido cada = (Pedido) e.proximoElemento ();
    divida += cada.valor ();
   }
   return divida;
}
```



# *Extract Method (110)*
## depois de compilar e testar

```
void imprimeDivida () {
   imprimeCabecalho ();
   double divida = calculaDivida ();
   imprimeDetalhes (divida);
}

double calculaDivida ()
{
   Enumerate e = _pedidos.elementos ();
   double resultado = 0.0;
   while (e.temMaisElementos ()){
    Pedido cada = (Pedido) e.proximoElemento ();
    resultado += cada.valor ();
   }
   return resultado;
}
```



# *Extract Method (110)*
## depois de compilar e testar

- dá para ficar mais curto ainda:

```
void imprimeDivida () {
   imprimeCabecalho ();
   imprimeDetalhes (calculaDivida ());
}
```

- mas não é necessariamente melhor pois é um pouco menos claro.

# *Inline Method (117)*

- **Nome**: *Inline Method*
- **Resumo:** a implementação de um método é tão clara quanto o nome do método. Substitua a chamada ao método pela sua implementação.
- **Motivação:** bom para eliminar indireção desnecessária. Se você tem um grupo de métodos mal organizados, aplique *Inline Method* em todos eles seguido de uns bons *Extract Method* s.



# *Inline Method (117)*

- **Mecânica:**
  - Verifique se o método não é polimórfico ou se as suas subclasses o especializam
  - Ache todas as chamadas e substitua pela implementação
  - Compile e teste
  - Remova a definição do método
  - Dica: se for difícil -> não faça.
- **Exemplo:**

```
int bandeiradaDoTaxi (int hora) {
   return (depoisDas22Horas (hora)) ? 2 : 1);
}
int depoisDas22Horas (int hora) {
   return hora > 22;
}
```

```
int bandeiradaDoTaxi (int hora) {
    return (hora > 22) ? 2 : 1);
}
```



# *Replace Temp with Query (120)*

- **Nome**: *Replace Temp with Query*
- **Resumo:** Uma variável local está sendo usada para guardar o resultado de uma expressão. Troque as referências a esta expressão por um método.
- **Motivação:** Variáveis temporárias encorajam métodos longos (devido ao escopo). O código fica mais limpo e o método pode ser usado em outros locais.



# *Replace Temp with Query (120)*

- **Mecânica:**
  - Encontre variáveis locais que são atribuídas uma única vez
    - Se `temp` é atribuída mais do que uma vez use *Split Temporary Variable* (128)
  - Declare `temp` como `final`
  - Compile (para ter certeza)
  - Extraia a expressão
    - Método privado - efeitos colaterais
  - Compile e teste

# Replace Temp with Query (120)

```
double getPreco() {
    int precoBase = _quantidade * _precoItem;
    double fatorDesconto;
    if (precoBase > 1000) fatorDesconto = 0.95;
    else fatorDesconto = 0.98;
    return precoBase * fatorDesconto;
}

double getPreco() {
    final int precoBase = _quantidade * _precoItem;
    final double fatorDesconto;
    if (precoBase > 1000) fatorDesconto = 0.95;
    else fatorDesconto = 0.98;
    return precoBase * fatorDesconto;
}
```



# Replace Temp with Query (120)

```
double getPreco() {
    final int precoBase = precoBase();          // 1
    final double fatorDesconto;
    if (precoBase > 1000) fatorDesconto = 0.95; //2
    else fatorDesconto = 0.98;
    return precoBase * fatorDesconto;
}

private int precoBase() {
    return _quantidade * _precoItem;
}
```



# Replace Temp with Query (120)

```
double getPreco() {
    final double fatorDesconto;
    if (precoBase() > 1000) fatorDesconto = 0.95; //2
    else fatorDesconto = 0.98;
    return precoBase() * fatorDesconto;
}

private int precoBase() {
    return _quantidade * _precoItem;
}
```



# Replace Temp with Query (120)

```
double getPreco() {
    final double fatorDesconto;
    if (precoBase() > 1000) fatorDesconto = 0.95; //2
    else fatorDesconto = 0.98;
    return precoBase() * fatorDesconto;
}

private int fatorDesconto() {
    if (precoBase() > 1000)
        return 0.95;
    return 0.98;
}

private int precoBase() {
    return _quantidade * _precoItem;
}
```

# *Replace Temp with Query (120)*

```
double getPreco() {
    final double fatorDesconto = fatorDesconto();
    return precoBase() * fatorDesconto;
}

private int fatorDesconto() {
    if (precoBase() > 1000)
        return 0.95;
    return 0.98;
}

private int precoBase() {
    return _quantidade * _precoItem;
}
```



# *Replace Temp with Query (120)* resultado final…

```
double getPreco() {
    return precoBase() * fatorDesconto();
}

private int fatorDesconto() {
    if (precoBase() > 1000)
        return 0.95;
    return 0.98;
}

private int precoBase() {
    return _quantidade * _precoItem;
}
```



# *Replace Inheritance With Delegation (352)*

- **Resumo:** *Quando uma subclasse só usa parte da funcionalidade da superclasse ou não precisa herdar dados: na subclasse, crie um campo para a superclasse, ajuste os métodos apropriados para delegar para a ex-superclasse e remova a herança.*
- **Motivação:** *herança é uma técnica excelente, mas muitas vezes, não é exatamente o que você quer. Às vezes, nós começamos herdando de uma outra classe mas daí descobrimos que precisamos herdar muito pouco da superclasse. Descobrimos que muitas das operações da superclasse não se aplicam à subclasse. Neste caso, delegação é mais apropriado.*



# Não se esqueça

"**Premature optimization is the root of all evil (or at least most of it) in programming**".

Donald Knuth and Tony Hoare

"Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

Rob Pike

# Replace Inheritance With Delegation (352)

- **Mecânica:**
  - Crie um campo na subclasse que se refere a uma instância da superclasse, inicialize-o com `this`
  - Mude cada método na subclasse para que use o campo delegado
  - Compile e teste após mudar cada método
    - Cuidado com as chamadas a `super`
  - Remova a herança e crie um novo objeto da superclasse
  - Para cada método da superclasse utilizado, adicione um método delegado
  - Compile e teste



# Replace Inheritance With Delegation (352)

Exemplo: pilha subclasse de vetor.

```
Class MyStack extends Vector {

   public void push (Object element) {
    insertElementAt (element, 0);
}

   public Object pop () {
    Object result = firstElement ();
    removeElementAt (0);
    return result;
}
```



# Replace Inheritance With Delegation (352)

Crio campo para superclasse.

```
Class MyStack extends Vector {
   private Vector _vector = this;
   public void push (Object element) {
    _vector.insertElementAt (element, 0);
}

   public Object pop () {
    Object result = _vector.firstElement ();
    _vector.removeElementAt (0);
    return result;
}
```



# Replace Inheritance With Delegation (352)

Removo herança.

```
Class MyStack extends Vector {
   private Vector _vector = this; new Vector ();
   public void push (Object element) {
    _vector.insertElementAt (element, 0);
  }

   public Object pop () {
    Object result = _vector.firstElement ();
    _vector.removeElementAt (0);
    return result;
  }
```

# *Replace Inheritance With Delegation (352)*

Crio os métodos de delegação que serão necessários.

```
public int size () {
   return _vector.size ();
  }

public int isEmpty () {
   return _vector.isEmpty ();
  }

}// end of class MyStack
```



# *Collapse Hierarchy (344)*

- **Resumo:** *A superclasse e a subclasse não são muito diferentes. Combine-as em apenas uma classe.*
- **Motivação:** *Depois de muito trabalhar com uma hierarquia de classes, ela pode se tornar muito complexa. Depois de refatorá-la movendo métodos e campos para cima e para baixo, você pode descobrir que uma subclasse não acrescenta nada ao seu desenho. Remova-a.*



# *Collapse Hierarchy (344)*

- **Mecânica:**
  - Escolha que classe será eliminada: a superclasse ou a subclasse
  - Use Pull Up Field (320) and Pull Up Method (322) ou Push Down Method (328) e Push Down Field (329) para mover todo o comportamento e dados da classe a ser eliminada
  - Compile e teste a cada movimento
  - Ajuste as referências a classe que será eliminada
    - isto afeta: declarações, tipos de parâmetros e construtores.
  - Remove a classe vazia
  - Compile e teste



# Replace Conditional With Polymorphism (255)

```
class Viajante {
   double getBebida () {
    switch (_type) {
        case ALEMAO:
            return cerveja;
        case BRASILEIRO:
            return pinga + limao;
        case AMERICANO:
            return coca_cola;
    }
    throw new RunTimeException (“Tipo desconhecido!”);
   }
}
```

# Replace Conditional With Polymorphism (255)

```
class Alemao extends Viajante {
   double getBebida () {
    return cerveja;
   }
}
class Brasileiro extends Viajante {
   double getBebida () {
    return pinga + limao;
   }
}
class Americano extends Viajante {
   double getBebida () {
    return coca_cola;
   }
}
```



# Principio Básico

*Quando o código cheira mal, refatore-o!*

| Cheiro | Refatoração a ser aplicada |
|---|---|
| Código duplicado | *Extract Method (110)*<br>*Substitute Algorithm (139)* |
| Método muito longo | *Extract Method (110)*<br>*Replace Temp With Query (120)*<br>*Introduce Parameter Object (295)* |
| Classe muito grande | *Extract Class (149)*<br>*Extract Subclass (330)*<br>*Extract Interface (341)*<br>*Duplicate Observed Data (189)* |
| Intimidade inapropriada | *Move Method (142)*<br>*Move Field (146)*<br>*Replace Inheritance with Delegation(352)* |



# Principio Básico

*Quando o código cheira mal, refatore-o!*

| Cheiro | Refatoração a ser aplicada |
|---|---|
| **Comentários** (desodorante ☺) | Extract Method (110)<br>Introduce Assertion (267) |
| **Muitos parâmetros** | Replace Parameter with Method (292)<br>Preseve Whole Object (288)<br>Introduce Parameter Object (295) |



# Outros Princípios Básicos

- Refatoração muda o programa em passos pequenos. Se você comete um erro, é fácil consertar.
- Qualquer um pode escrever código que o computador consegue entender. Bons programadores escrevem código que pessoas conseguem entender.
- Três repetições? Está na hora de refatorar.
- Quando você sente que é preciso escrever um comentário para explicar o código melhor, tente refatorar primeiro.

# Mais Princípios Básicos

- Os testes tem que ser automáticos e ser capazes de se auto-verificarem.
- Uma bateria de testes é um exterminador de *bugs* que pode lhe economizar muito tempo.
- Quando você recebe um aviso de *bug*, primeiro escreva um teste que reflita esse *bug.*
- Pense nas situações limítrofes onde as coisas podem dar errado e concentre os seus testes ali.



# Ferramentas para Refatoração

- Refactoring Browser Tool.
  - Dá suporte automatizado para uma série de refatorações.
  - Pode melhorar em muito a produtividade.
  - Existem há vários anos para Smalltalk.
  - Já há vários para C++ e Java.
  - Iniciativas acadêmicas (Ralph@UIUC).
- Agora, integrado no Eclipse e no Visual Works.



# *Refactoring Browser Tool*

- "*It completely changes the way you think about programming*". "Now I use probably half [of the time] refactoring and half entering new code, all at the same speed". Kent Beck.
- A ferramenta torna a refatoração tão simples que nós mudamos a nossa prática de programação.
- **`http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser`**



# Mais Informações

- Livros:
  a) Fowler, M. *Refatorações*, ed. Bookman, 2004.
  Tradução de *:*
  - Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley. 2000.

  b) Joshua Kerievsky, Refatorações para Padrões, Bookman, 2008.
  Tradução de *:*
  - Joshua Kerievsky. *Refactoring to Patterns*, Addison-Wesley. 2005.
- Link:
  www.refactoring.com