



# TÉCNICAS ÁGEIS DE PROGRAMAÇÃO

---

*Michael da Costa Móra – Aula 03*

# Professores

## **DANIEL WILDT**

Professor Convidado

Profissional de tecnologia preocupado com desenvolvimento de produtos e serviços com equipes focadas em aprendizado, melhoria contínua e autonomia. Mentora e produz conteúdo em vídeo, áudio e texto sobre: consciência de tempo, experiência de usuário, empreendedorismo e metodologias ágeis. Sócio e mentor na Wildtech, Blogger/YouTuber no [danielwildt.com](http://danielwildt.com), sócio e diretor na uMov.me.

## **GUILHERME LACERDA**

Professor Convidado

Graduado em Informática pela Universidade da Região da Campanha (2000). Mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (2005). Atualmente, cursa Doutorado em Ciência da Computação na UFRGS, na área de Engenharia de Software. Consultor/Instrutor associado da Wildtech, trabalhando com coaching e mentoring nas áreas de Engenharia de Software, Gerência de Projetos e Produtos e Metodologias Ágeis (eXtreme Programming, SCRUM, Lean). Possui mais de 20 anos de experiência em desenvolvimento de software. Atuou por vários anos como analista/projetista/desenvolvedor de software. Possui as certificações de SCRUM Master (SCM) e SCRUM Professional (CSP) pela SCRUM Alliance. Membro do IASA (International Association of Software Architects). Fundador do Grupo de Usuários de Métodos Ágeis (GUMA), vinculado a SUCESU-RS. É docente de graduação (Ciência da Computação, Análise e Desenvolvimento de Sistemas e Sistemas de Informação, Gestão de TI - Unisinos) e pós-graduação (Engenharia de Software, Desenvolvimento de Aplicações Móveis - Unisinos e Desenvolvimento Full Stack - PUCRS).



## **MICHAEL DA COSTA MÓRA**

Professor PUCRS

Graduado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS), mestre em Computação e doutor em Ciência da Computação pela mesma universidade. Professor-adjunto do Instituto de Informática. Tem experiência na área de ciência da computação com ênfase em inteligência artificial, atuando principalmente nos seguintes temas: inteligência artificial, aprendizagem de máquina, agentes inteligentes e sistemas multiagentes, engenharia de software e desenvolvimento de sistemas, ensino de programação e de ciência da computação.

# *Ementa da disciplina*

Fundamentos da agilidade: primórdios, manifesto ágil, princípios da agilidade. Panorama das metodologias ágeis. Extreme programming: características, valores, práticas, as práticas na prática. Test driven development (TDD): origens, codificar – testar – projetar, benefícios e armadilhas, variações, TDD na prática. Behaviour driven design (BDD): origens e princípios, BDD x TDD, benefícios e armadilhas, BDD na prática.

# EXTREME PROGRAMMING - XP

---

# O surgimento do XP

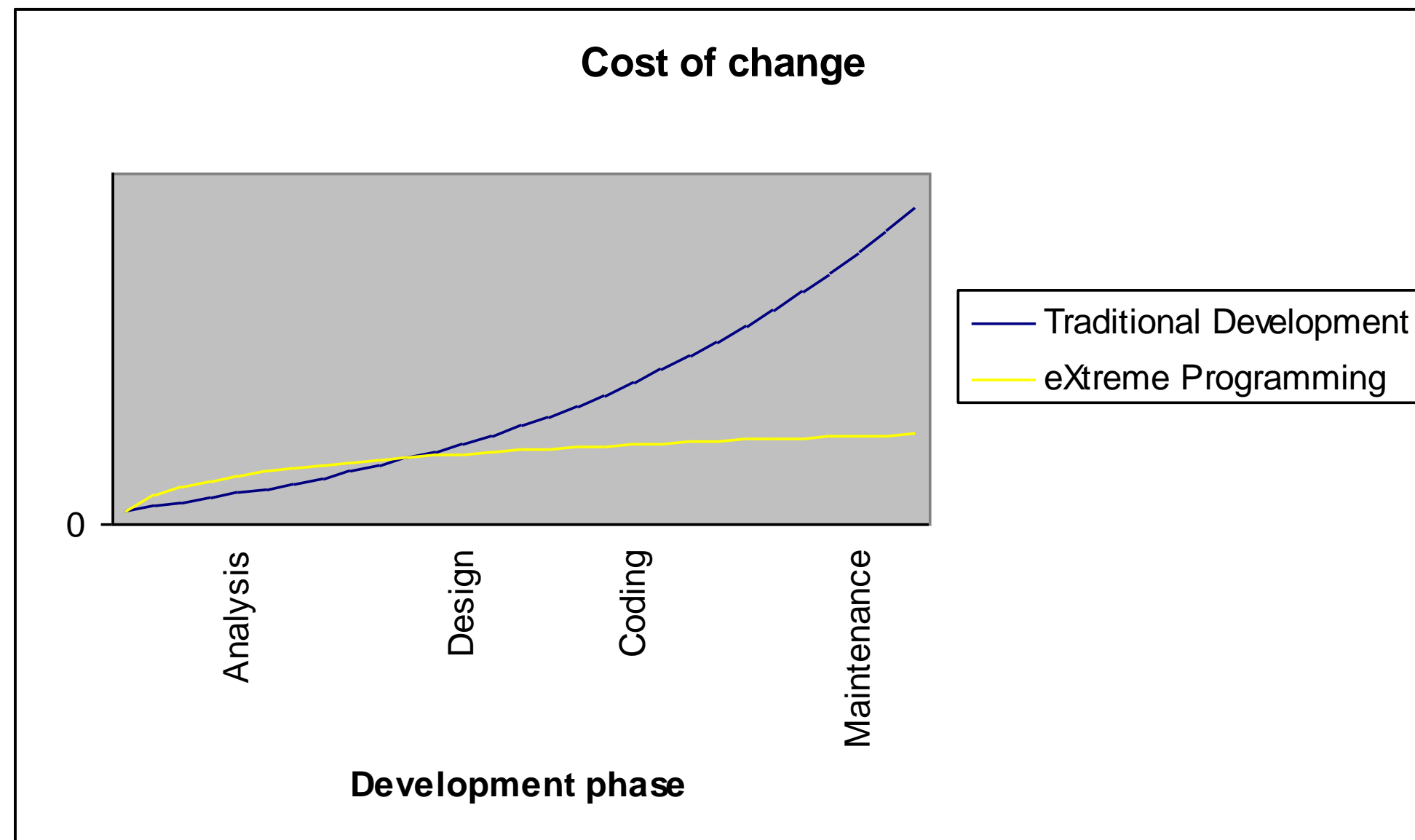
- Em meados de 1990, *Kent Beck* buscou formas simples e eficientes para desenvolver software
- Março/1996 - projeto com novos conceitos que resultaram na metodologia *eXtreme Programming(XP)*



# O que é *eXtreme Programming*?

- Metodologia ágil, leve
- Desenvolvida para:
  - Equipes pequenas e médias (2 a 12 pessoas)
  - Requisitos vagos e em constante evolução
- Baseada nos seguintes valores:
  - Simplicidade
  - Comunicação
  - Coragem
  - Feedback

# XP x Modelo Tradicional





# Boas Práticas

- Planning Game

Objetivo: definir escopo das releases

Resultado: Conjunto de User Stories

- Small Releases

Versões frequentes e pequenas (maior ganho, minimiza riscos)

- Metaphor

Uso de histórias simples e compartilhadas durante o desenvolvimento

- Simple Design

Projetar de forma simples sem redundância, duplicação.

# Boas Práticas

- Test First Design

Testes -> Implementação -  
> Design

- Refactoring

Melhorar o código sem  
alterar a funcionalidade

- Stand-up Meeting

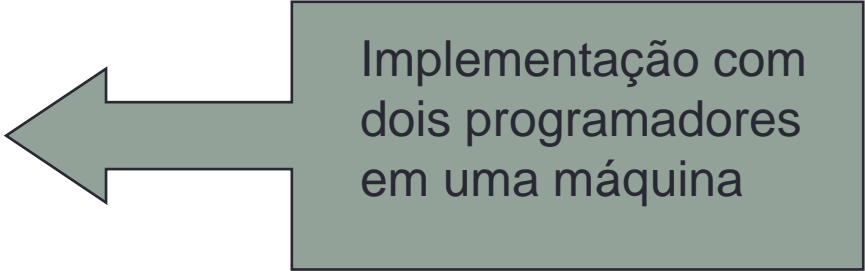
Reuniões rápidas e  
diárias com a equipe

- Continuous Integration

Integrar o sistema várias  
vezes ao dia, a cada tarefa  
finalizada

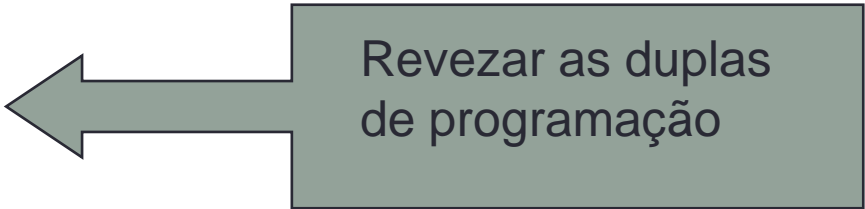
# Boas Práticas

- Pair Programming



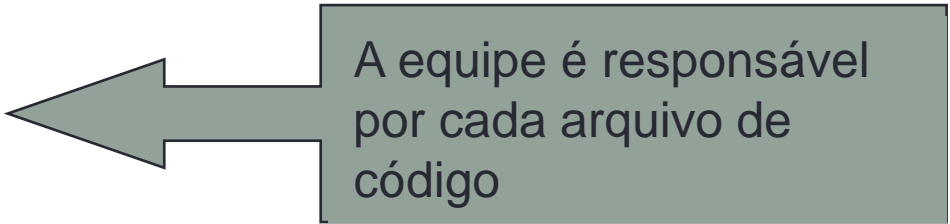
Implementação com dois programadores em uma máquina

- Move People Around



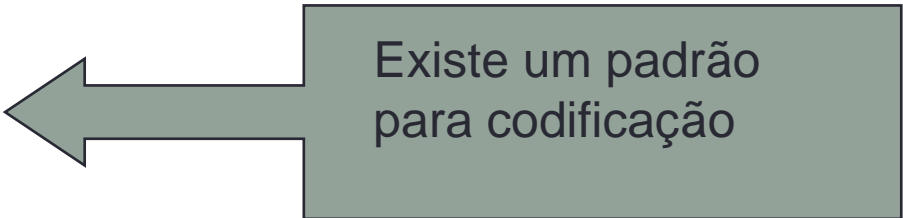
Revezar as duplas de programação

- Collective Code Ownership



A equipe é responsável por cada arquivo de código

- Coding Standards



Existe um padrão para codificação

# Boas Práticas

- 40-Hour Week



Como regra, não trabalhar mais de 40 horas semanais

- On-Site Customer

Ter um papel de cliente na equipe XP em tempo integral para responder as perguntas

- Acceptance Tests

Critérios de aceitação do software definidos pelo usuário

# Papéis no XP

## Big Boss / XpManager

### Deve fazer:

- Agendar reuniões;
- Escrever atas, manter registros;
- Manter o XP Tracker informado dos acontecimentos das reuniões

### Não deve fazer:

- Deixar que problemas externos interfiram no desenvolvimento
- Dizer quando as coisas devem acontecer
- Dizer às pessoas o que fazer
- Cobrar das pessoas

# Papéis no XP

## Cliente

### Xp Gold Owner

É quem paga pelo sistema , geralmente o dono da empresa.

### Xp Goal Donor

#### Deve fazer:

- Escrever User Stories
- Definir prioridades
- Definir testes de aceitação
- Validar testes de aceitação
- Esclarecer dúvidas

#### Não deve fazer:

- Implementar código
- Definir quanto tempo uma tarefa leva para ser feita

# Papéis no XP

## Coordenador

Xp Coach	Xp Tracker
----------	------------

Responsável pela negociação com o cliente quanto ao escopo e pela coordenação do *Planning Game*.

### Deve fazer:

- Coletar métricas
- Manter todos informados do que está acontecendo
- Definir testes de aceitação
- Tomar atitudes diante de problemas
- Sugerir sessões de CRC (Class, Responsibilities, Collaboration)

# Papéis no XP

## Programador (Driver/Partner)

### Deve fazer:

- Estimar prazos de User Stories
- Implementar código de produção
- Trabalhar em par
- Fazer refactoring
- Corrigir bugs

### Não deve fazer:

- Criar/Alterar novas funcionalidades
- Escrever testes de aceitação



# Papéis no XP

Alguns papéis podem ser combinados numa só pessoa

**Xp Manager + Xp Tracker**

Alguns papéis não podem ser combinados numa só pessoa

**Xp Programmer + Xp Tracker**

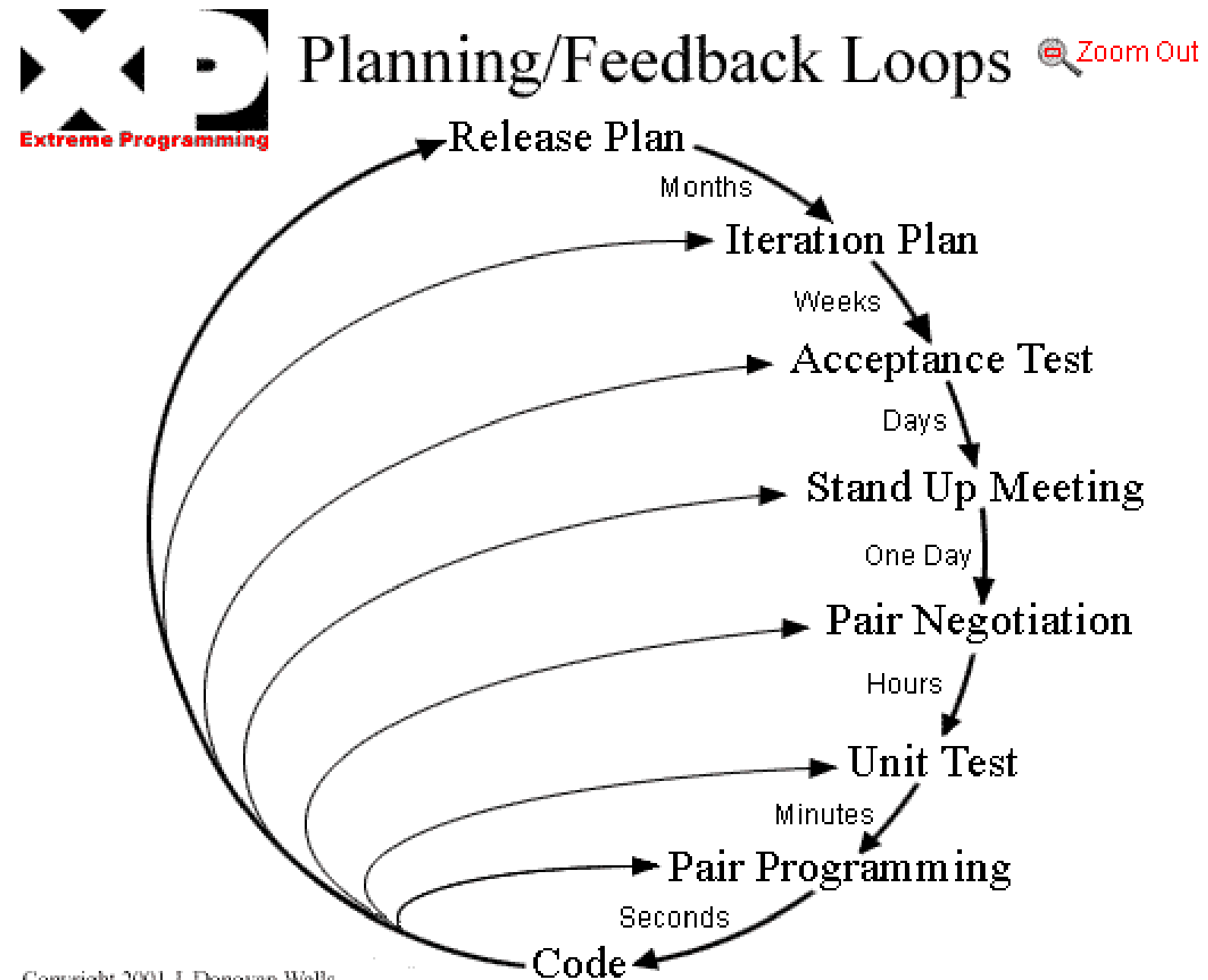
**Xp Customer+ Xp Programmer**

**Xp Coach + Xp Tracker**

# Executando XP

- **Planejar Iteração**
  - Detalhar Histórias(Criar Tarefas)
  - Descrever Prioridades
  - Estimar Tarefas
    - Estimar por Comparação
    - Estimar por Intuição
    - Spike Solutions
  - Comprar/Distribuir Histórias
- **Construir Testes de Aceitação**
- **Programar**
  - Fazer Teste Unitário
  - Implementar
- **Stand-up meetings**
- **Executar Testes de Aceitação**
- **Disponibilizar Iteração**

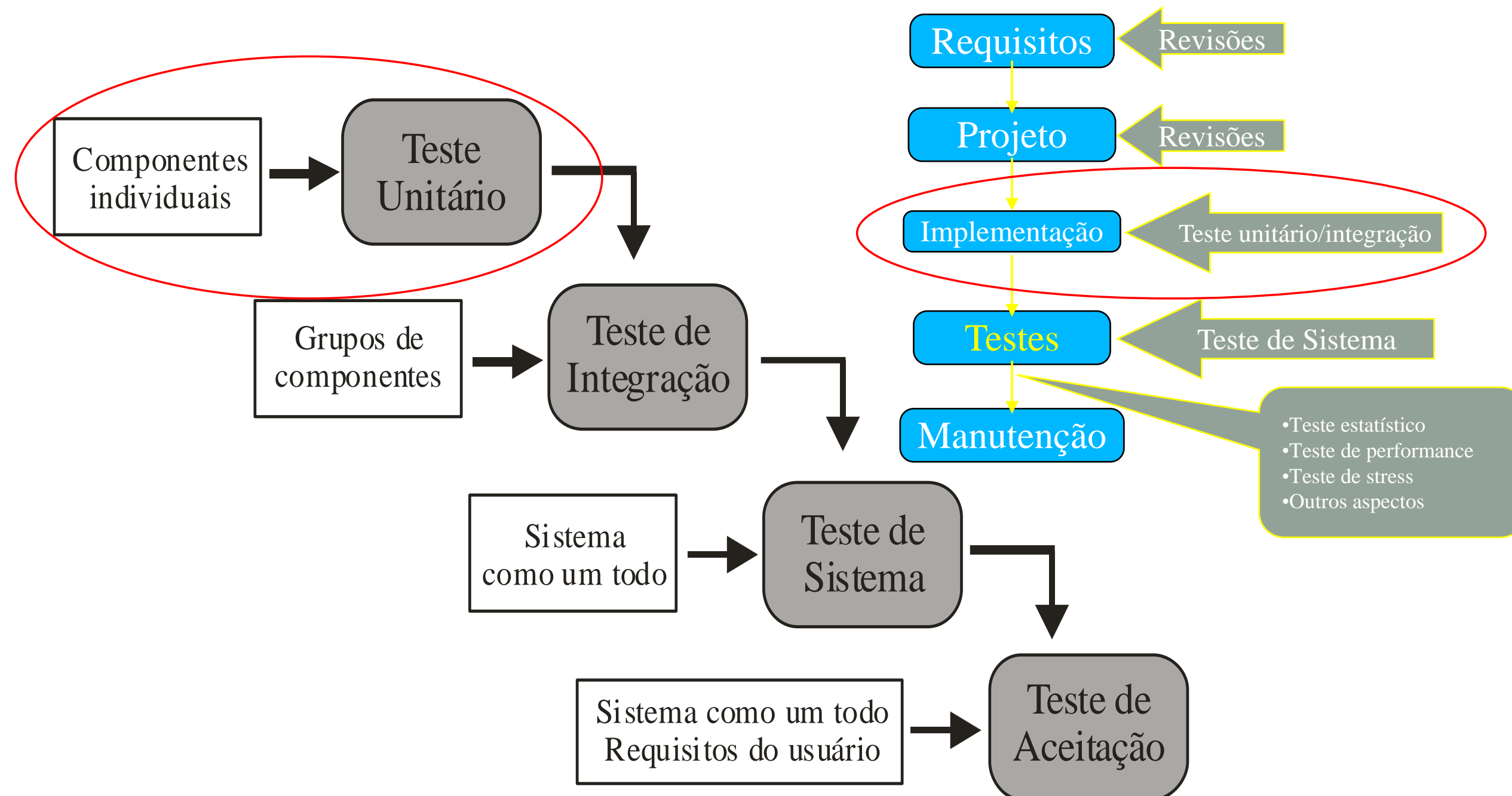
# Executando XP



# TESTES UNITÁRIO

---

# Teste unitário no ciclo de vida de desenvolvimento de software



# Entradas para o teste unitário

- Especificação do módulo antes da sua implementação:
  - Fornece subsídios para o desenvolvimento de casos de teste.
  - Fundamental como oráculo.
- Código fonte do módulo:
  - Desenvolvimento de casos de teste complementares após a implementação do módulo.
  - Não pode ser usado como oráculo.

# Artefatos gerados pelo teste unitário

- Classes “drivers”
  - São as classes que contêm os casos de teste.
  - Procuram exercitar os métodos da classe “alvo” buscando detectar falhas.
  - Normalmente: uma classe “driver” para cada classe do sistema.
- Dublês (“mockups”)
  - Simulam o comportamento de classes necessárias ao funcionamento da classe “alvo” e que ainda não foram desenvolvidas.
  - Quando a classe correspondente ao “dublê” estiver pronta, será necessário re-executar o “driver” que foi executado usando-se o “dublê”.

# Exemplo de criação de classe “driver”

- A partir do projeto de uma classe pode-se especificar os casos de teste.
- Deve-se criar um conjunto de casos de teste capaz de cobrir as funcionalidades básicas da classe.

ContaCorrente
- numero : int - nome : String - saldo : double
+ ContaCorrente(numero : int, nome : String) + depositar(valor : double) + sacar(valor : double) + getNome() : String + getNumero() : int + getSaldo() : double + toString() : String



# Exemplo de conjunto de casos de teste

Configuração	Conta C1: Nome: Fulano Número: 100 Saldo inicial: R\$ 0,00
Casos de teste	Efetuar um depósito de R\$ 1000,00. Conferir saldo.
	Efetuar depósito negativo. Verificar lançamento de exceção.
	Efetuar uma retirada de R\$ 1000,00. Conferir saldo.
	Efetuar uma retirada de R\$ 6000,00. Verificar lançamento de exceção.
	Efetuar uma retirada negativa. Verificar lançamento de exceção.

# Exemplo de classe driver

```
public class ContaCorrenteTest {  
  
    public void testaDeposito() {  
        ContaCorrente target = new ContaCorrente(100, "Fulano");  
        target.depositar(1000.0);  
        if (target.getSaldo() == 1000.0)  
            System.out.println("TestaDeposito: Pass");  
        else System.out.println("TestaDeposito: Fail");  
    }  
  
    public void testaRetirada() {  
        ContaCorrente target = new ContaCorrente(100, "Fulano");  
        target.depositar(6000.0);  
        target.retirar(1000.0);  
        if (target.getSaldo() == 5000.0)  
            System.out.println("TestaRetirada: Pass");  
        else System.out.println("TestaRetirada: Fail");  
    }  
  
    public void testaDepositoNegativo {  
        ...  
    }  
}
```

# Vantagens no uso de classes drivers

- Exige que se reflita sobre as funcionalidades da classe e sua implementação antes de seu desenvolvimento.
- Permite a identificação rápida de bugs mais simples.
- Permite garantir que a classe cumpre um conjunto de requisitos mínimos (os garantidos pelos testes).
- Facilita a detecção de efeitos colaterais no caso de manutenção ou refactoring.

# Dificuldades no uso das classes drivers

- Necessidade de construção do “cenário” em cada método.
- Necessidade de construir um programa para execução dos casos de teste.
- Dificuldade em se trabalhar com grandes conjuntos de dados de teste.
- Dificuldade para coletar os resultados.
- Dificuldade para automatizar a execução dos testes.

# Solucionando as dificuldades

- Uso de ferramentas de automação de teste unitário:
  - JUnit
  - NUnit
  - CppUnit
  - TestNG
  - ...

# XUNIT

---

# Teste unitário

- O nível de teste unitário é todo baseado na construção de classes “driver”.
- Para cada classe que se deseja testar deve-se construir uma classe “driver” de teste que exercita a classe original procurando identificar defeitos.
- A execução do teste unitário implica na execução dos métodos de teste que compõem as classes drivers.
- O uso de “dublês” pode ser necessário quando:
  - Uma classe depende de outra que ainda não está disponível
  - Uma classe depende de outra que não é adequada para uso nos testes que se pretende executar.

# O framework XUnit

- Foi criado no contexto do surgimento do eXtreme Programming em 1998;
- Permite a criação de testes unitários:
  - Estruturados
  - Eficientes
  - Automatizados
- Sua concepção adapta-se facilmente aos IDEs de desenvolvimento
- JUnit: versão para Java do framework



# Recomendações

- Projete casos de teste independentes uns dos outros;
- Não teste apenas o “positivo”. Garanta que seu código responde adequadamente em todos os cenários;
- Crie um driver para cada classe;
- Inclua o nome do método em cada teste. Ex:
  - PositiveLoadTest
  - NegativeLoadTest
  - PositiveScalarLoadtest
- Depure os testes quando for o caso. Não se esqueça de que os testes também são código !!

# Limites

- O teste unitário não deve cruzar certos limites !!!
- Um teste não é um teste unitário se:
  - “Conversa” com o banco de dados
  - “Comunica-se” através da rede
  - “Interage” com o sistema de arquivos
  - Não pode ser executado ao mesmo tempo que os demais testes unitários
  - Necessita de ajustes na configuração do ambiente (edição de arquivos de configuração) para poder ser executado.

# Limites

- Testes que não respeitam os limites não são “de todo maus” ...
- Respeitando os limites teremos:
  - Testes que executam rapidamente
  - Testes com alto grau de acoplamento apenas com as classes que testam
  - Testes sem acoplamento com a camada de persistência ou de interface
  - Testes que “resistem” melhor a manutenção do código !!

# Recursos do JUnit

```
import ContaCorrente.*;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
```

```
public class ContaCorrenteTest{
```

```
    @Test
```



Anotações

```
    public void retirarTest(){
```

```
        ContaCorrente target = new ContaCorrente(100, "Fulano");
```

```
        target.depositar(5000.0M);
```

```
        target.retirar(1000);
```

```
        assertEquals(4000.0M, target.getSaldo());
```

```
    }
```



Asserções

```
    @Test
```

```
    public void depositarTest(){
```

```
        ContaCorrente target = new ContaCorrente(100, "Fulano");
```

```
        target.depositar(5000.0M);
```

```
        target.depositar(1000.0M);
```

```
        assertEquals(6000.0, target.getSaldo());
```

```
    }
```

```
}
```

# Anotações

- São rótulos incluídos no código fonte e que servem para indicar o papel do método na classe driver.
- Exemplos:
  - **@BeforeEach**: indica que o método deve ser executado antes de cada teste.
  - **@AfterEach**: indica que o método deve ser executado depois de cada teste
  - **@Test**: indica que é um método de teste

# Assertões

- Assertões são declarações do que acreditamos ser correto. Quando construímos classes drivers, usamos assertões para verificar as condições que desejamos testar.
- O JUnit captura as exceções lançadas pela assertões. Quando uma assertão falha ela lança uma exceção que é capturada pelo JUnit que detecta a falha.
- As falhas detectadas pelo JUnit são então compiladas em um relatório.

# Exemplo: classe Funcionario

```
public class Funcionario {  
    private String nome;  
    private int codigo;  
    private double salarioBase;  
  
    public Funcionario(int codigo, String nome, double salarioBase) {  
        this.codigo = codigo; this.nome = nome; this.salarioBase = salarioBase;  
    }  
  
    public String getNome() { return nome; }  
    public int getCodigo() { return codigo; }  
    public double getSalarioBase(){ return(salarioBase); }  
    public void setSalarioBase(double salB){ this.salarioBase = salB; }  
  
    public double getSalarioLiquido(){  
        double inss = salarioBase * 0.1;  
        double ir = 0.0;  
        if (salarioBase > 2000.0){  
            ir = (salarioBase-2000.0)*0.12;  
        }  
        return(salarioBase - inss - ir );  
    }  
  
    public String toString() { ... }  
}
```

O salário líquido é calculado descontando-se 10% de INSS e mais 12% de IR sobre o que exceder R\$ 2000,00.

# Exemplo: casos de teste p/Funcionario

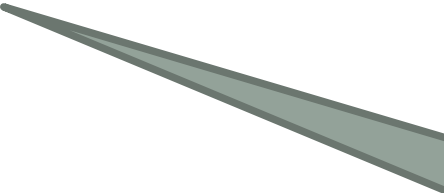
```
public class FuncionarioTest {
    private Funcionario func;

    @Before
    public void setUp() { func = new Funcionario(200, "Ze", 1900.0); }

    @Test
    public void testGetSalarioLiquidoMenosQue2000() {
        double dif = func.getSalarioLiquido() - 1710.0;
        assertTrue(Math.abs(dif) < 1.0);
    }

    @Test
    public void testGetSalarioLiquidoIgual2000() {
        func.setSalarioBase(2000.0);
        double dif = func.getSalarioLiquido() - 1900.0;
        assertTrue(Math.abs(dif) < 1.0);
    }

    @Test
    public void testGetSalarioLiquidoMaior2000() {
        func.setSalarioBase(3000.0);
        double dif = func.getSalarioLiquido() - 2580.0;
        assertTrue(Math.abs(dif) < 1.0);
    }
}
```



Cuidado ao fazer  
comparações com tipos  
ponto flutuante



# JUnit annotations

- `@BeforeAll`: executa no início da bateria de testes (fixture)
- `@AfterAll`: executa quando encerra a bateria de testes
- `@BeforeEach`: executa antes de cada teste
- `@AfterEach`: executa depois de cada teste
- `@Test`: indica um método de teste

# JUnit asserts

- assertEquals: deprecated para tipos ponto flutuante
- assertFalse: sobre uma condição
- assertTrue: sobre uma condição
- assertNull: sobre uma referência
- assertNotNull: sobre uma referência
- assertEquals: se referenciam o mesmo objeto (não equals)
- assertEquals: oposto de assertEquals
- fail(msg): falha o teste e exibe msg

# O processo de teste unitário

- “Processo” de desenvolvimento com teste unitário:
  1. Definir a interface (esqueleto) da classe alvo
  2. Definir o conjunto de casos de teste
  3. Implementar a classe driver
  4. Completar a codificação da classe alvo
  5. Executar os testes
  6. Corrigir os bugs, se houverem
  7. Complementar os testes

# Exemplo: parquimetro

- A classe *Parquimetro* modela um parquimetro
  - Aceita moedas de 1, 5, 10, 25, 50 e 100 centavos (simul. 1 real)
  - Permite emitir tickets de 2 reais se tiver saldo
  - Capaz de informar o saldo restante
- Etapas:
  1. Definir a interface da classe
  2. Definir os casos de teste
  3. Implementar a classe driver
  4. Implementar a classe alvo (*Parquimetro*)
  5. Executar os testes
  6. Corrigir os bugs
  7. Complementar os testes

# E1: definir interface da classe

```
public class Parquimetro{  
    // Permite inserir moedas no parquimetro (soma no saldo)  
    // Valores possíveis: 1, 5, 10, 25, 50 e 100 (1 real)  
    // Gera IllegalArgumentException no caso de valor inválido  
    public void insereMoeda(int valor)  
        throws IllegalArgumentException{}  
  
    // Retorna o saldo acumulado no parquimetro  
    public double getSaldo(){    }  
  
    // Emite um ticket de 2 reais se houver saldo disponível.  
    // Retorna true se a operação foi possível  
    public boolean emiteTicket(){    }  
  
    // Devolve o saldo existente. Retorna o valor devolvido  
    public int devolve(){    }  
}
```

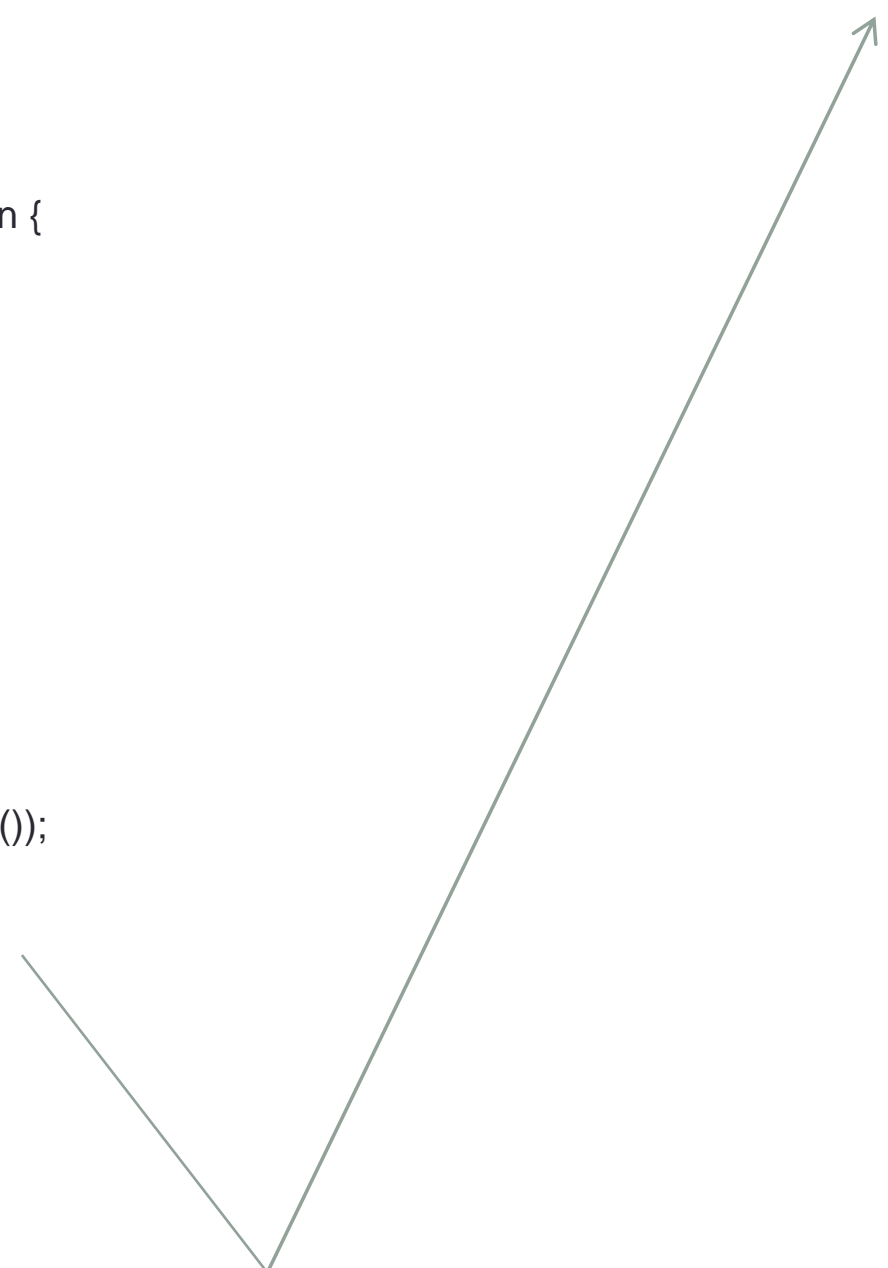
## E2: definir os casos de teste

Configuração	Parquimetro P1: Saldo inicial: 100,00 (100 centavos, 1 real)
Casos de teste	Inserir moedas de 1, 5, 10, 25, 50, 100 Conferir saldo = 2,91
	Inserir moeda de 20. Verificar lançamento de exceção
	Inserir 50, 50, 100 e emitir ticket Deve retornar true
	Emitir ticket Deve retornar false
	Inserir 50, 25, 25, 50, 25. Emitir ticket. Retornar saldo. Deve retornar 25

# E3: implementar o driver

```
public class ParquimetroTest {  
  
    private Parquimetro parq;  
  
    @BeforeEach  
    public void setUp() throws Exception {  
        parq = new Parquimetro();  
        parq.inserirMoeda(100);  
    }  
  
    @Test  
    public void testInserirMoeda() {  
        parq.inserirMoeda(1);  
        parq.inserirMoeda(5);  
        parq.inserirMoeda(10);  
        parq.inserirMoeda(25);  
        parq.inserirMoeda(50);  
        parq.inserirMoeda(100);  
        assertEquals(291, parq.getSaldo());  
    }  
}
```

```
@Test  
public void testGetSaldo() {  
    int actual = parq.getSaldo();  
    assertEquals(100, actual);  
}  
  
@Test  
public void testEmitirTicket() {  
    parq.inserirMoeda(50);  
    parq.inserirMoeda(50);  
    parq.inserirMoeda(100);  
    boolean actual = parq.emitirTicket();  
    assertEquals(true, actual);  
}  
  
@Test  
public void testDevolver() {  
    parq.inserirMoeda(50);  
    parq.inserirMoeda(50);  
    parq.inserirMoeda(100);  
    parq.emitirTicket();  
    int actual = parq.devolver();  
    assertEquals(100, actual);  
}
```



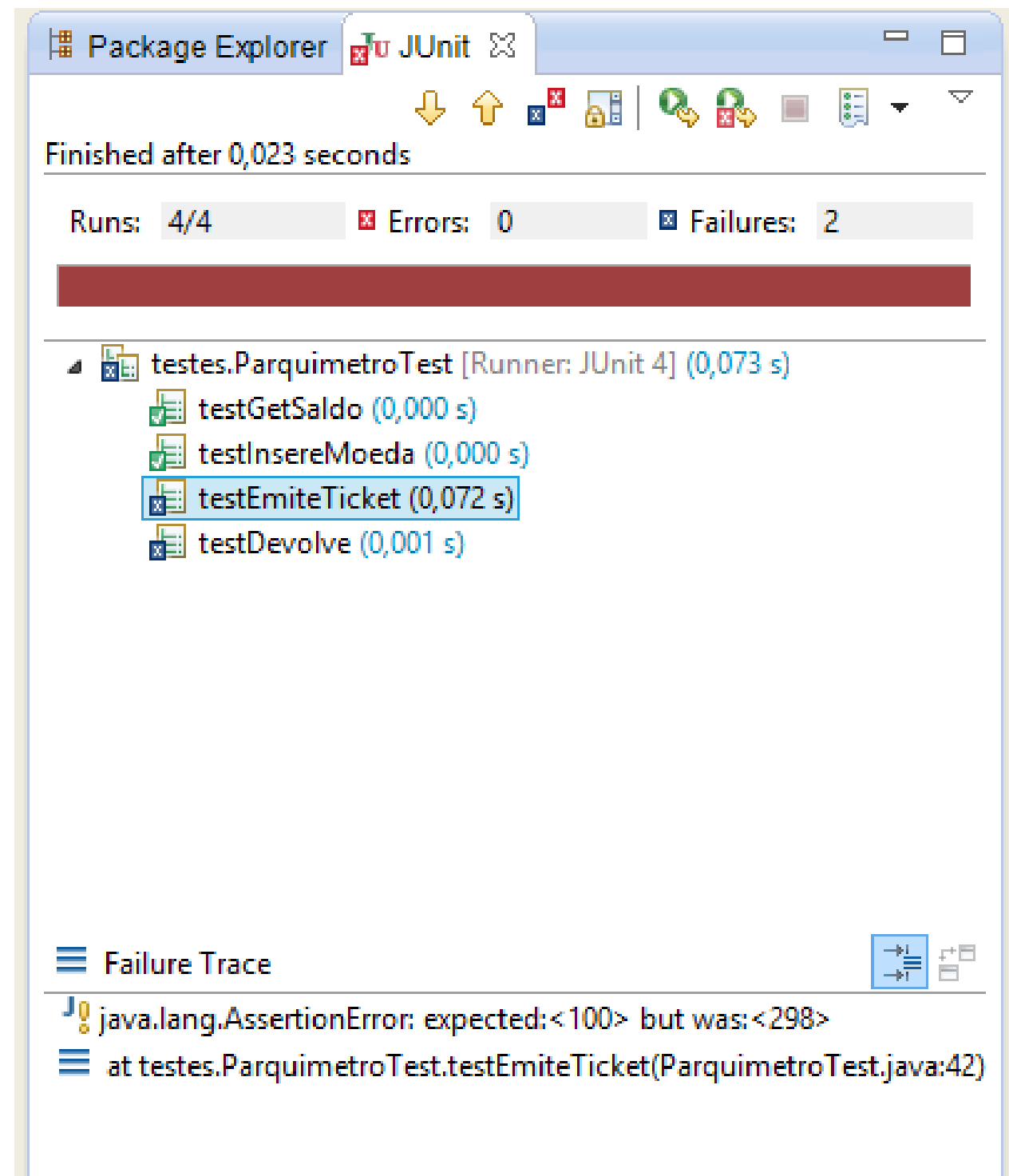
## E4: implementa a classe alvo

```
public class Parquimetro {  
    private int saldo;  
  
    public Parquimetro() {saldo = 0; }  
  
    public void insereMoeda(int valor) {  
        switch (valor) {  
            case 1:  
            case 5:  
            case 10:  
            case 25:  
            case 50:  
            case 100:  
                saldo += valor;  
                break;  
            default:  
                throw new Illegal ... ;  
        }  
    }  
}
```

```
    public int getSaldo(){return  
    (saldo); }  
  
    public boolean emiteTicket() {  
        if (getSaldo() < 2) {  
            return (false);  
        }  
        saldo -= 2;  
        return (true);  
    }  
  
    public int devolve(){  
        int tmp = saldo;  
        saldo = 0;  
        return (tmp);  
    }  
}
```



## E5: executa os testes



## E6: corrigir os bugs (p1)

- Qual o problema no código?
- Veja que o método `emiteTicket` subtrai o valor 2 do saldo.
- Ocorre que o saldo está sendo mantido em centavos e não em reais.
- Então o correto é subtrair 200.
- Veja também que o teste específico criado para o método `emiteTicket` era fraco e não detectou o erro !

## E6: corrigir os bugs (P2)

```
public bool emitTicket() {  
    if (Saldo < 200) return (false);  
    saldo -= 200;  
    return (true);  
}
```

# E7: completar os testes

- Analisar o código fonte e completar os casos de teste se for o caso
  - Cobertura de código
  - Cobertura de condição
  - Cobertura de condição múltipla
  - Cobertura de laços

# OUTRAS POSSIBILIDADES DO JUNIT

---

# Recursos avançados

- O JUnit5 oferece uma série de recursos adicionais para facilitar a implementação dos casos de teste.
- Estes permitem entre outros:
  - Verificação de exceções
  - Verificação do tempo de execução
  - Comparação de coleções

# Verificação de exceções

- Como escrever um teste que é bem sucedido quando uma exceção é gerada:

```
//JUnit5
@Test
public void testDivideEx() {
    assertThrows(ArithmeticException.class, () ->
        calc.divide(0));
}
```

- O teste irá reportar um erro se a exceção não for gerada !!!
- Para testes que devem falhar quando uma exceção é gerada, basta deixar a exceção ocorrer !!!

# Verificação do tempo de execução

- Pode-se verificar se um método não está ultrapassando o tempo de execução previsto para ele:

```
//JUnit 5
@Test
public void testCalculoDemorado() {
    assertTimeout(Duration.ofMillis(500),
        () -> calc.calculoDemorado());
    assertEquals(55, calc.get());
}
```



# Comparação de coleções

- O método *assertEquals* está sobrecarregado para trabalhar com arranjos.

```
void assertEquals(Object[] esp, Object[] real);
```

- Dois arranjos são considerados iguais se o conteúdo de todas as suas posições for igual.

# E os Dublês

- Dublês
  - Mocks aren't Stubs, Martin Fowler (2007)
    - <http://martinfowler.com/articles/mocksArentStubs.html>
- Fowler (na linha proposta por Meszaros):
  - Dummy: dublê não utilizado
  - Fake: protótipos limitados (“banco de dados” em memória)
  - Stubs: respostas estáticas
  - Mocks: utilizam especificações a respeito da ordem de chamadas recebidas
  - Mockistas preferem evitar testes que integram objetos reais
- Estado ou Comportamento

# ESTRATÉGIAS DE GERAÇÃO DE CASOS DE TESTE

---

# Técnicas de Teste Funcional

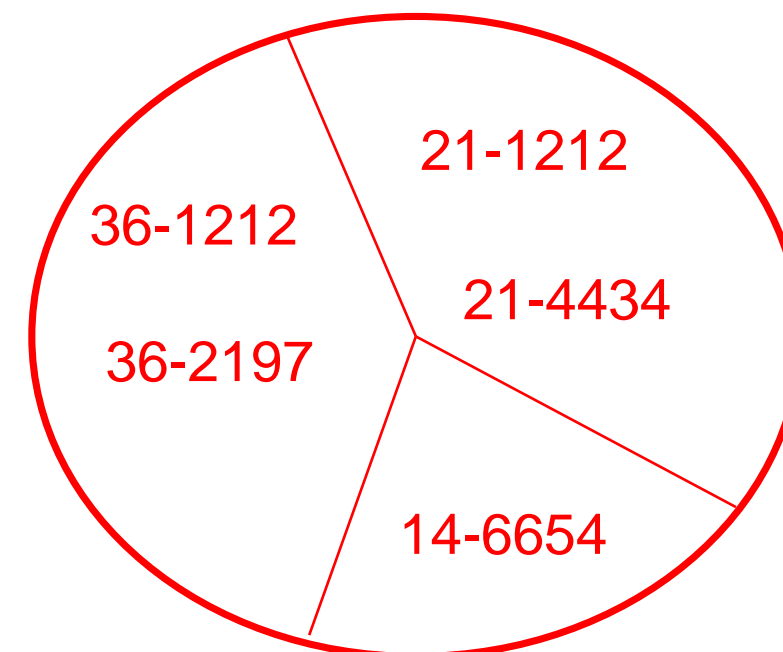
- Usam como entrada a especificação do módulo
- Especificação/Contratos:
  - É fundamental gerar casos de teste com valores válidos buscando verificar se o módulo se comporta como especificado.
  - No caso de programação por contratos deve-se gerar casos de teste que busquem verificar se a implementação atende as especificações do contrato

# Técnicas de Teste Funcional

- Valor Limite
  - Funciona bem quando o programa a ser testado é função de várias variáveis independentes que representam conjuntos que tenham uma relação de ordem.
  - Geração de casos de teste:
    - Fixa-se o valor de todas as variáveis menos uma em seus valores nominais
    - A variável escolhida assume os valores: {MIN,MIN+1,NOMINAL,MAX-1,MAX}
    - Exemplo:
      - $v1:\text{int} \in [10,20]$
      - Casos de teste: {10, 11, 15, 19, 20}
  - Se existir mais de uma variável, deve-se testar as combinações possíveis entre elas
  - Se a variável for uma enumeração deve-se testar todos os valores

# Técnicas de Teste Funcional

- Classes de equivalência
  - Princípio: dividir o domínio de entrada em subconjuntos de maneira que o comportamento de um dos membros do conjunto seja representativo do comportamento de todos os membros do conjunto.
  - Gerar pelo menos um caso de teste para cada classe
  - Ex: Nros de telefone (não tem relação de ordem) onde o prefixo indica a operadora.



# Técnicas de Teste Funcional

- Diagramas de estado
  - Se há um diagrama de estados disponível, gerar casos de teste que garantam cobertura de estados e transições.
- Condições de erro
  - Gerar casos de teste que procurem gerar as condições de erro previstas (ex: geração de exceções)
- Valores inválidos
  - Gerar caso de teste com valores de entrada inválidos

# Técnicas de Teste Estrutural

- Usa como entrada o código fonte.
- Usado para refinar os casos de teste
  - Gerar o grafo de programa
  - Procurar garantir a cobertura do grafo de programa
    - Verificar se passa por todos os comandos
    - Verificar se exercita cada uma das opções de cada condição
    - Verificar se cada laço itera pelo menos k vezes



# REFATORAÇÃO

---

# 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 em ciclos de desenvolvimento anteriores.
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”
- Ter um bom vocabulário de *refatorações* e saber aplicá-las criteriosamente e sistematicamente.

# Refatoração Sempre Existiu

- Mas não tinha um nome.
- Fazia-se de forma *ad hoc*.
- A novidade está em criar um vocabulário comum e em catalogá-las.
- Viabilizada / estimulada pela prática de testes unitários

# Quando Usar Refatoração

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

# Origens

- Surgiu (nesta forma) 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 do 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
  - Catálogo: 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].

## 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 em Qualquer Refatoração

- 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.

# 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 explicita 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.*



## *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;
    }
}
```

# Introduce Null Object (260)

```
Result meuORBCorba (String parametros[])
{
    Result r;
    if (pre_interceptor != NULL)
        pre_interceptor.chamada ();
    if (meuObjeto != NULL && meuObjeto.metodo() != NULL)
        r = meuObjeto.metodo().invoke (parametros);
    if (pos_interceptor != NULL)
        r = pos_interceptor.chamada (r);
    return r;
}
```

# Introduce Null Object (260)

- Substitua o valor NULL por um objeto do tipo Nulo.

```
Result meuORBCorba (String parametros[])  
{  
    pre_interceptor.chamada ();  
    Result r = meuObjeto.metodo().invoke (parametros);  
    return pos_interceptor.chamada (r);  
}
```

```
class Pre_InterceptorNulo extends Pre_Interceptor {  
    void chamada () {}  
}  
class MeuObjetoNulo extends MeuObjeto {  
    MetodoCORBA metodo () { return MetodoCORBANulo; }  
}
```

# Princípio Básico

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

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

# Princípio Básico

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

<b>Cheiro</b>	<b>Refatoração a ser aplicada</b>
<b>Comentários</b> (desodorante 😊)	Extract Method (110) Introduce Assertion (267)
<b>Muitos parâmetros</b>	Replace Parameter with Method (292) Preseve Whole Object (288) 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.

# 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



---

**PUCRS** online  **uol**edtech.