Notas sobre Emergência e Refinamento Sucessivo

Type	Info
Topics	📚 Studies
Notebooks	Programação Profissional com Clean Code
Related Notes	Ementa do Curso

Design Emergente

- Definição
  - Boas práticas e princípios que podem emergir naturalmente no processo de desenvolvimento, ao invés de serem pré-definidos
  - 4 regras de Kent Beck; ajudam a criar designs eficientes; facilitam a aplicação de SRP (responsabilidade única) e DIP (inversão de dependência)
    - Executar todos os testes; incentiva a criação de classes pequenas e com única responsabilidade
    - Não conter duplicação; eliminar torna o código mais claro; leva a melhores abstrações e reutilização
    - Expressar a intenção do programador; nomear variáveis, classes e métodos de forma significativa é crucial; expressa claramente a intenção
    - Minimizar o número de classes e métodos; ajuda a manter o sistema mais compreensível e gerenciável

Práticas de Refatoração

- Definição

    - Reestruturar o código existente sem alterar o comportamento externo; processo contínuo e essencial
    - Melhorar nomes de variáveis e métodos; isolar funções; reduzir duplicação; introduzir abstrações
    - “First, make it work”, garantir que o código funcione antes de otimizar ou refatorar, que atenda os requisitos funcionais, priorize resultados esperados e passe em todos os testes
        - Entender os requisitos; interações com o usuário, entradas e saídas esperadas, regras de negócio
        - Escrever testes que verifiquem se o código atende aos requisitos; pode incluir testes de unidade, integração e aceitação
        - Implementar a solução inicial; prioridade deve ser o código funcionar
        - Verificar e validar; executar todos os testes e garantir que o código funciona
        - Corrigir os bugs; assegurar a correção funcional do código
    - “Then make it right”, envolve melhorar a estrutura, legibilidade, manutenção e desempenho; melhorar a qualidade interna do código sem alterar o comportamento externo
        - Refatorar, como extração de métodos, renomeação de variáveis, redução de duplicação e simplificação de estruturas complexas
        - Melhorar nomes; garantir que sejam descritivos e comunicativos
        - Simplificar lógica; estruturas complexas em partes menores e mais manejáveis
        - Reduzir acoplamento; minimizar as dependências para tornar o código mais flexível, o que facilita a alteração
        - Aumentar coesão; seguir a responsabilidade única (SRP)
        - Identificar pontos de gargalo no desempenho; evitar a otimização prematura para trazer somente benefícios reais
        - Reexecutar os testes para garantir que o comportamento não foi alterado; melhorias não comprometeram a funcionalidade externa
- Exemplos

- Versão inicial (case DataUtils)

```
public class DataUtils
{
    private int dia;
    private int mes;
    private int ano;

    public DataUtils(int dia, int mes, int ano)
    {
        this.dia = dia;
        this.mes = mes;
        this.ano = ano;
    }

    public bool EhAnoBissexto()
    {
        if ((ano % 4 == 0 && ano % 100 != 0) || (ano % 
        {
            return true;
        }
        return false;
    }

    public int DiasNoMes()
    {
        if (mes == 1 || mes == 3 || mes == 5 || mes == 
        {
            return 31;
        }
        else if (mes == 4 || mes == 6 || mes == 9 || me
        {
            return 30;
        }
        else if (mes == 2)
        {
            if (EhAnoBissexto())
            {
```

```
                return 29;
            }
            return 28;
        }
        return 0; // Caso de erro para meses inválidos
    }

    public override string ToString()
    {
        return $"{dia}/{mes}/{ano}";
    }
}
```

- Refatoração 1 (nomes melhorados para tornar mais descritivos)

```
public class DataUtils
{
    private int dia;
    private int mes;
    private int ano;

    public DataUtils(int dia, int mes, int ano)
    {
        this.dia = dia;
        this.mes = mes;
        this.ano = ano;
    }

    public bool EhAnoBissexto()
    {
        if ((ano % 4 == 0 && ano % 100 != 0) || (ano % 
        {
            return true;
        }
        return false;
    }

    public int DiasNoMes()
```

```
    {
        if (mes == 1 || mes == 3 || mes == 5 || mes == 
        {
            return 31;
        }
        else if (mes == 4 || mes == 6 || mes == 9 || me
        {
            return 30;
        }
        else if (mes == 2)
        {
            if (EhAnoBissexto())
            {
                return 29;
            }
            return 28;
        }
        return 0; // Caso de erro para meses inválidos
    }

    public override string ToString()
    {
        return $"{dia}/{mes}/{ano}";
    }
}
```

- Refatoração 2 (métodos menores e mais focados)

```
public class DataUtils
{
    private int dia;
    private int mes;
    private int ano;

    public DataUtils(int dia, int mes, int ano)
    {
        this.dia = dia;
        this.mes = mes;
```

```
        this.ano = ano;
    }

    public bool EhAnoBissexto()
    {
        return (ano % 4 == 0 && ano % 100 != 0) || (ano
    }

    // Método para verificar meses com 31 dias
    private bool MesTem31Dias()
    {
        return mes == 1 || mes == 3 || mes == 5 || mes =
    }

    // Método para verificar meses com 30 dias
    private bool MesTem30Dias()
    {
        return mes == 4 || mes == 6 || mes == 9 || mes =
    }

    // Método para validar o mês
    private bool MesValido()
    {
        return mes >= 1 && mes <= 12;
    }

    // Método principal agora usa métodos auxiliares
    public int ObterDiasNoMes()
    {
        if (!MesValido())
        {
            throw new ArgumentException("Mês inválido")
        }
        if (MesTem31Dias())
        {
            return 31;
        }
        if (MesTem30Dias())
```

```
        {
            return 30;
        }
        return EhAnoBissexto() ? 29 : 28;
    }

    public override string ToString()
    {
        return $"{dia}/{mes}/{ano}";
    }
}
```

- Refatoração 3 (reduzir a duplicação)

```
public class DataUtils
{
    private int dia;
    private int mes;
    private int ano;

    public DataUtils(int dia, int mes, int ano)
    {
        this.dia = dia;
        this.mes = mes;
        this.ano = ano;
    }

    public bool EhAnoBissexto()
    {
        return (ano % 4 == 0 && ano % 100 != 0) || (ano
    }

    // Uso de arrays para reduzir duplicação
    private bool MesTem31Dias() => new[] { 1, 3, 5, 7, 

    // Uso de arrays para reduzir duplicação
    private bool MesTem30Dias() => new[] { 4, 6, 9, 11 
```

```
    // Método para validar o mês centralizado
    private void ValidarMes()
    {
        if (mes < 1 || mes > 12)
        {
            throw new ArgumentException("Mês inválido")
        }
    }

    // Método principal agora usa métodos auxiliares e
    public int ObterDiasNoMes()
    {
        ValidarMes();
        if (MesTem31Dias())
        {
            return 31;
        }
        if (MesTem30Dias())
        {
            return 30;
        }
        return EhAnoBissexto() ? 29 : 28;
    }

    public override string ToString()
    {
        return $"{dia}/{mes}/{ano}";
    }
}
```

Refinamento Sucessivo

- Prática contínua de melhorar o código através de frequentes modificações; foco em pequenas melhorias
- Testes frequentes para que cada alteração não introduza novos problemas; documentação clara das mudanças facilita a compreensão e manutenção