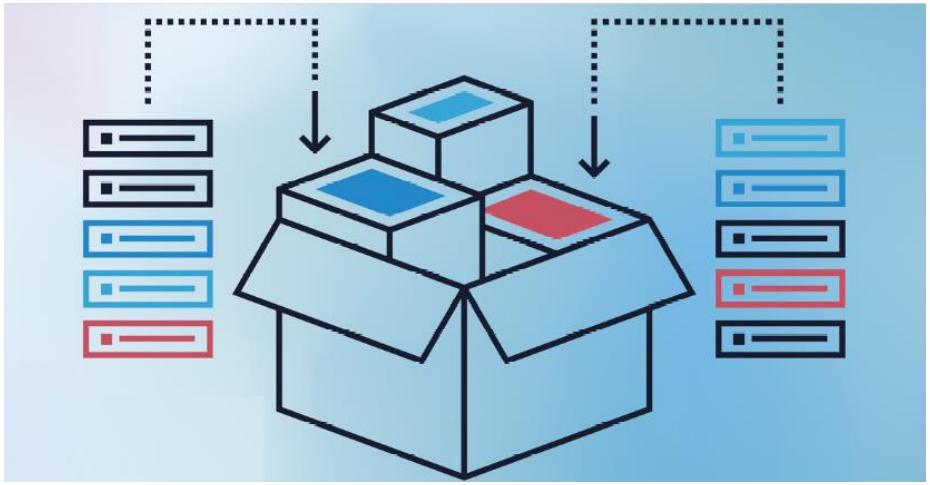
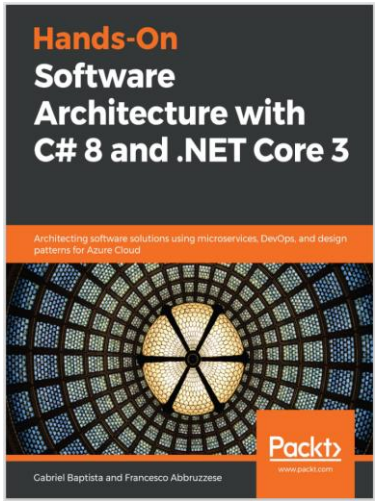
Arquitetura de Software

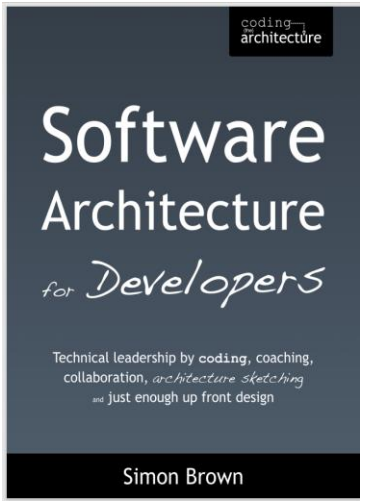
Unidade 3 - Arquitetura Monolítica com Design Modular

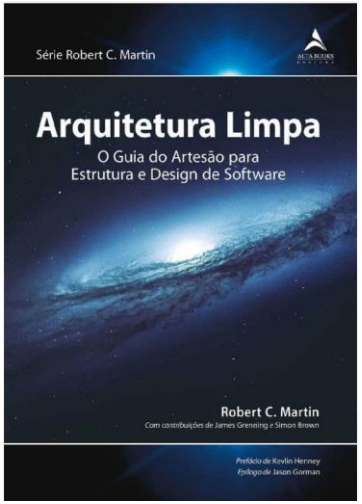


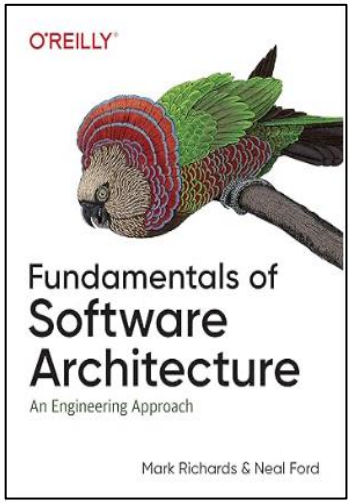
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@online.uscs.edu.br
aparecidovfreitas@gmail.com

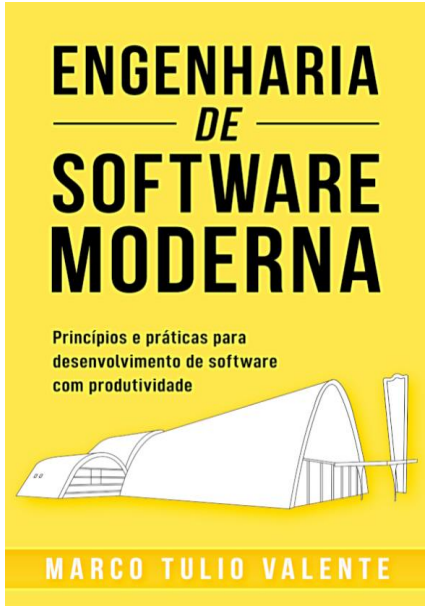
Bibliografia

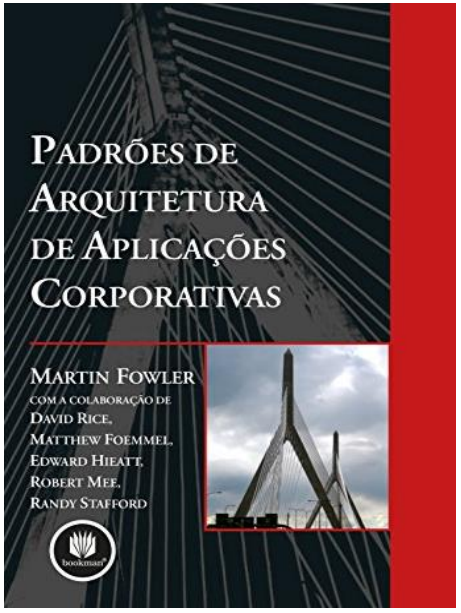


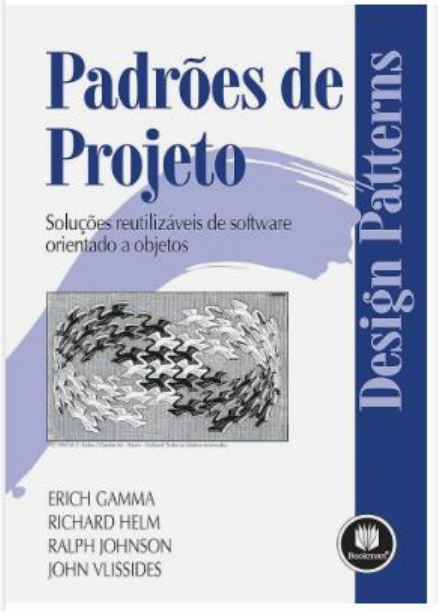




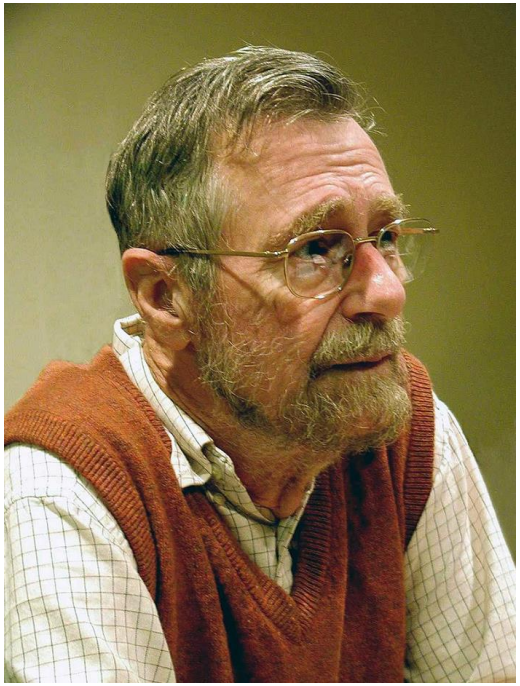








Design de Software



Edsger Dijkstra

1982. On the Role of Scientific Thought. In: Selected Writings on Computing: A personal Perspective.

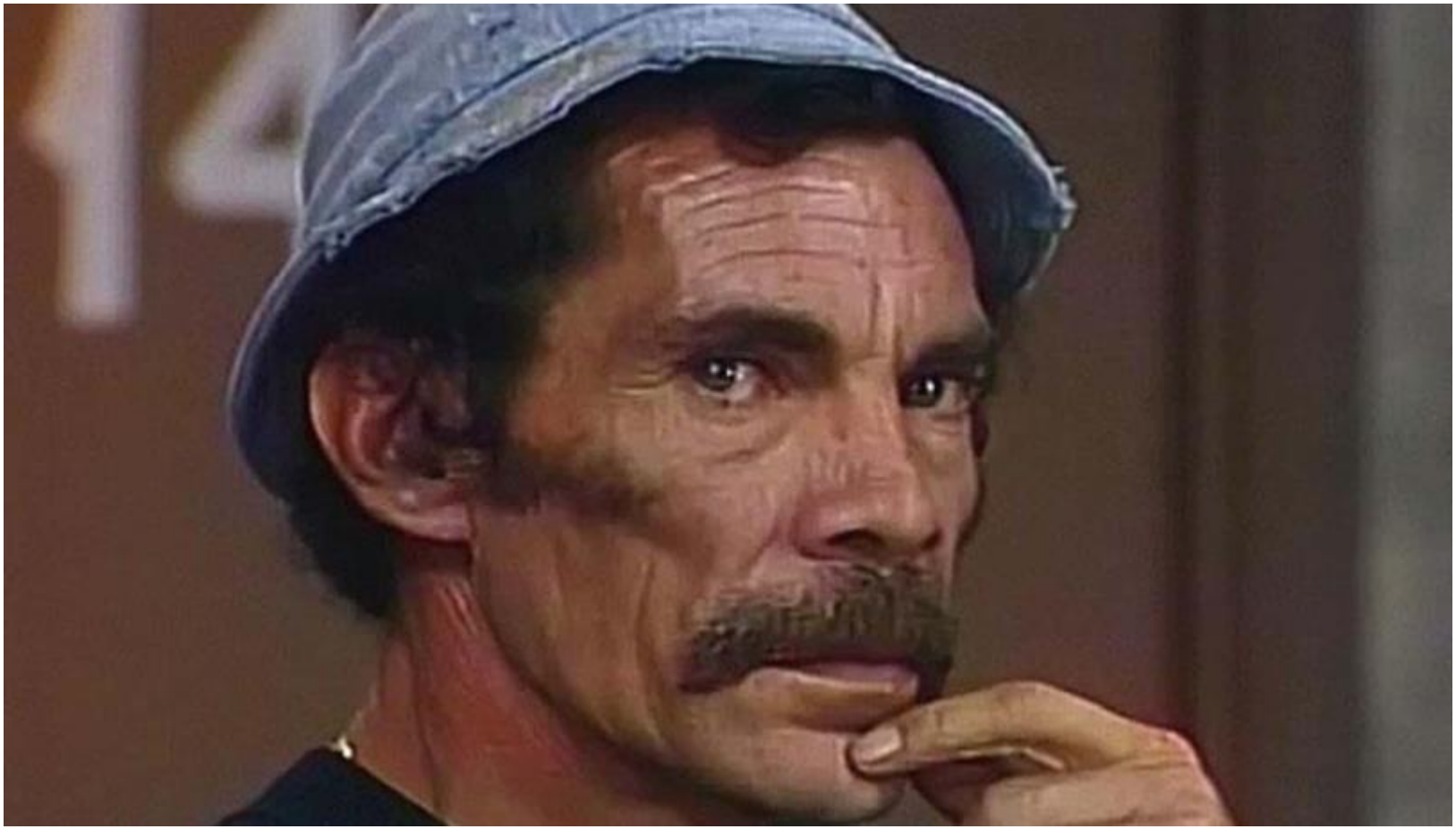
- Um problema **complexo** pode ser mais facilmente manuseado se for subdividido em partes que podem ser resolvidas independentemente;
- Isso leva a uma estratégia de **dividir para conquistar** — é mais fácil resolver um problema complexo quando você o **divide** em **partes gerenciáveis**;
- Este princípio de **design** de software está relacionado aos conceitos de: **modularidade**, **independência funcional** e **refinamento**.

Estratégia Divisão e Conquista

- Imagine que você está construindo um **quebra-cabeça gigante**;
- Ao invés de tentar encaixar todas as peças de uma vez, você separa o **quebra-cabeça** em seções menores que podem ser completadas individualmente e depois juntadas para formar a imagem completa;
- Essa é a essência da Técnica "**Divisão-e-Conquista**" largamente utilizada em **Projetos de Algoritmos**.

Estratégia Divisão e Conquista

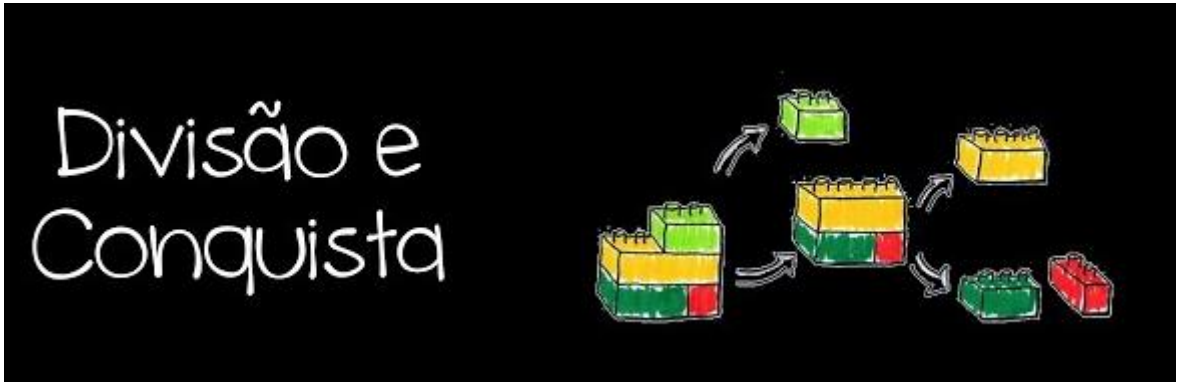
- Imagine que você está construindo um **quebra-cabeça gigante**;
- Ao invés de tentar encaixar todas as peças de uma vez, você separa o **quebra-cabeça** em seções menores que podem ser completadas individualmente e depois juntadas para formar a imagem completa;
- Essa é a essência da Técnica "**Divisão-e-Conquista**" largamente utilizada em Projetos de Algoritmos.



Como funciona a Estratégia Divisão e Conquista ?

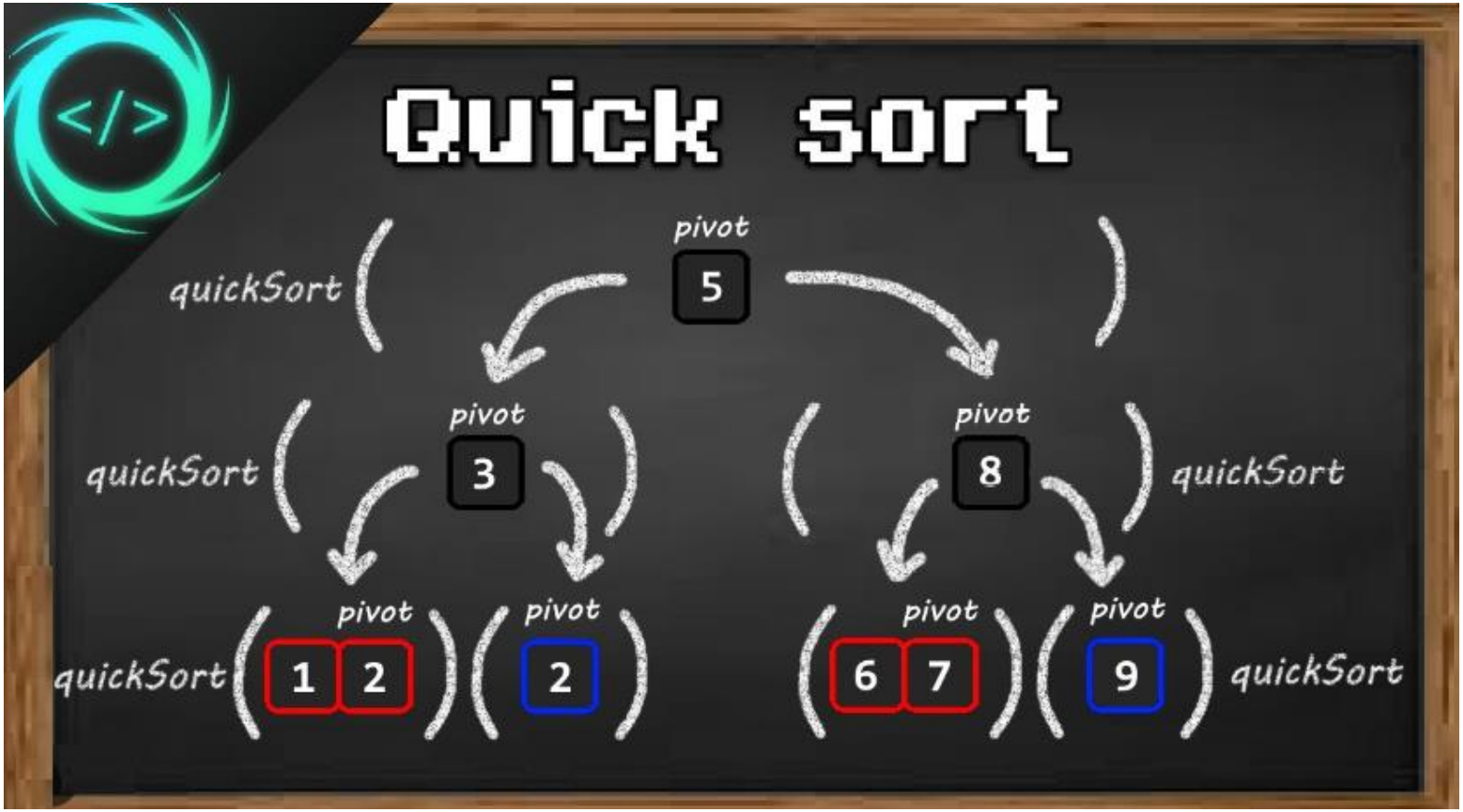
Como funciona a Estratégia Divisão e Conquista ?

1. **Divisão**: Primeiro, você quebra o problema grande e complexo em problemas menores e mais gerenciáveis. Esses problemas menores devem ser mais fáceis de resolver do que o problema original.
2. **Conquista**: Em seguida, você resolve cada um desses problemas menores. Às vezes, esses problemas menores ainda são complexos, então você pode precisar dividir ainda mais. O processo de divisão continua até que os problemas sejam suficientemente simples para serem resolvidos diretamente.
3. **Combinação**: Por fim, você combina as soluções dos problemas menores para formar a solução para o problema original.



Estratégia Divisão e Conquista - Exemplos

Estratégia Divisão e Conquista - Exemplos



Estratégia Divisão e Conquista - Exemplos

A estratégia de Divisão e Conquista aplicada em Algoritmos pode auxiliar na Arquitetura de Software ?

A estratégia de Divisão e Conquista aplicada em Algoritmos pode auxiliar na Arquitetura de Software ?

- Sim, na **Arquitetura de Software** pode-se empregar uma estratégia semelhante à **divisão e conquista** que corresponde à se dividir um sistema complexo em partes menores (**funções** ou **módulos**) com base em suas funcionalidades;

- Essa estratégia na Arquitetura de Software é conhecida por **Decomposição Funcional**.

Decomposição Funcional - Referências

- **"Software Engineering: A Practitioner's Approach" por Roger S. Pressman**: Este livro é uma fonte abrangente que cobre vários aspectos do desenvolvimento de software, incluindo técnicas de decomposição.
- **"Design Patterns: Elements of Reusable Object-Oriented Software" por Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides**: Embora focado em padrões de design, este livro também toca na importância da decomposição de software em componentes reutilizáveis.
- **"Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design" por Edward Yourdon e Larry L. Constantine**: Uma obra clássica que introduz conceitos fundamentais de design estruturado e decomposição.

Decomposição Funcional

- Aplicar **decomposição funcional** na arquitetura de um software significa **dividir** o sistema em componentes menores, com base nas funções que eles realizam;

- Essencialmente, é o processo de **quebrar um sistema complexo** em partes mais gerenciáveis, onde cada parte lida com um aspecto **específico** ou uma **funcionalidade** do sistema total;

- Essas partes menores são chamadas de **funções**, **módulos**, ou **serviços**.

Decomposição Funcional

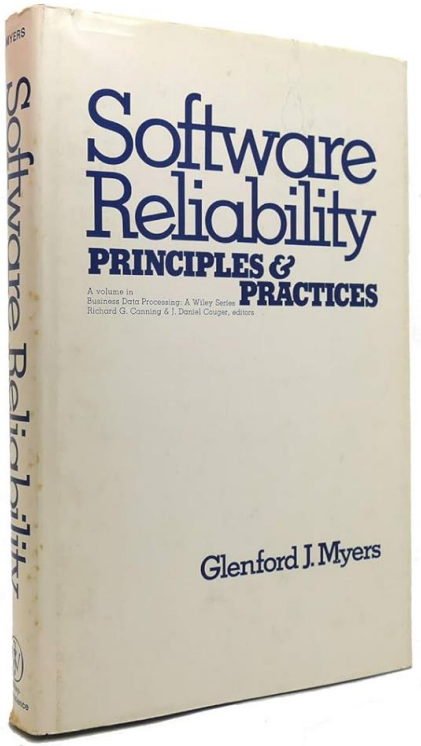
- **Aplicação na Arquitetura de Software**: Ao se aplicar **decomposição funcional** na arquitetura de um software, está se estruturando o software em **módulos** ou **componentes** que refletem diferentes funções ou tarefas que o sistema precisa realizar;

- Isso ajuda a organizar o **design** do software de maneira que cada componente tenha uma **responsabilidade** claramente definida, facilitando o desenvolvimento do sistema e a manutenção.

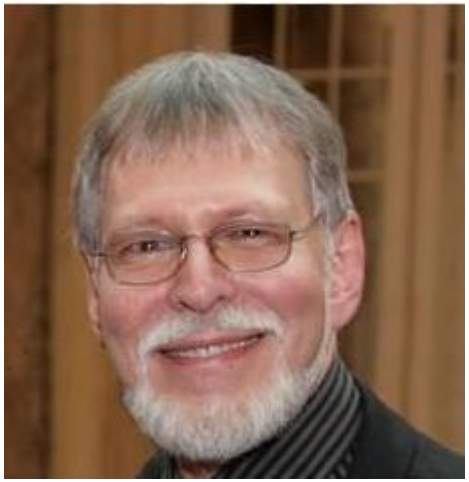
Modularidade

- Foi afirmado que "a **modularidade** é o único atributo do software que permite que um programa seja intelectualmente gerenciável" [Mye78].

Glenford Myers



Modularidade



Glenford Myers

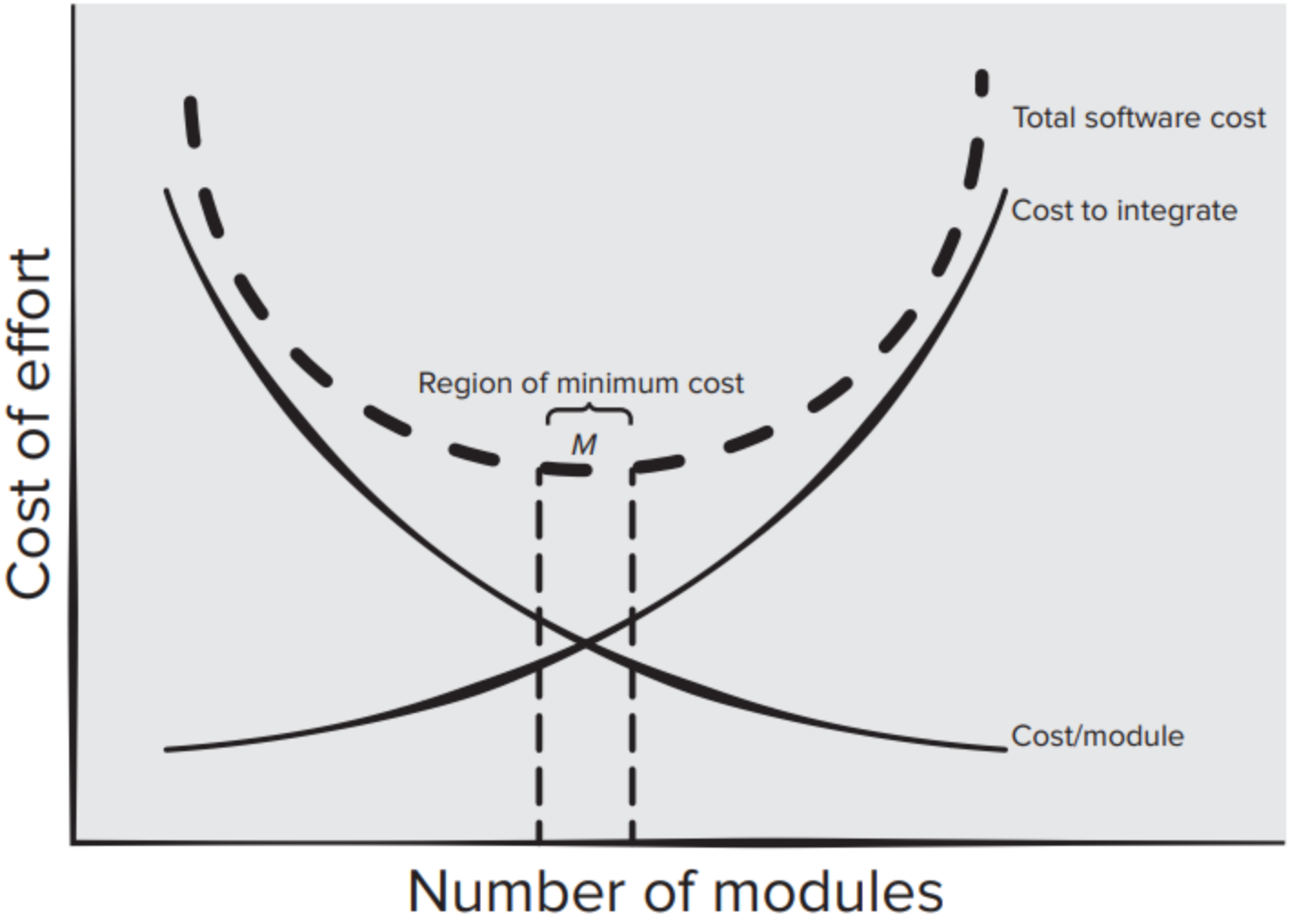
- Um grande programa composto por um único módulo, pode **não** ser facilmente compreendido por um engenheiro de software;
- O número de caminhos de controle, extensão de referência, número de variáveis e a complexidade geral tornariam a compreensão praticamente impossível;
- Em quase todas as instâncias, deve-se quebrar o **design** em **muitos módulos**, com o objetivo de tornar a compreensão mais fácil e reduzir o custo necessário para se construir o software.

Modularidade

- Dado o mesmo conjunto de requisitos, quanto mais módulos forem usados em seu programa, menores serão os tamanhos individuais;

- No entanto, à medida que o número de módulos cresce, o **esforço (custo) associado à integração dos módulos** uns com os outros **aumenta**.

Modularidade



Como decompor o software para se obter o melhor conjunto de módulos ?

Como decompor o software para se obter o melhor conjunto de módulos ?

David Parnas

On Criteria to Be Used in Decomposing Systems into Modules," CACM, vol. 14, no. 1, April 1972, pp. 221-227

- Em outras palavras, os **módulos** devem ser especificados e projetados de forma que as informações (algoritmos e dados) contidas dentro de um módulo sejam inacessíveis a outros módulos que não necessitam de tais informações;
- Esse princípio é conhecido por **Information Hiding** (Ocultamento da informação).

Ocultamento da informação

- Implica que a **modularidade** eficaz pode ser alcançada definindo-se um conjunto de módulos independentes que comunicam uns com os outros apenas as informações necessárias para alcançar a função do software;

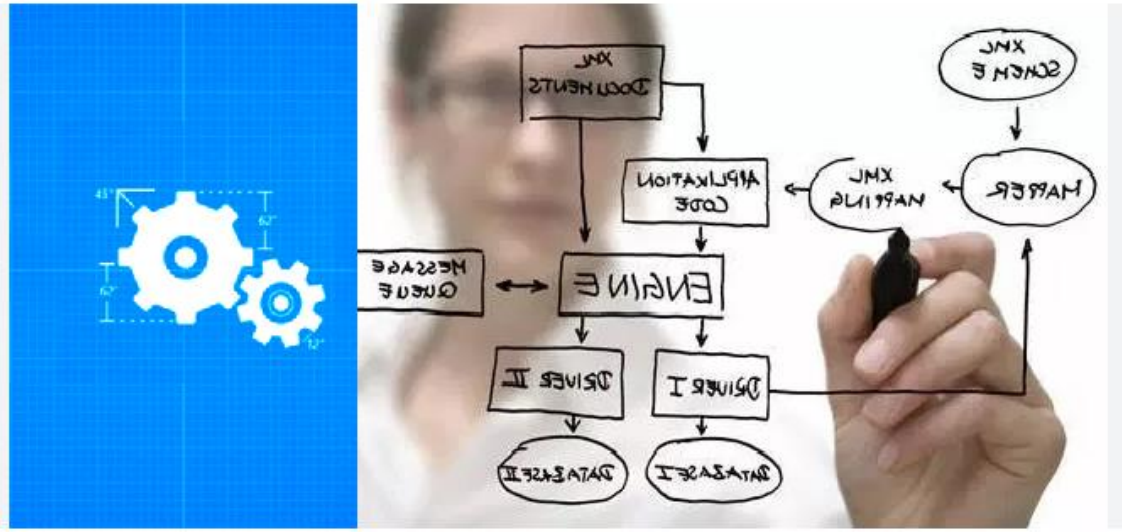
- Benefícios ocorrem quando modificações são necessárias durante o **teste** e, posteriormente, durante a **manutenção** do software;

- Como a maioria dos dados e detalhes procedurais são ocultados de outras partes do software, erros inadvertidos introduzidos durante a modificação têm menos probabilidade de se propagar para outros locais dentro do software (**Side Effects**).

Independência Funcional

- A **independência funcional** é alcançada desenvolvendo módulos com uma função "única" e sem interação excessiva com outros módulos;

- **Módulos independentes** são mais fáceis de se manter (e testar) porque efeitos secundários causados por modificação de design ou código são limitados, a propagação de erros é reduzida, e módulos reutilizáveis são possíveis.

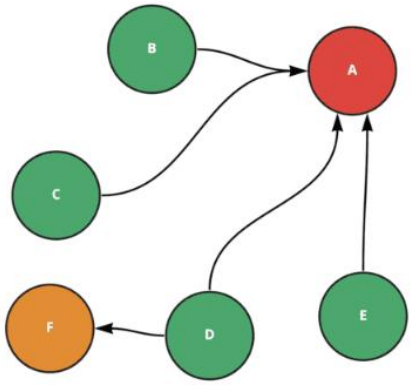


Como avaliar a Independência Funcional ?

Coesão

- **Coesão** é uma extensão natural do conceito de ocultamento de informação. Um **módulo** coeso executa uma **única tarefa**, requerendo pouca interação com outros componentes em outras partes de um programa;

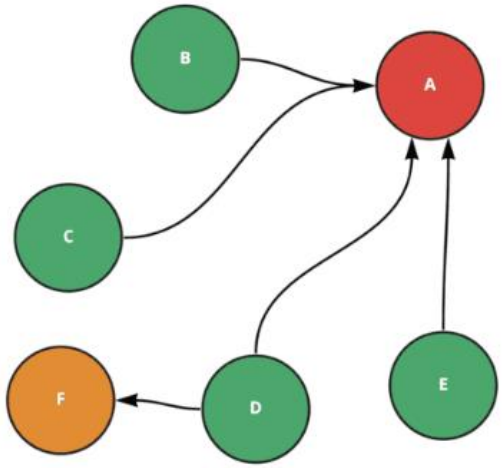
- Dito de forma simples, um **módulo coeso** deve (idealmente) fazer apenas uma coisa. Embora você deva sempre almejar alta **coesão** (ou seja, foco único), muitas vezes é necessário que um componente de software desempenhe múltiplas funções. No entanto, **módulos** que desempenham muitas funções não relacionadas devem ser **evitados** se um bom design for alcançado.



Acoplamento

- **Acoplamento** é uma indicação de interconexões entre módulos em uma estrutura de software;

- O **Acoplamento** depende da complexidade da interface entre módulos, do ponto em que a entrada ou referência é feita a um módulo, e de quais dados passam pela interface. No design de software, deve-se se esforçar para ter o **menor acoplamento** possível. Conectividade simples entre módulos resulta em software que é mais fácil de entender e menos propenso a propagar erros encontrados em um módulo para outros módulos do sistema.



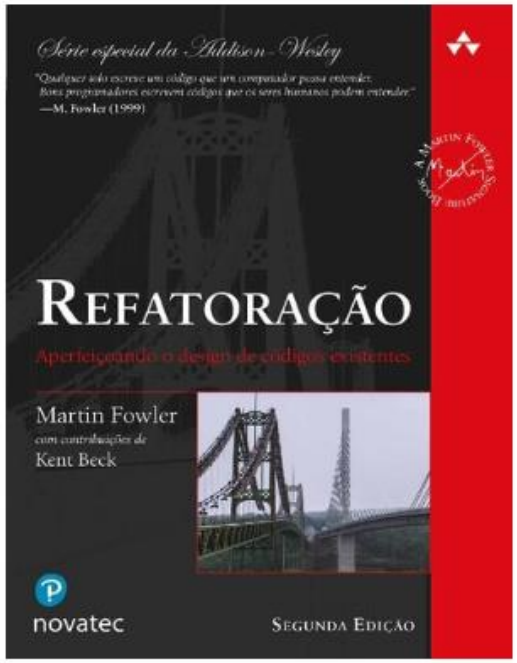
Refatoração

Martin Fowler

- Técnica de reorganização que simplifica o design (ou código) de um componente sem alterar sua função ou comportamento;
- Fowler define da seguinte maneira: "**Refatoração** é o processo de se mudar um sistema de software de tal forma que não se altere o comportamento externo do código e ainda melhore sua estrutura interna."

Refatoração

- Quando um software é **refatorado**, o design existente é examinado em busca de redundâncias, elementos de design não utilizados, algoritmos ineficientes ou desnecessários, estruturas de dados mal construídas ou inadequadas, ou qualquer outra falha de design que possa ser corrigida para gerar um design melhor.

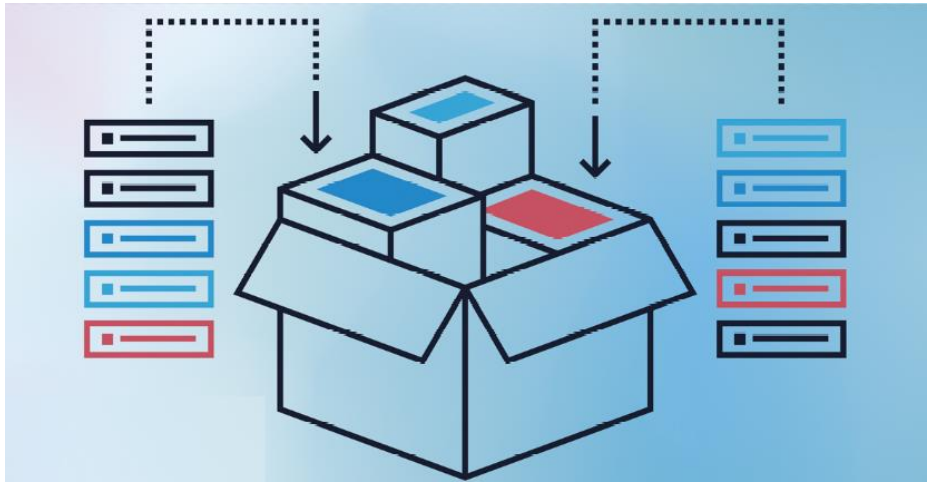


Refatoração

- Por exemplo, uma primeira iteração de design pode resultar em um grande módulo que exibe **baixa coesão** (ou seja, realiza **três funções** que têm apenas uma relação limitada entre si);

- Após uma consideração cuidadosa, pode-se decidir que o módulo deve ser **refatorado** em três módulos separados, cada um dos quais exibindo **alta coesão**. O resultado será um software mais fácil de integrar, mais fácil de testar e mais fácil de manter.

Design Modular



Design Modular

Importância:

1. **Facilita a compreensão**: Como cada módulo é uma unidade separada com uma função específica, fica mais fácil para os desenvolvedores entenderem como o sistema funciona.
2. **Promove a reutilização**: Os módulos podem ser projetados de forma que possam ser reutilizados em diferentes partes do sistema ou até mesmo em projetos diferentes, economizando tempo e esforço.
3. **Simplifica a manutenção e atualizações**: Quando é necessário fazer manutenção ou atualizar uma parte do sistema, os desenvolvedores podem focar apenas no módulo específico afetado, sem precisar mexer no sistema inteiro.
4. **Facilita o trabalho em equipe**: Como os módulos podem ser desenvolvidos de forma independente, diferentes equipes ou desenvolvedores podem trabalhar em diferentes módulos simultaneamente, aumentando a eficiência.
5. **Melhora a testabilidade**: Testar um sistema grande pode ser complicado, mas com a arquitetura modular, é possível testar cada módulo de forma independente, facilitando a detecção e correção de erros.

Design Modular

Detalhes Adicionais:

- **Encapsulamento**: Cada módulo encapsula suas próprias funcionalidades e dados, expondo apenas interfaces necessárias para a interação com outros módulos, protegendo suas implementações internas.
- **Coesão e Acoplamento**: A arquitetura modular promove alta coesão dentro dos módulos e baixo acoplamento entre eles. Isso significa que as funções dentro de um módulo estão fortemente relacionadas e trabalham juntas para uma tarefa específica, enquanto a dependência entre módulos diferentes é mantida mínima.
- **Interoperabilidade**: Os módulos comunicam-se através de interfaces bem definidas ou APIs (Application Programming Interfaces), permitindo que mudanças em um módulo tenham impacto mínimo nos outros módulos, desde que as interfaces sejam mantidas.
- **Flexibilidade e Reusabilidade**: Módulos independentes podem ser facilmente reutilizados em diferentes partes do projeto ou em projetos diferentes, economizando tempo e recursos.

Design Modular – Tarefa 3_01

Código **Java**, com arquitetura **monolítica**, que cria uma lista com **10 perguntas** sobre as capitais de diferentes países;

A aplicação, **via console**, envia a pergunta e o usuário digita a resposta e envia para o programa. No final do quiz, são mostradas as respostas corretas, erradas e a nota do usuário (de 0 a 10). A aplicação é monolítica uma vez que qualquer alteração irá requerer recompilação de todo o código;

Refatore o código desenvolvido na **Tarefa 2_01** e aplique os conceitos de Design **Modular**.

```java
package br.uscs;
//-------------------------------------------------
public class DadosQuiz {

    private String[][] capitaisPaises = {

            { "Itália", "Roma" },
            { "França", "Paris" },
            { "Espanha", "Madrid" },
            { "Alemanha", "Berlim" },
            { "China", "Pequim" },
            { "Portugal", "Lisboa" },
            { "Rússia", "Moscou" },
            { "Suiça", "Berna" },
            { "Áustria", "Viena" },
            { "Egito", "Cairo" } };

//-------------------------------------------------
    public String[][] getCapitaisPaises() {
        return capitaisPaises;
    }
}
//-------------------------------------------------
```

```java
package br.uscs;
//-----------------------------------------------------------
import java.util.Scanner;
//-----------------------------------------------------------
class LogicaQuiz {
    private DadosQuiz dadosQuiz = new DadosQuiz();
    private int score = 0;
//-----------------------------------------------------------
    public void processarQuiz() {

        Scanner scanner = new Scanner(System.in);
        String[][] capitaisPaises = dadosQuiz.getCapitaisPaises();

        for (int i = 0; i < capitaisPaises.length; i++) {
            System.out.println("Qual é a capital do país " + capitaisPaises[i][0] + "?");
            String respostaUsuario = scanner.nextLine();
            verificarResposta(respostaUsuario, capitaisPaises[i][1]);
        }

        scanner.close();
        finalizarQuiz(capitaisPaises.length);
    }
//-----------------------------------------------------------
```

```
23 //------------------------------------------------------------------
24     private void verificarResposta(String resposta, String respostaCorreta) {
25         if (resposta.trim().equalsIgnoreCase(respostaCorreta)) {
26             System.out.println("Correto!");
27             score++;
28         } else {
29             System.out.println("Errado! A resposta correta é: " + respostaCorreta);
30         }
31     }
32 //------------------------------------------------------------------
33     private void finalizarQuiz(int totalPerguntas) {
34         System.out.println("\nO quiz terminou. Você acertou " + score + " de " +
35                 totalPerguntas + " perguntas.");
36         int nota = score * 10 / totalPerguntas;
37         System.out.println("Sua nota é: " + nota);
38         avaliarDesempenho(nota);
39     }
40 //------------------------------------------------------------------
41     private void avaliarDesempenho(int nota) {
42         if (nota > 8) {
43             System.out.println("Ótimo desempenho!");
44         } else if (nota < 6) {
45             System.out.println("Baixo desempenho!");
46         } else {
47             System.out.println("Bom desempenho!");
48         }
49     }
50 }
51 //------------------------------------------------------------------
```

```java
package br.uscs;

public class Quiz_Capitais_1_A {
    public static void main(String[] args) {
        new LogicaQuiz().processarQuiz();
    }
}
```

Design Modular – Tarefa 3_02

Código **Java**, com arquitetura **monolítica**, que cria uma lista com **10 perguntas** sobre as capitais de diferentes países;

A aplicação, **via console**, envia a pergunta e o usuário digita a resposta e envia para o programa. No final do quiz, são mostradas as respostas corretas, erradas e a nota do usuário (de 0 a 10). A aplicação é monolítica uma vez que qualquer alteração irá requerer recompilação de todo o código;

Refatore o código desenvolvido na **Tarefa 2_02** e aplique os conceitos de Design **Modular**.