

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo - IFSP Câmpus Jacareí

**Tecnologia em Análise e Desenvolvimento de Sistemas -
ADS**

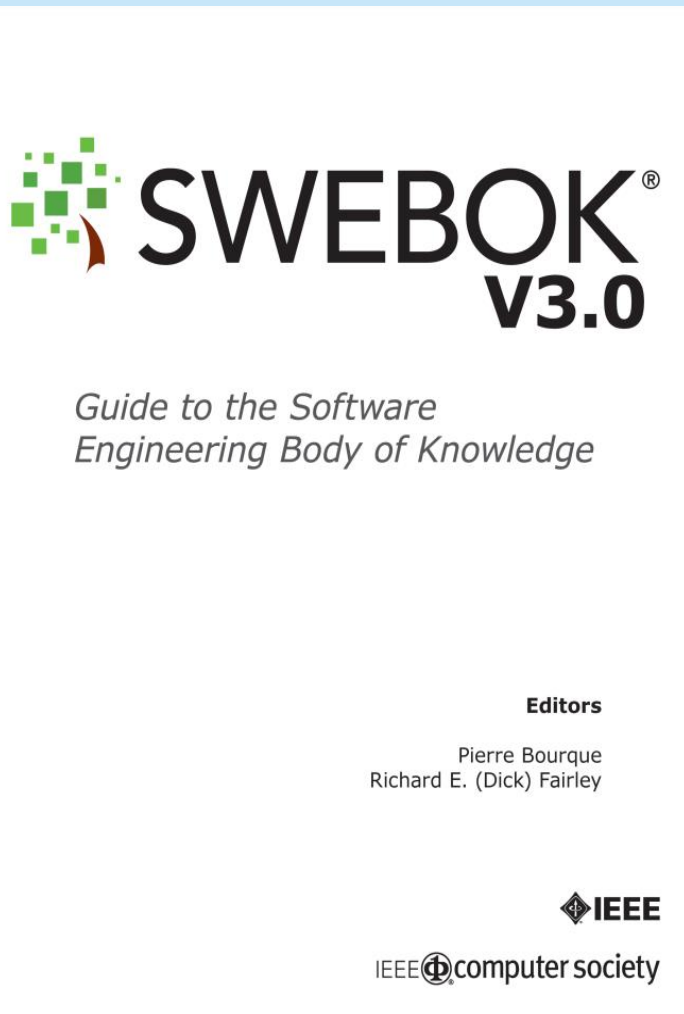
2º Semestre de 2023

Engenharia de Software – JCRESW1

Prof. Lineu Mialaret

**Aula 5: Conceitos Básicos em Engenharia de
Software (3)**

Área da Engenharia de Software



- O *Guide to the Software Engineering Body of Knowledge* (SWEBOK) é um documento organizado pela IEEE *Computer Society* (uma sociedade científica internacional), com o apoio de diversos pesquisadores e profissionais da indústria de computação.
- O objetivo do SWEBOK é precisamente documentar o corpo (campo) de conhecimento que caracteriza a área que hoje se chama de Engenharia de Software.

Campos da Engenharia de Software

- O SWEBOK define 12 campos de conhecimento em Engenharia de Software:

- Engenharia de Requisitos
- Projeto de Software
- Construção de Software
- Teste de Software
- Manutenção de Software
- Gerência de Configuração
- Gerência de Projeto
- Processo de Software
- Modelos de Software
- Qualidade de Software
- Prática Profissional
- Aspectos Econômicos

O SWEBOK inclui ainda mais três áreas de conhecimento:

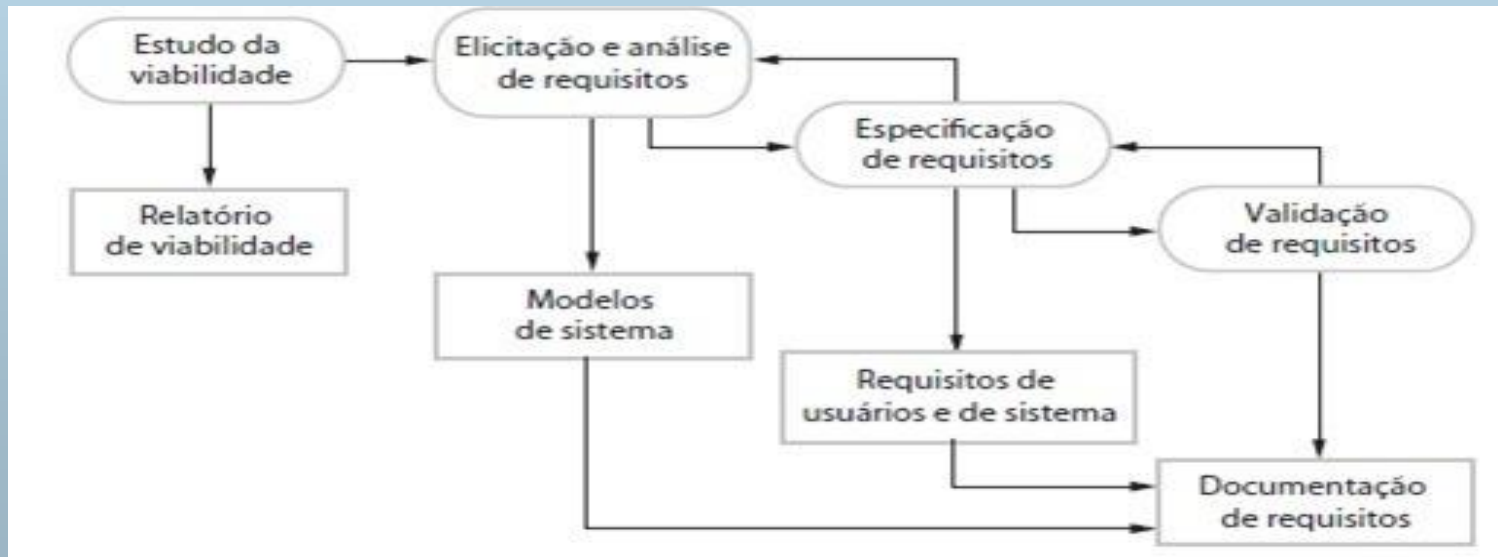
- Fundamentos de Computação.
- Fundamentos de Matemática.
- Fundamentos de Engenharia.

Chapter 14: Mathematical Foundations

1. Set, Relations, Functions
 - 1.1. Set Operations
 - 1.2. Properties of Set
 - 1.3. Relation and Function
 2. Basic Logic
 - 2.1. Propositional Logic
 - 2.2. Predicate Logic
 3. Proof Techniques
 - 3.1. Methods of Proving Theorems
 4. Basics of Counting
 5. Graphs and Trees
 - 5.1. Graphs
 - 5.2. Trees
 6. Discrete Probability
 7. Finite State Machines
 8. Grammars
 - 8.1. Language Recognition
 9. Numerical Precision, Accuracy, and Errors
 10. Number Theory
 - 10.1. Divisibility
 - 10.2. Prime Number, GCD
 11. Algebraic Structures
 - 11.1. Group
 - 11.2. Rings
- Matrix of Topics vs. Reference Material

Engenharia de Requisitos

- Os requisitos de um sistema de software definem o que ele deve fazer e como ele deve operar.
- A Engenharia de Requisitos inclui o conjunto de atividades realizadas com o objetivo de elicitar, definir, analisar, documentar e validar os requisitos de um sistema de software.



Engenharia de Requisitos (cont.)

- Em uma primeira classificação, os requisitos podem ser Funcionais ou Não-Funcionais.
- Requisitos Funcionais definem o que um sistema deve fazer, ou seja, quais as funcionalidades ou os serviços ele deve implementar.

Id	Descrição
RF1	O usuário pode enviar arquivos para seu disco virtual
RF2	O usuário pode recuperar arquivos de seu disco virtual
RF3	O usuário pode excluir arquivos do seu disco virtual
RF4	O usuário pode compartilhar arquivos ou pastas com outros usuários
RF5	O usuário pode tornar dado arquivo público para ser recuperado por qualquer pessoa
RF6	O usuário pode pesquisar arquivos por nome ou conteúdo
RF7	O administrador dos usuários pode criar, modificar ou excluir usuários
RF8	O administrador precisa de ferramentas para tarifar o usuário de acordo com o uso de seu disco virtual (espaço usado, bytes transferidos, etc.)

Tabela 1 – Requisitos Funcionais

Engenharia de Requisitos (cont.)

- Requisitos Não-Funcionais definem como um sistema deve operar, sob quais restrições e com qual qualidade de serviço.
- Exemplos de Requisitos Não-Funcionais:
 - Desempenho, Disponibilidade, Tolerância a Falhas, Segurança, Privacidade, Interoperabilidade, Capacidade, Manutenibilidade e Usabilidade, dentre outros.

Id	Descrição
RNF1	Tempo de resposta
RNF2	Disponibilidade dos serviços
RNF3	Disponibilidade dos arquivos
RNF4	Segurança e privacidade
RNF5	Integridade dos dados
RNF6	Consistência dos dados
RNF7	Durabilidade dos dados

Tabela 2 – Requisitos Não Funcionais

Engenharia de Requisitos (cont.)

- Exemplo: Sistema de Caixa Eletrônico.
- Requisitos Funcionais:

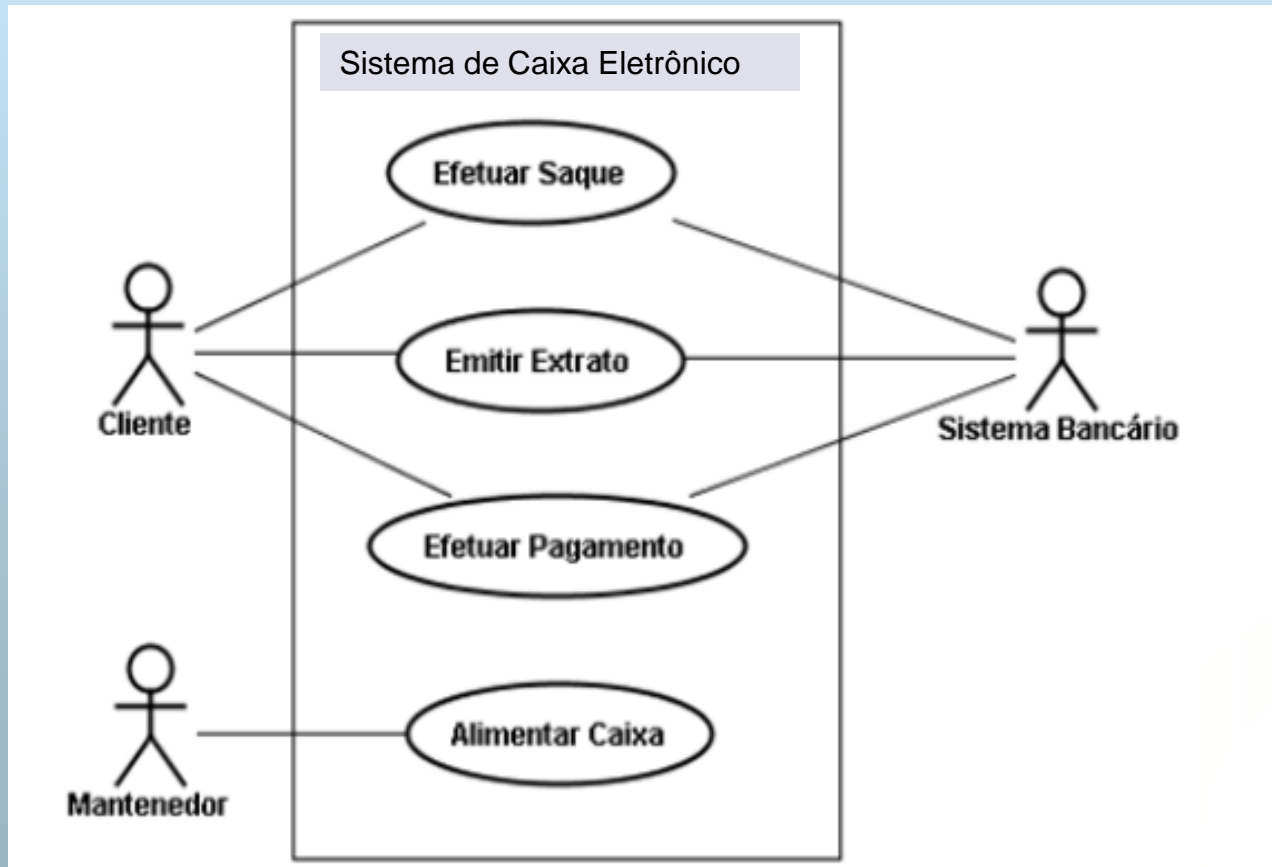


Diagrama de Casos de Uso

Engenharia de Requisitos (cont.)

- Exemplo: Sistema de Caixa Eletrônico.
- Requisitos Não Funcionais:
 - Desempenho - informar o saldo da conta em menos de 3 segundos.
 - Disponibilidade - estar on-line 99% do tempo.
 - Tolerância a Falhas - continuar operando mesmo se um determinado banco (centro de dados) ficar *off-line*.
 - Segurança - criptografar todos os dados trocados com as agências.
 - Privacidade - não disponibilizar para terceiros dados de clientes.
 - Interoperabilidade - integrar com sistemas do Banco Central.
 - Capacidade - ser capaz de armazenar dados de 1 milhão de clientes.
 - Usabilidade - ter uma versão para deficientes visuais.

Projeto (*Design*) de Software

- O Projeto (*design*) define a estruturação do software.
- Durante o projeto (*design*) de um sistema de software, são definidas suas principais unidades de código, inicialmente apenas no nível de interfaces, incluindo Interfaces Providas e Interfaces Requeridas.
 - Interfaces Providas são aqueles serviços que uma unidade de código torna público para uso pelo resto do sistema.
 - Interfaces Requeridas são aquelas interfaces das quais uma unidade de código depende para funcionar.
- Durante o projeto de um sistema de software, costuma-se não entrar em detalhes de implementação de cada unidade de código, tais como a implementação dos métodos de uma classe, caso o sistema seja implementado em uma linguagem orientada a objetos.

Projeto (*Design*) de Software (cont.)

- Exemplo:
 - Durante o projeto (*design*) de um sistema de *home-banking*, pode-se propor uma classe *ContaBancaria* para representar as contas bancárias, como a apresentada a seguir.

```
class ContaBancaria {  
    private Cliente cliente;  
    private double saldo;  
    public double getSaldo() { ... }  
    public String getNomeCliente() { ... }  
    public String getExtrato (Date inicio) { ... }  
    ...  
}
```

Projeto (*Design*) de Software (cont.)

- Exemplo:
 - Durante o projeto de um sistema de *home-banking*, pode-se propor uma classe *ContaBancaria* para representar as contas bancárias, como a apresentada a seguir

```
class ContaBancaria {  
    private Cliente cliente;  
    private double saldo;  
    public double getSaldo() { ... }  
    public String getNomeCliente() { ... }  
    public String getExtrato (Date inicio) { ... }  
    ...  
}
```

- *ContaBancaria* oferece uma interface para as demais classes do sistema, na forma de três métodos públicos, que constituem a interface provida pela classe.

Projeto (*Design*) de Software

■ Exemplo:

- Durante o projeto de um sistema de *home-banking*, pode-se propor uma classe *ContaBancaria* para representar as contas bancárias, como a apresentada a seguir

```
class ContaBancaria {  
  { private Cliente cliente;  
    private double saldo;  
    public double getSaldo() { ... }  
    public String getNomeCliente() { ... }  
    public String getExtrato (Date inicio) { ... }  
    ...  
}
```

- *ContaBancaria* também depende de uma outra classe, *Cliente*; logo, a interface de *Cliente* é uma interface requerida por *ContaBancaria*.
- Muitas vezes, interfaces requeridas são chamadas de Dependências.
- Isto é, *ContaBancaria* possui uma dependência para *Cliente*.

Projeto (*Design*) de Software

- Quando o projeto (*design*) é realizado em um nível mais alto de granularização e as unidades de código possuem maior granularidade (constituem-se nos pacotes ou módulos), ele é classificado como um Projeto Arquitetural ou Arquitetura de Software.
 - Arquitetura de Software trata da organização de um sistema em um nível de abstração mais alto do que aquele que envolve classes ou construções semelhantes.

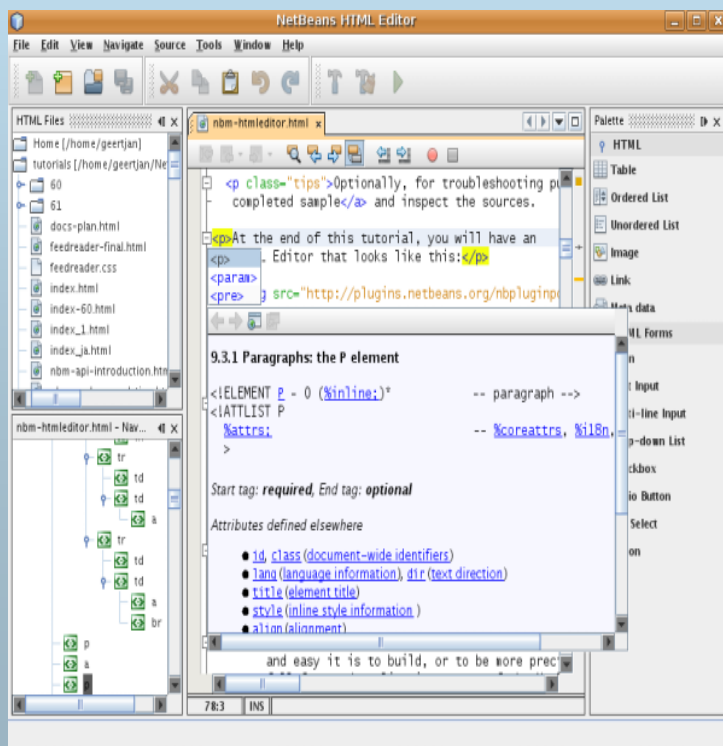


Construção de Software

- Construção trata da implementação, isto é, codificação do sistema.
- Nesse momento, existem diversas decisões que precisam ser tomadas:
 - Definir os algoritmos e estruturas de dados que serão utilizados e implementados.
 - Definir os *frameworks* e bibliotecas de terceiros que serão usados.
 - Definir técnicas para tratamento de exceções.
 - Definir padrões na documentação de código.
 - ...

Construção de Software (cont.)

- Definir as ferramentas que serão usadas no desenvolvimento, incluindo compiladores, ambientes integrados de desenvolvimento (*Integrated Development Environment* - IDE), depuradores, gerenciadores de bancos de dados, ferramentas para construção de interfaces, etc.



Teste de Software

- Teste de Software pode ser entendido em parte como a execução de um programa com um conjunto finito de casos de testes, com o objetivo de verificar se ele possui o comportamento esperado.
 - A seguinte frase seguir, bastante famosa, de Edsger W. Dijkstra (Prêmio Turing em Computação em 1982), sintetiza não apenas os benefícios de testes, mas também suas limitações.

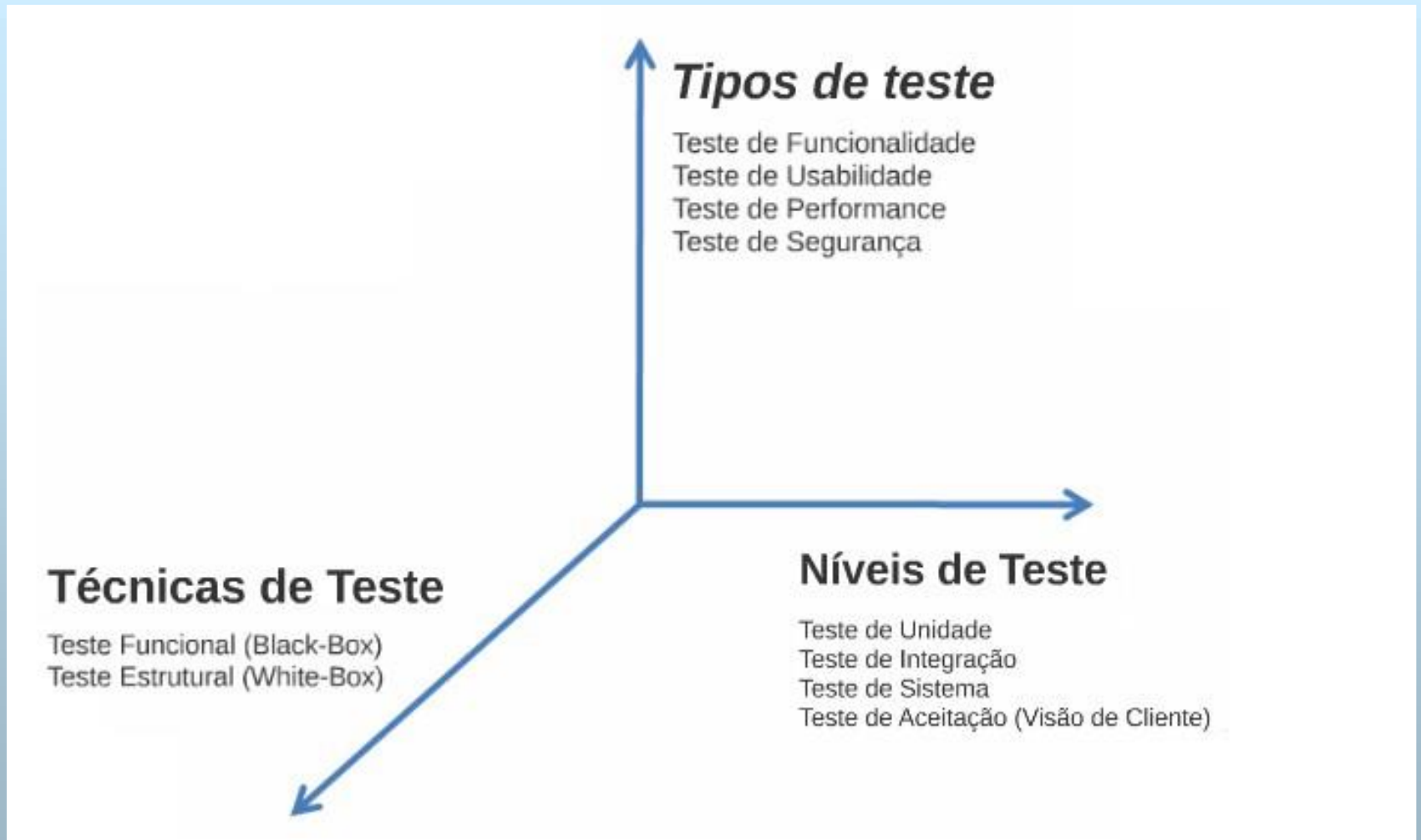


"Testes de software mostram a presença de bugs, mas não a sua ausência."

Teste de Software (cont.)

- Existem diversos tipos de testes.
 - Testes de Unidade (Componente), quando se testa uma pequena unidade do código, como uma classe ou método.
 - Testes de Integração, quando se testa uma unidade de maior granularidade, como um conjunto de classes.
 - Testes de Performance, quando se submete o sistema a uma carga de processamento para verificar seu desempenho.
 - Testes de Usabilidade, quando o objetivo é verificar a usabilidade da interface do sistema.
 - Teste de Caixa Preta, quando objetivo é testar a funcionalidade do sistema.
 - Etc.

Teste de Software (cont.)



Caracterização dos Diferentes Níveis, Tipos e Técnicas de Teste de Software.

Teste de Software (cont.)

- Testes podem ser utilizados tanto para Verificação como para Validação de sistemas.
 - Verificação tem como o objetivo garantir que um sistema atende à sua especificação.
 - Validação tem como objetivo garantir que um sistema atende às necessidades de seus clientes.
- A diferença entre os conceitos só faz sentido porque pode acontecer da especificação de um sistema não expressar as necessidades de seus clientes.
- Exemplo:
 - Essa diferença pode ser causada por um erro no levantamento de requisitos, ou seja, os desenvolvedores não entenderam os requisitos do sistema ou o cliente não foi capaz de explicá-los precisamente.

Teste de Software (cont.)

- Existem duas frases, muito usadas, que resumem as diferenças entre Verificação e Validação:

- **Verificação:** estamos implementando o sistema corretamente? Isto é, de acordo com seus requisitos.
- **Validação:** estamos implementando o sistema correto? Isto é, aquele que os clientes ou o mercado está querendo.

- Quando se realiza um teste de um método, para verificar se ele retorna o resultado especificado, está-se realizando uma Atividade de Verificação.
- Quando se realiza um teste funcional de aceitação, ao lado do cliente, isto é, mostrando para ele os resultados e funcionalidades do sistema, está-se realizando uma Atividade de Validação.

Teste de Software (cont.)

- É importante definir e distinguir os conceitos relacionados a testes: Erro ou Engano, Defeito ou *Bug* e Falha.
- Para ilustrar a diferença entre eles, seja o seguinte código para calcular a área de um círculo, dependendo de uma determinada condição:

```
if (condicao)
    area = pi * raio * raio * raio;
```

- Esse código possui um Defeito (ocasionado por um Erro ou Engano cometido pelo programador), pois a área de um círculo é π vezes raio ao quadrado, e não ao cubo.
 - *Bug* é um termo mais informal, usado com objetivos às vezes diversos.
 - Mas o uso mais comum é como sinônimo de Defeito.

Teste de Software (cont.)

```
if (condicao)
    area = pi * raio * raio * raio;
```

- Uma Falha ocorre quando um código com Defeito for executado
 - Se a condição do *if* do programa acima for verdadeira, e isso levar o programa a apresentar um resultado incorreto.
- Portanto, nem todo Defeito ou *Bug* ocasiona Falhas, pois pode ser que o código defeituoso nunca seja executado.
- Resumindo, Código Defeituoso é aquele que não está de acordo com a sua especificação.
 - Se esse código for executado e de fato levar o programa a apresentar um resultado incorreto, diz-se que ocorreu uma Falha.

Manutenção de Software

- Assim como os sistemas tradicionais de engenharia, software também precisa de manutenção.
 - Manutenção = ação de manter, sustentar, consertar ou conservar alguma coisa ou algo.
- Será utilizada a seguinte classificação para os tipos de manutenções que podem ser realizadas em sistemas de software:
 - Manutenção Corretiva
 - Manutenção Preventiva
 - Manutenção Adaptativa
 - Manutenção Evolutiva
 - *Refactorings* (Refatoramentos)

Manutenção Corretiva

- Manutenção Corretiva tem como objetivo corrigir *bugs* reportados por usuários ou outros desenvolvedores.
- Exemplo de Manutenção Corretiva:
 - Corrigir uma instrução de um código fonte que foi codificada com engano ou erro.



Manutenção Preventiva

- Manutenção Preventiva tem como objetivo corrigir *bugs* latentes no código, que ainda não causaram falhas junto aos usuários do sistema.
- Exemplo de Manutenção Preventiva:
 - Um exemplo de manutenção preventiva foram as atividades de manutenção realizadas por diversas empresas antes da virada do último milênio, de 1999 para 2000.
 - Nessa época, diversos sistemas armazenavam o ano de uma data com dois dígitos, isto é, as datas tinham o formato DD-MM-AA.
 - As empresas ficaram receosas de que, em 2000 e nos anos seguintes, algumas operações envolvendo datas retornassem valores incorretos, pois uma subtração 00 - 99, por exemplo, poderia dar um resultado inesperado.

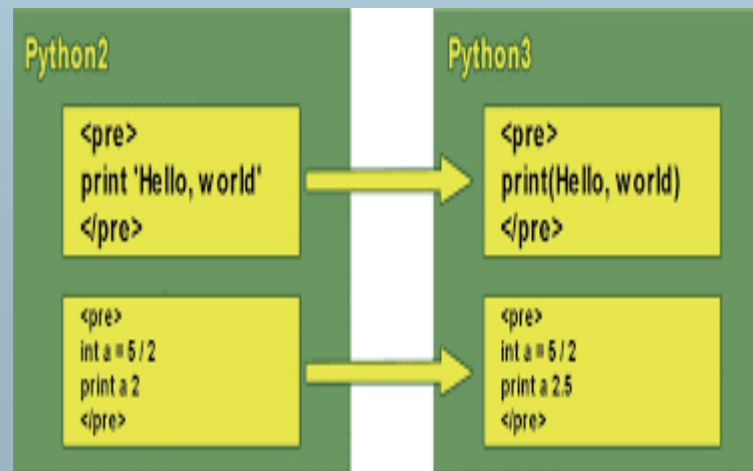
Manutenção Preventiva (cont.)

- As empresas montaram então grupos de trabalho para realizar manutenções em seus sistemas e converter todas as datas para o formato DD-MM-AAAA.
- Como essas atividades foram realizadas antes da virada do milênio, elas representam um exemplo de Manutenção Preventiva.



Manutenção Adaptativa

- Manutenção Adaptativa tem como objetivo adaptar um sistema a uma mudança em seu ambiente, incluindo tecnologia, legislação, regras de integração com outros sistemas ou demandas de novos clientes.
- Exemplo de Manutenção Adaptativa:
 - A migração de um sistema de Python 2.7 para Python 3.0.
 - A adaptação de um sistema para atender a uma mudança de legislação ou outra mudança contextual.



Manutenção Evolutiva

- Manutenção Evolutiva é aquela realizada para incluir uma nova funcionalidade ou introduzir aperfeiçoamentos importantes em funcionalidades existentes.
 - Sistemas de software podem ser usados por décadas exatamente porque eles sofrem manutenções evolutivas, que preservam o seu valor para os clientes.



Manutenção Evolutiva

- Exemplo de Manutenção Evolutiva:
 - Sistemas bancários ainda utilizados atualmente (Sistemas Legados), os quais foram criados nas décadas de 70 e 80, em linguagens como COBOL.
 - No entanto, eles já sofreram diversas evoluções e melhorias.
 - Hoje, esses sistemas possuem interfaces Web e para celulares, que se integram aos módulos principais, implementados há dezenas de anos.



Sistemas Legados

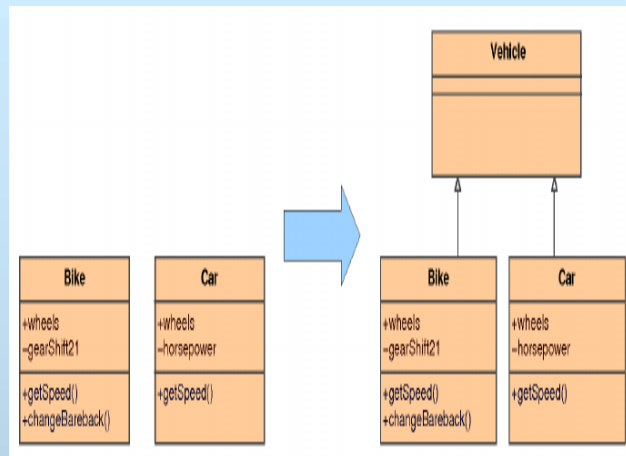
- Sistemas Legados são sistemas antigos, baseados em linguagens, sistemas operacionais e bancos de dados tecnologicamente ultrapassados (obsoletos).
 - Por esse motivo, a manutenção desses sistemas costuma ser mais custosa e arriscada.
 - Porém, é importante ressaltar que legado não significa irrelevante, pois muitas vezes esses sistemas realizam operações críticas para seus clientes.



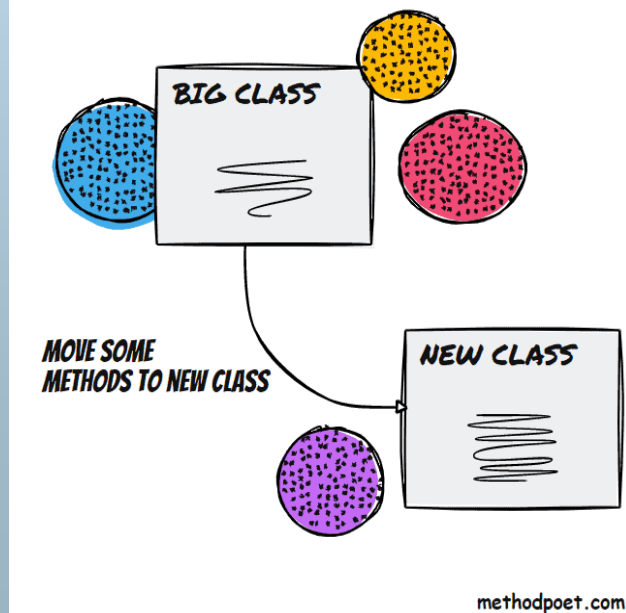
Refactorings

- *Refactorings* (Refatoramentos) são modificações realizadas em um software, preservando seu comportamento e visando exclusivamente a melhoria de seu código ou projeto (*design*).
- Exemplos de *Refactorings*:

- As operações como renomeação de um método ou de uma variável (para dar um nome mais intuitivo e fácil de lembrar).
- Divisão de um método longo em dois métodos menores (para facilitar o entendimento).
- Movimentação de um método para uma classe mais apropriada (aumentar a coesão).





EXTRACT CLASS



Gerência de Configuração

- Atualmente, é inconcebível desenvolver um software sem um sistema de controle de versões, como *git* ou *GitHub*.
 - Esses sistemas armazenam todas as versões de um software, não só do código fonte, mas também de documentação, manuais, páginas web, relatórios, etc.
 - Eles também permitem restaurar uma determinada versão, ou seja, se foi realizada uma mudança no código que introduziu um *bug* crítico, pode-se com relativa facilidade recuperar e retornar para a versão antiga, anterior à introdução do *bug*.

 git	 GitHub
1. It is a software	1. It is a service
2. It is installed locally on the system	2. It is hosted on Web
3. It is a command line tool	3. It provides a graphical interface
4. It is a tool to manage different versions of edits, made to files in a git repository	4. It is a space to upload a copy of the Git repository
5. It provides functionalities like Version Control System Source Code Management	5. It provides functionalities of Git like VCS, Source Code Management as well as adding few of its own features

Gerência de Configuração (cont.)

- No entanto, Gerência de Configuração é mais do que apenas usar um sistema como *git*.
 - Ela inclui, por exemplo, a definição de um conjunto de políticas para gerenciar as diversas versões de um sistema e seis itens de configuração.
- Ela preocupa-se com o esquema usado para identificar as *releases* (liberações) de um software, ou seja, as versões de um sistema que serão liberadas para seus clientes finais.
- Exemplo:
 - Um time de desenvolvedores pode definir que as *releases* de uma determinada biblioteca que eles estão desenvolvendo serão identificadas no formato x.y.z, onde x, y e z são inteiros.

Gerência de Configuração (cont.)

- Um incremento em *z* ocorre quando se lança uma nova *release* com apenas correções de *bugs* (chamada de *patch*).
 - Um incremento em *y* ocorre quando se lança uma *release* da biblioteca com pequenas funcionalidades (chamada de versão *minor*).
 - Um incremento em *x* ocorre quando se lança uma *release* com funcionalidades muito diferentes daquelas da última *release* (chamada de versão *major*).
- Esse esquema de numeração de releases é conhecido como Versionamento Semântico.



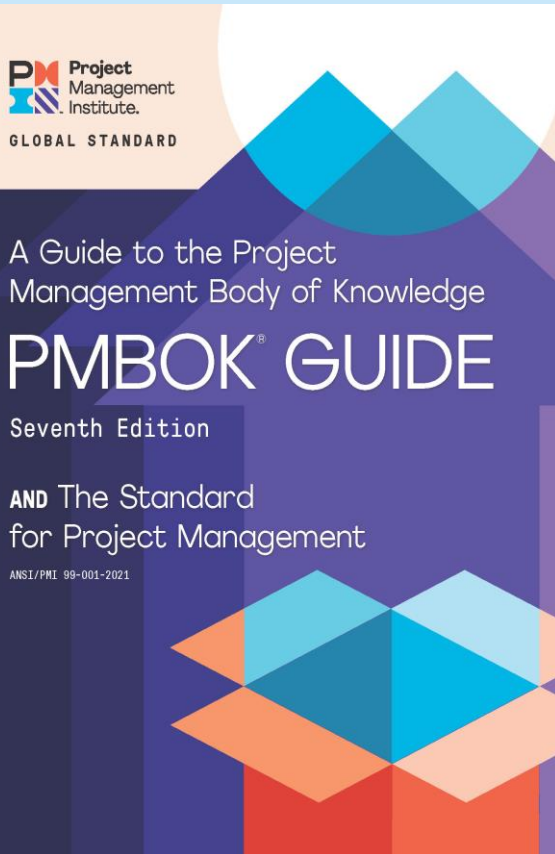
Gerência de Projeto

- Um projeto é um esforço único, temporário e progressivo empreendido para criar um produto, serviço ou resultado exclusivo.
 - Um projeto tem início e fim determinados (é, portanto, temporário), e um objetivo final.
 - Além disso, os recursos de um projeto não são ilimitados e a gestão deles deve ser planejada previamente.



Gerência de Projeto (cont.)

- Segundo o Guia PMBOK, o sucesso de um projeto é medido pela qualidade do produto e do projeto, pela pontualidade, pelo cumprimento do orçamento e pelo grau de satisfação do cliente.
- Exemplos de projetos:
 - Desenvolver um novo produto, serviço ou resultado.
 - Efetuar uma mudança na estrutura, envolvendo pessoas e/ou processos.
 - Adquirir, modificar ou desenvolver um sistema.
 - Realizar uma pesquisa cujo resultado será divulgado.
 - Construir um prédio, planta industrial ou infraestrutura.



Gerência de Projeto (cont.)

- A principal diferença entre projetos e processos é que, enquanto um processo é contínuo e repetido inúmeras vezes, os projetos são temporários e realizados uma única vez.
 - Mesmo que alguns procedimentos sejam repetidos no projeto, a repetição não muda o fato de que as suas entregas sejam únicas.
- Um exemplo de processo poderia ser o processo de vendas de uma empresa.
 - Há uma série de etapas que os vendedores seguem para realizar as vendas, e trata-se de um ciclo contínuo.
- Um exemplo de projeto seria a construção de um prédio.
 - Por mais que a equipe de construção possa repetir os mesmos processos de trabalho ao longo da construção, o projeto termina quando o objetivo for atingido (prédio construído), e cada novo prédio será um plano diferente.

Gerência de Projeto (cont.)

PROJETO



Tempo determinado



Resultado exclusivo



Trabalho inédito

PROCESSO



Tempo indeterminado



Resultado padrão



Trabalho repetitivo

Vs.

Gerência de Projeto (cont.)

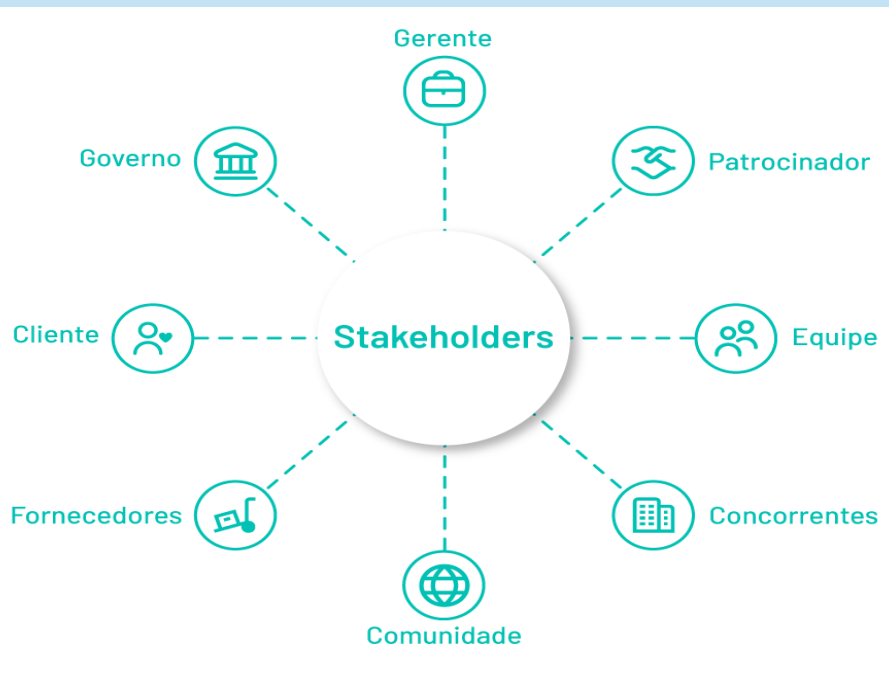
- Desenvolvimento de Software é um Projeto de Desenvolvimento de Software (PDS) e requer o uso de práticas e atividades de gerência de projetos para:



- Negociação de contratos com clientes (com definição de prazos, valores, cronogramas, etc.).
- Gerência de recursos humanos (incluindo contratação, treinamento, políticas de promoção, remuneração, etc.)
- Gerência de riscos.
- Acompanhamento da concorrência.
- Marketing.
- Finanças.
- etc.

Gerência de Projeto (cont.)

- Em um projeto, normalmente usa-se o termo *stakeholder* para designar todas as partes interessadas no mesmo, ou seja, os *stakeholders* são aqueles que afetam ou que são afetados pelo projeto, podendo ser pessoas físicas ou organizações.



- Exemplo:
 - *Stakeholders* comuns em projetos de software incluem, seus desenvolvedores, clientes, gerentes da equipe de desenvolvimento, empresas subcontratadas, fornecedores de qualquer natureza, talvez algum nível de governo, etc.

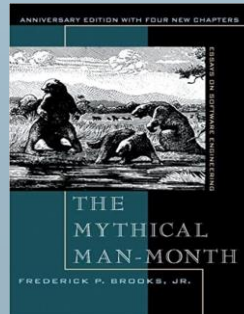
Gerência de Projeto (cont.)

- Existe uma frase muito conhecida, também de Frederick Brooks, que captura uma peculiaridade de projetos de desenvolvimento de software.
 - Segundo Brooks:



"A inclusão de novos desenvolvedores em um projeto que está atrasado contribui para torná-lo ainda mais atrasado."

- Essa frase ficou tão famosa que hoje é conhecida como Lei de Brooks.



Gerência de Projeto (cont.)

"A inclusão de novos desenvolvedores em um projeto que está atrasado contribui para torná-lo ainda mais atrasado."

- Esse efeito acontece porque os novos desenvolvedores terão primeiro que entender e compreender todo o sistema, seus requisitos, sua arquitetura e codificação, antes de começarem a produzir código útil.
- Além disso, equipes maiores exigem um maior esforço de comunicação e coordenação para tomar e explicar decisões.
- Exemplo:
 - ❖ Se um time tem 3 desenvolvedores (d_1, d_2, d_3), existem 3 canais de comunicação possíveis (d_1-d_2, d_1-d_3 e d_2-d_3); se o time aumenta para 4 desenvolvedores, o número de canais duplica, para 6 canais.

Gerência de Projeto (cont.)

"A inclusão de novos desenvolvedores em um projeto que está atrasado contribui para torná-lo ainda mais atrasado."

- ❖ Se o time aumenta para 10 desenvolvedores, passam a existir 45 canais de comunicação.
- Por isso, modernamente, software é desenvolvido em times pequenos, em média com uma dezena de engenheiros de software, no máximo.



Projeto x *Design*

- Em Inglês, há as duas palavras:
 - *Project*, que representa um esforço colaborativo para resolver problemas; e
 - *Design*, que significa desenho ou proposta de uma solução.
- Em Português, se tem uma única palavra:
 - Projeto.
- Neste curso, Projeto = *Design*.
 - O termo Projeto de Software (*Design*) refere-se à estruturação (desenho) da solução proposta.
 - *Design* = Arquitetura do Software.
- Quando se referenciar Projeto de Software, sem ser *Design*, será formalizado por Projeto de Desenvolvimento de Software (PDS).

Processo de Desenvolvimento de Software

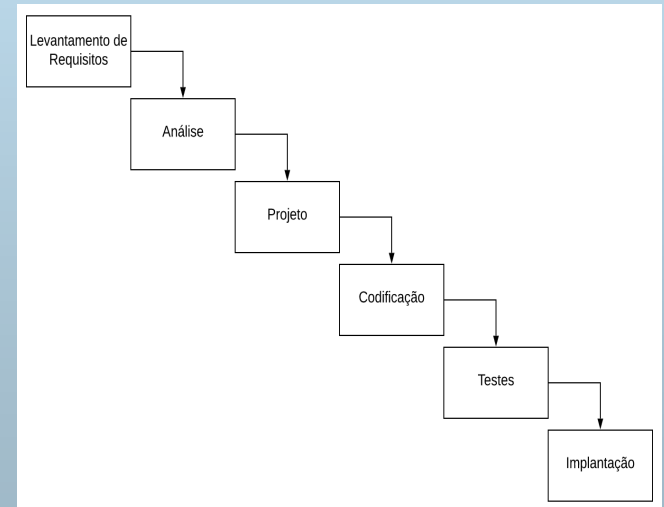
- Um Processo de Desenvolvimento de Software define quais atividades e etapas devem ser seguidas para construir e entregar um sistema de software.
 - Uma analogia pode ser feita com a construção de prédios, que ocorre de acordo com algumas etapas: fundação, alvenaria, cobertura, instalações hidráulicas, instalações elétricas, acabamento, pintura, etc.
- Historicamente, existem dois grandes tipos de processos que podem ser adotados na construção de sistemas de software:
 - Processos *Waterfall* (ou em Cascata ou Tradicionais)
 - Processos Ágeis (ou Incrementais e/ou Iterativos).

Processo de Desenvolvimento de Software (cont.)

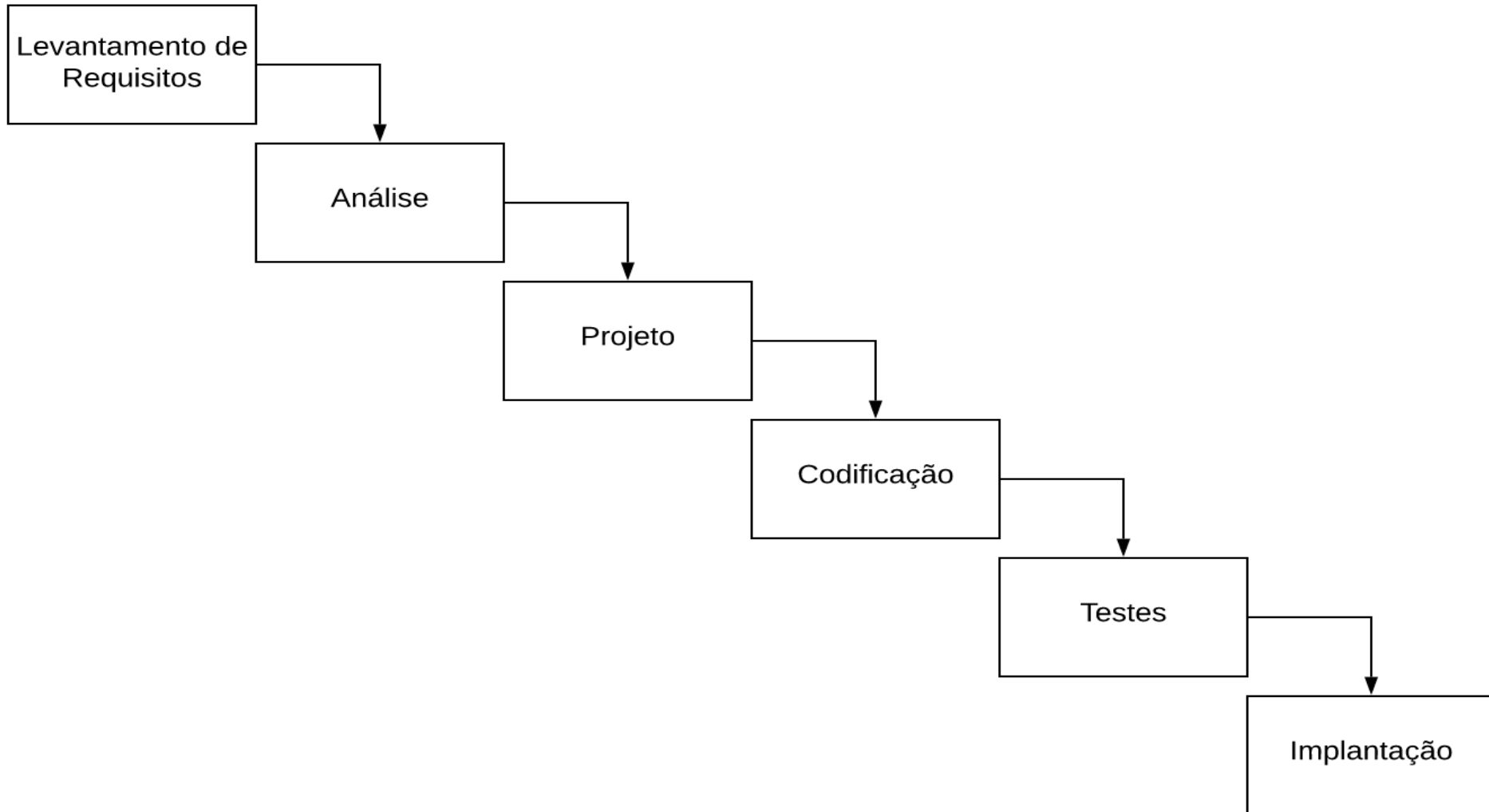
- Processos *Waterfall* foram os primeiros a serem propostos, ainda na década de 70, quando a Engenharia de Software começava a ganhar envergadura.
 - De forma compreensível, eles foram inspirados nos processos usados em engenharias tradicionais, os quais são largamente sequenciais, como ilustrado no exemplo da construção do prédio.
- Processos *Waterfall* foram muito usados até a década de 1990 e grande parte desse sucesso deve-se a uma padronização lançada pelo Departamento de Defesa Norte-Americano (DoD), em 1985.
 - Basicamente, eles estabeleceram que todo software comprado ou contratado pelo DoD deveria ser construído usando *Waterfall*.

Processo de Desenvolvimento de Software (cont.)

- Processos *Waterfall* (também chamados de Processos Dirigidos por Planejamento - *plan-driven process*) propõem que a construção de um sistema deve ser feita em etapas sequenciais, como em uma cascata de água, na qual a água vai escorrendo de um nível para o outro.
- Essas etapas são as seguintes:
 - Levantamento de Requisitos
 - Análise (ou projeto de alto nível)
 - Projeto Detalhado
 - Codificação e
 - Testes e Implantação.
- Finalizado esse *pipeline*, o sistema é liberado para produção, isto é, para uso efetivo pelos seus usuários.



Processo de Desenvolvimento de Software (cont.)



Processo de Desenvolvimento de Software (cont.)

- No entanto, os Processos *Waterfall*, a partir do final da década de 90, passaram a ser muito criticados, devido aos atrasos e problemas recorrentes em projetos de software, que ocorriam com frequência nessa época.
 - O principal problema é que este tipo de processo pressupõe um levantamento completo de requisitos, depois um projeto detalhado, depois uma implementação completa, etc.
 - Para só então validar o sistema com os usuários, o que pode acontecer anos após o início do projeto.
 - No entanto, nesse período, o mundo pode ter mudado, bem como as necessidades dos clientes, que podem não mais precisar do sistema que ajudaram a especificar anos antes.

Processo de Desenvolvimento de Software (cont.)

Manifesto for Agile Software Development

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.

- Reunidos em uma cidade de Utah, Estados Unidos, em fevereiro de 2001, um grupo de 17 Engenheiros de Software propôs um modo alternativo para construção de software, que eles chamaram de Ágil (nome do manifesto que eles produziram nessa reunião)

Contrastando com Processos Tradicionais, a ideia de Processos Ágeis é que um sistema deve ser construído de forma incremental e iterativa.

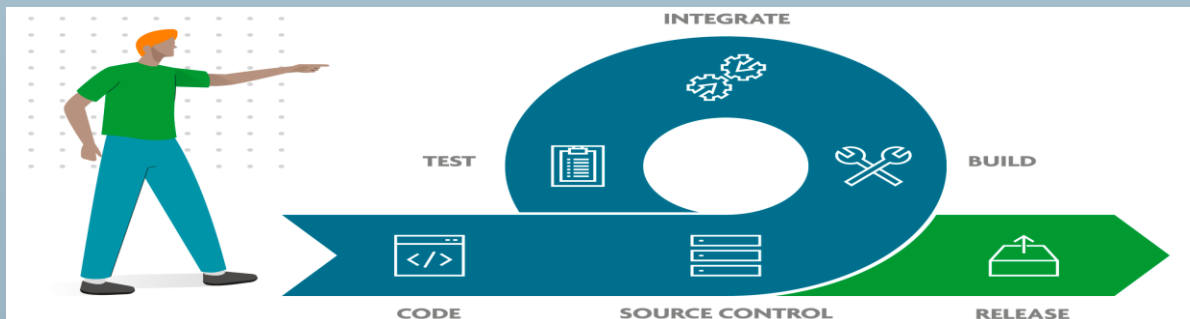
- Pequenos incrementos de funcionalidade são produzidos, em intervalos de cerca de um mês e, logo em seguida, validados pelos usuários.
- Uma vez que o incremento produzido seja aprovado, o ciclo se repete.

Processo de Desenvolvimento de Software (cont.)

- Processos Ágeis tiveram um profundo impacto na indústria de software.
 - Hoje, eles são usados pelas mais diferentes organizações que produzem software, desde pequenas empresas até as grandes companhias da Internet.
 - Diversos métodos que concretizam os princípios ágeis foram propostos, tais como XP, Scrum, Kanban e *Lean Development*.
- Esses métodos também ajudaram a disseminar diversas boas práticas de desenvolvimento de software, como
 - Testes automatizados.
 - *Test-driven development* (isto é, escrever os testes primeiro, antes do próprio código).
 - Integração Contínua (*Continuous Integration*).

Processo de Desenvolvimento de Software (cont.)

- Integração Contínua recomenda que desenvolvedores integrem o código que produzem imediatamente, se possível todo dia.
 - O objetivo é evitar que desenvolvedores fiquem muito tempo trabalhando localmente, em sua máquina, sem integrar o código que estão produzindo no repositório principal do projeto.
 - Quando o time de desenvolvimento é maior, isso aumenta as chances de conflitos de integração, que ocorrem quando dois desenvolvedores alteram em paralelo os mesmos trechos de código.

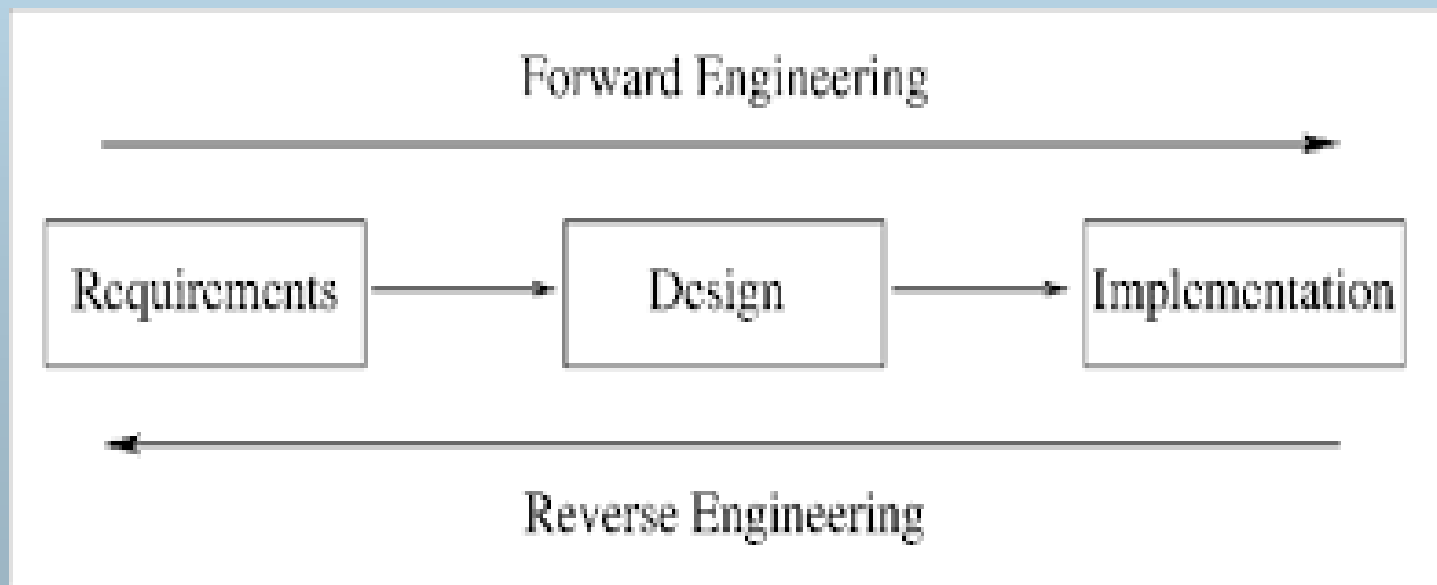


Modelos de Software

- Um modelo oferece uma representação em mais alto nível de um sistema do que o seu código fonte.
 - O objetivo é permitir que desenvolvedores possam analisar propriedades e características essenciais de um sistema, de modo mais fácil e rápido, sem ter que mergulhar nos detalhes do código.
- Modelos podem ser criados antes do código, por exemplo, ainda na fase de projeto e nesse caso, eles são usados para apoiar a Engenharia Direta (*Forward Engineering*).
 - Primeiro cria-se um modelo para ter um entendimento de mais alto nível de um sistema, antes de partir para a implementação do código.

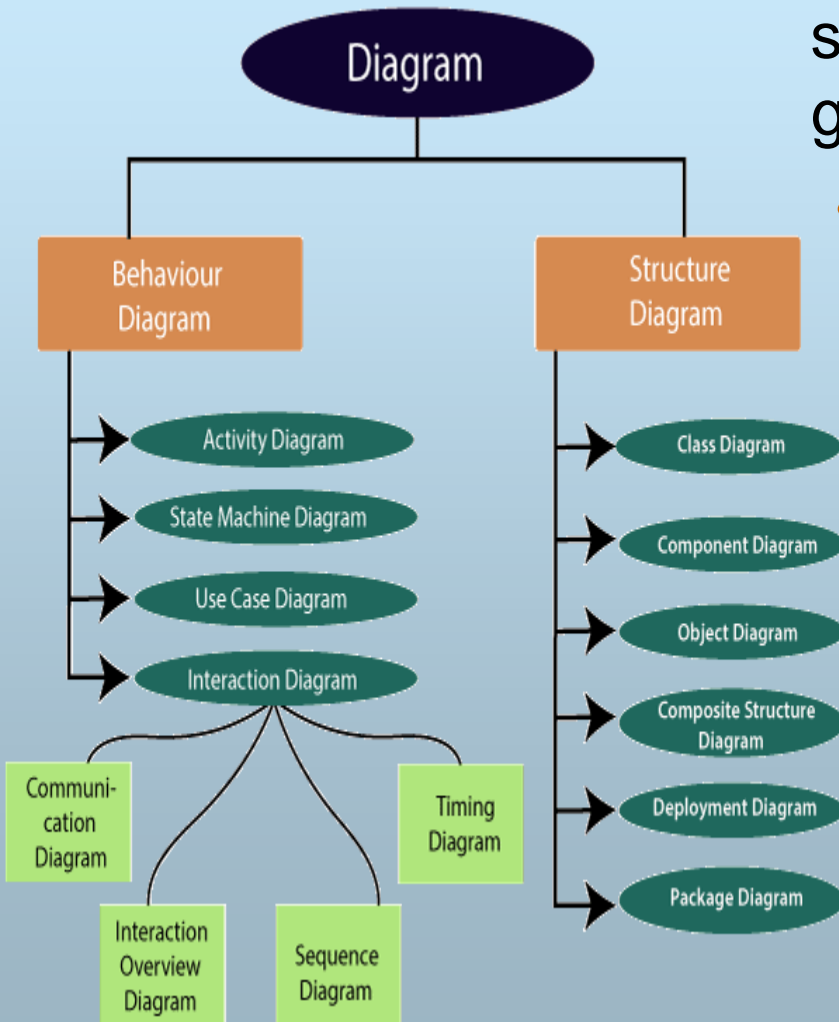
Modelos de Software (cont.)

- Por outro lado, eles podem ser criados para ajudar a entender um código existente e nesse caso, eles são um instrumento de Engenharia Reversa (*Reverse Engineering*).
 - Em ambos os casos, modelos são uma forma de documentar o código de um sistema.



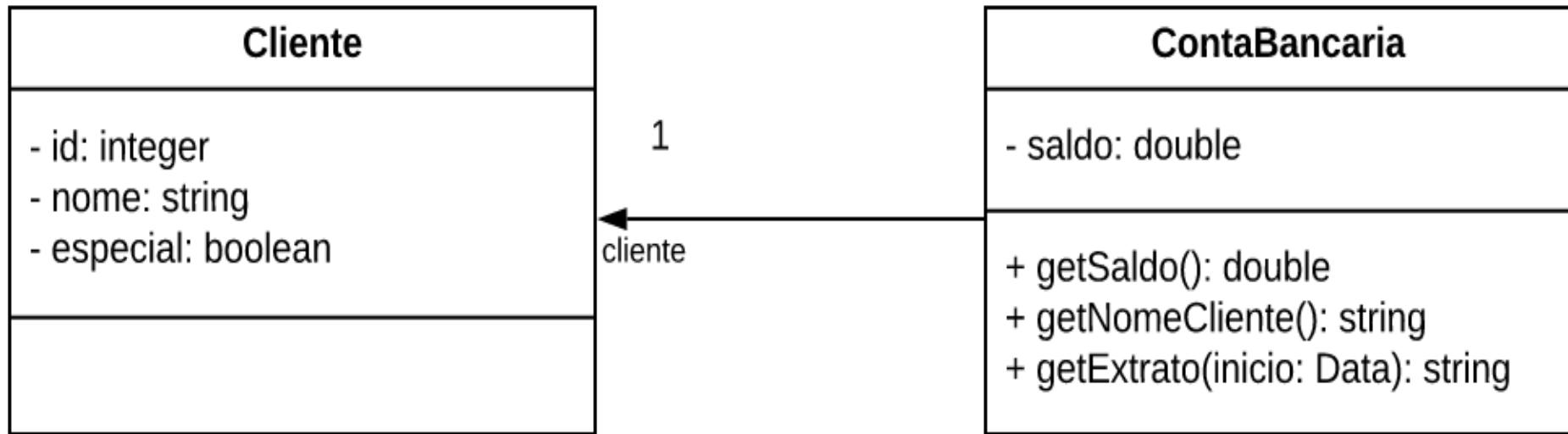
Modelos de Software (cont.)

- Frequentemente, modelos de software são baseados em notações gráficas.



- A UML (*Unified Modelling Language*) é uma linguagem de modelagem que define mais de uma dezena de diagramas gráficos para representar propriedades funcionais, estruturais e comportamentais de um sistema.

Modelos de Software (cont.)



- Na figura acima, mostra-se um diagrama da UML, o Diagrama de Classes, com duas classes.
- Nesse diagrama, as caixas retangulares representam classes do sistema, incluindo seus atributos e métodos.
- As setas são usadas para denotar relacionamentos entre as classes.
- Existem editores para criar diagramas UML, que podem ser usados, em um cenário de Engenharia Direta.

Qualidade de Software

- Qualidade é um objetivo recorrente em produtos de engenharia.
 - Fabricantes de automóveis, celulares, computadores, empresas de construção civil, etc. todos almejam e dizem que possuem produtos de qualidade.
- Esse contexto não é diferente quando o produto é um software.
 - Segundo uma classificação proposta por Bertrand Meyer, a qualidade de software pode ser avaliada em duas dimensões:
 - ❖ Qualidade Externa.
 - ❖ Qualidade Interna.

Qualidade de Software (cont.)

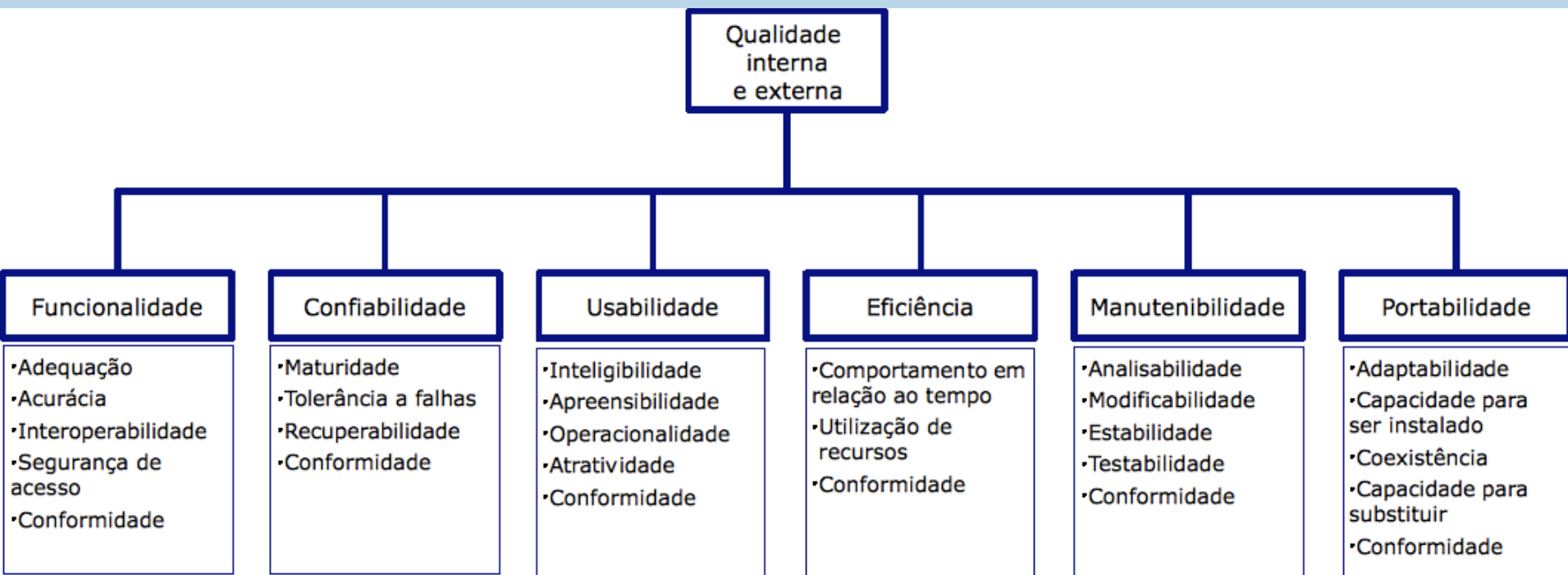
- Qualidade Externa considera os fatores que podem ser aferidos sem analisar o código e pode ser avaliada mesmo por usuários comuns, que não são especialistas em Engenharia de Software.
- Exemplo de fatores (ou atributos) de qualidade externa:
 - Correção - o software atende à sua especificação?
 - ❖ Por exemplo, nas situações normais, ele funciona como esperado?
 - Robustez - o software continua funcionando mesmo quando ocorrem eventos anormais, como uma falha de comunicação ou de disco?
 - ❖ Por exemplo, um software robusto não pode sofrer um *crash* (abortar) caso tais eventos anormais ocorram. Ele deve pelo menos avisar por qual motivo não está conseguindo funcionar conforme previsto.

Qualidade de Software (cont.)

- Eficiência - o software faz bom uso de recursos computacionais?
 - ❖ Por exemplo, ele precisa de um hardware extremamente poderoso e caro para funcionar?
- Portabilidade - é possível portar esse software para outras plataformas e sistemas operacionais?
 - ❖ Por exemplo, ele possui versões para os principais sistemas operacionais, como Windows, Linux e macOS?
 - ❖ Por exemplo, se for um app, ele possui versões para Android e iOS?
- Facilidade de Uso - o software possui uma interface amigável, mensagens de erro claras, suporta mais de uma língua, etc.
 - ❖ Por exemplo, pode ser também usado por pessoas com alguma deficiência, como visual ou auditiva?

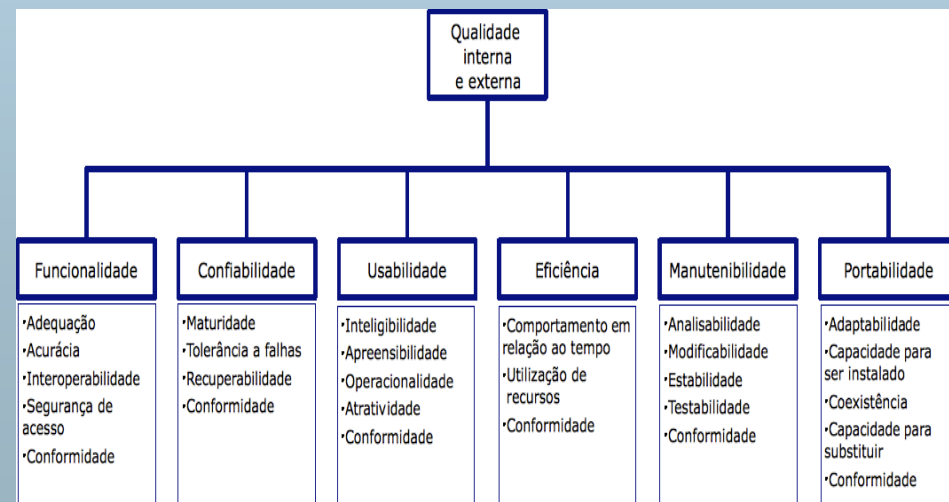
Qualidade de Software (cont.)

- Compatibilidade - o software é compatível com os principais formatos de dados de sua área?
 - ❖ Por exemplo, se o software for uma planilha eletrônica, ele importa arquivos em formatos XLS e CSV?

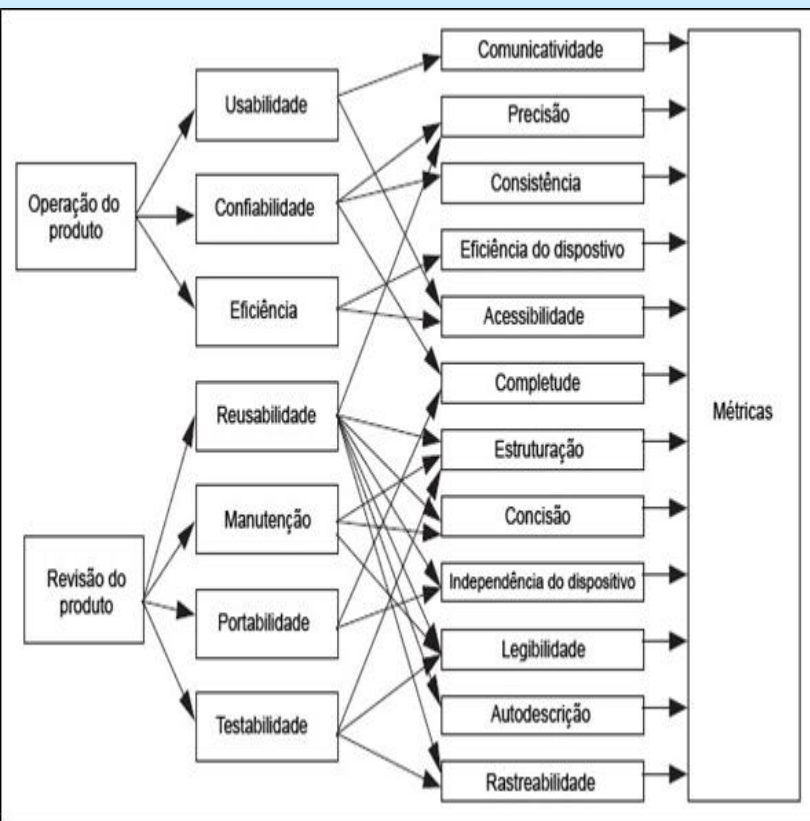


Qualidade de Software (cont.)

- A Qualidade Interna considera propriedades e características relacionadas com a implementação de um sistema.
 - Portanto, a qualidade interna de um sistema somente pode ser avaliada por um especialista em Engenharia de Software e não por usuários leigos.
- Exemplos de fatores (ou atributos) de qualidade interna:
 - Modularidade.
 - Legibilidade do código.
 - Manutenibilidade.
 - Testabilidade.
 - ...



Qualidade de Software (cont.)



- Para garantir a qualidade de software, diversas estratégias podem ser usadas.

- Métricas podem ser usadas para acompanhar o desenvolvimento de um produto de software, incluindo métricas de código fonte e métricas de processo.

- ❖ Um exemplo de métrica de código é o número de linhas de um programa, utilizado para dar uma ideia de seu tamanho.

- Métricas de processo incluem, por exemplo, o número de defeitos reportados em produção por usuários finais em um certo intervalo de tempo.

- O *Institute of Electrical and Electronic Engineers* (IEEE) define métricas como “uma medida quantitativa do grau em que um sistema, componente ou processo possui um determinado atributo”.
- O objetivo das métricas de software é controlar e identificar eficientemente os parâmetros essenciais que afetam o desenvolvimento de software.

Qualidade de Software (cont.)

- Existem ainda boas práticas que podem ser adotadas para garantir a produção de software com qualidade.
 - Modernamente, por exemplo, diversas organizações usam revisões de código, isto é, o código produzido por um desenvolvedor somente entra em produção depois de ser revisado e inspecionado por um outro desenvolvedor do time.
 - ❖ O objetivo é detectar *bugs* antecipadamente, antes de o sistema entrar em produção.
 - Além disso, revisões de código servem para garantir a qualidade interna do código (manutenibilidade, legibilidade, modularidade, etc.) e para disseminar estas boas práticas de Engenharia de Software entre os membros de um time de desenvolvimento.



Qualidade de Software (cont.)

■ Exemplo:

- Assumindo que a empresa que produziu esse código adotasse revisões de código, ele teria que ser analisado por um outro desenvolvedor, chamado de revisor, antes de entrar em produção.
- Esse revisor poderia perceber o *bug* e anotar o código com uma dúvida, antes de aprová-lo.
- Em seguida, o responsável pelo código poderia concordar que, de fato, existe um *bug*, corrigir o código e submetê-lo de novo para revisão.
- Finalmente, ele seria aprovado pelo revisor.

