Engenharia de Software
Implementação

Prof. Pablo Vargas

Tópicos Abordados

- Introdução
- Componentes
- Desenvolvimento baseado em Componentes
- Implementação
- Aspectos de Implementação
- Codificação
- Testes
- Documentação
- Exemplos
- Conclusão

Introdução

- A fase do desenvolvimento onde o **projeto/design é transformado em código executável**.
- Envolve a **escrita, compilação, e depuração** do **código** fonte.
- **Transformar** ideias e planos em um **produto funcional**.
- Influência direta na manutenção e evolução do software.

- Objetivos principais da fase de implementação:
  - Transformar especificações em código.
  - Assegurar a qualidade do software.
  - Facilitar a manutenção futura.

Componentes

- ...é o termo utilizado para descrever o elemento de *software* que **encapsula uma série de funcionalidades**.
  - Um componente é uma unidade **independente**, que pode ser utilizado com outros componentes para formar um sistema mais complexo.
  - Um sistema de software pode ser formado inteiramente somente por componentes, pois estes **se interligam através de suas interfaces**.
    - Esse processo de comunicação entre componentes é denominado **composição**.

Componentes

- "uma **não-trivial**, **quase independente**, e **substituível** parte de um sistema que cumpre uma função clara no contexto de uma arquitetura bem definida“ (Brown e Wallnau, 1996)
- Para Szyperski (1997), não é necessariamente uma tecnologia implementada especificamente e nem a aplicação, mas sim um **dispositivo de software** que possua uma **interface bem definida**.

Componentes

- A **colagem de componentes** (glueing) serve para **viabilizar a operação conjunta** de componentes originalmente incompatíveis.
  - Ocorre a inclusão de um novo elemento, a cola (glue), **entre os componentes incompatíveis**, possibilitando sua operação conjunta.
- Modelos de Componentes:
  - CMM (CORBA Component Model) do OMG (Object Management Group).
  - DCOM (Distributed Component Obejct) e COM/COM+ (Component Object Model) da Microsoft.
  - JavaBeans e Entreprise JavaBeans (EJB) da Sun.

Desenvolvimento baseado em Componentes

- Engenharia de Software Baseada em Componentes (*Component Based Software Engineering*, CBSE)
- A ideia foi apresentada por Mcilory na Conferência de Engenharia de Software da OTAN de 1968.
  - Em 1976, DeRemer propôs um paradigma de desenvolvimento através de um conjunto de módulos.
  - Com o surgimento da OO (Anos 80) e a possibilidade de reutilização, fortaleceu a proposta de produzir componentes.
- Foco está na **decomposição dos sistemas**, em componentes funcionais e lógicos com **interfaces bem definidas**, usadas para comunicação entre os próprios componentes.
  - Comunicação por **troca de mensagens** contendo dados.
- São considerados como estando num **nível de abstração mais alto** que do que Objetos.

Desenvolvimento baseado em Componentes

- **Desenvolvimento para reuso**: desenvolvimento de componentes para serem reusados em outras aplicações/sistemas.
  - Geralmente, envolve a generalização dos componentes que existem.
- **Desenvolvimento com reuso**: utilização dos componentes existentes para o desenvolvimento de novas aplicações/sistemas.

Desenvolvimento baseado em Componentes

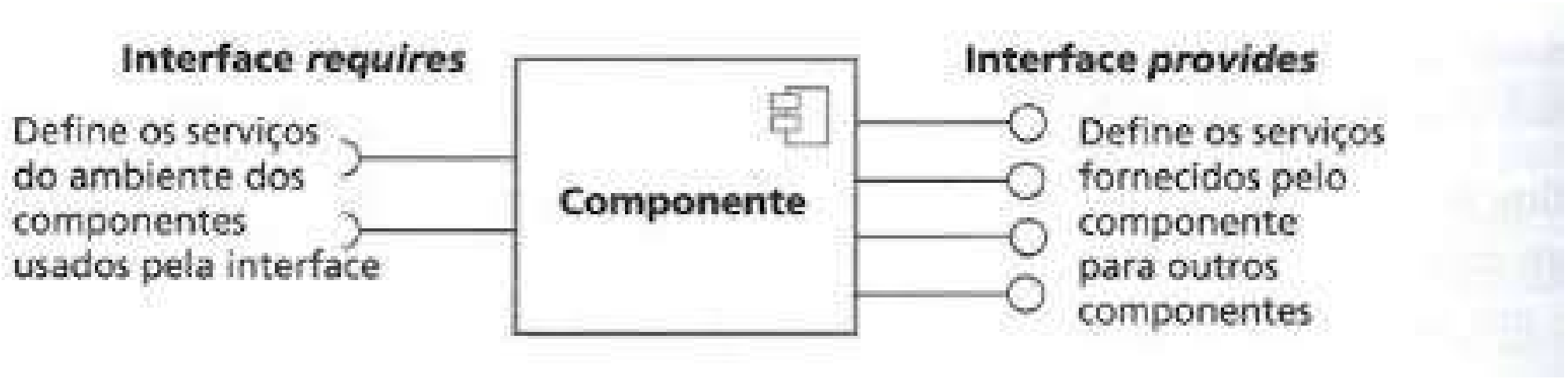
- **Características dos componentes:**
  - **Padronizado**: significa que um componente usado em um processo CBSE precisa obedecer a um modelo de componentes padrão.
    - Esse modelo pode definir as interfaces de componentes, metadados de componente, documentação, composição e implantação.
  - **Independente**: deve ser possível compor e implantá-lo sem precisar usar outros componentes específicos.
    - Quando o componente necessita de serviço externos, estes devem ser explicitamente definidos em uma especificação de interface "*requires*".

Desenvolvimento baseado em Componentes

- **Características dos componentes:**
  - **Passível de composição**: para um componente ser composto, todas as interações externas devem ter lugar por meio de interfaces publicamente definidas.
    - Além disso, ele deve proporcionar acesso externo a informações sobre si próprio, como seus métodos e atributos.
  - **Implantável**: deve ser capaz de operar como uma entidade autônoma em uma plataforma de componentes que forneça uma implementação do modelo de componentes.
  - **Documentando**: devem ser completamente documentados para que os potencias usuários possam decidir se satisfazem a suas necessidades.

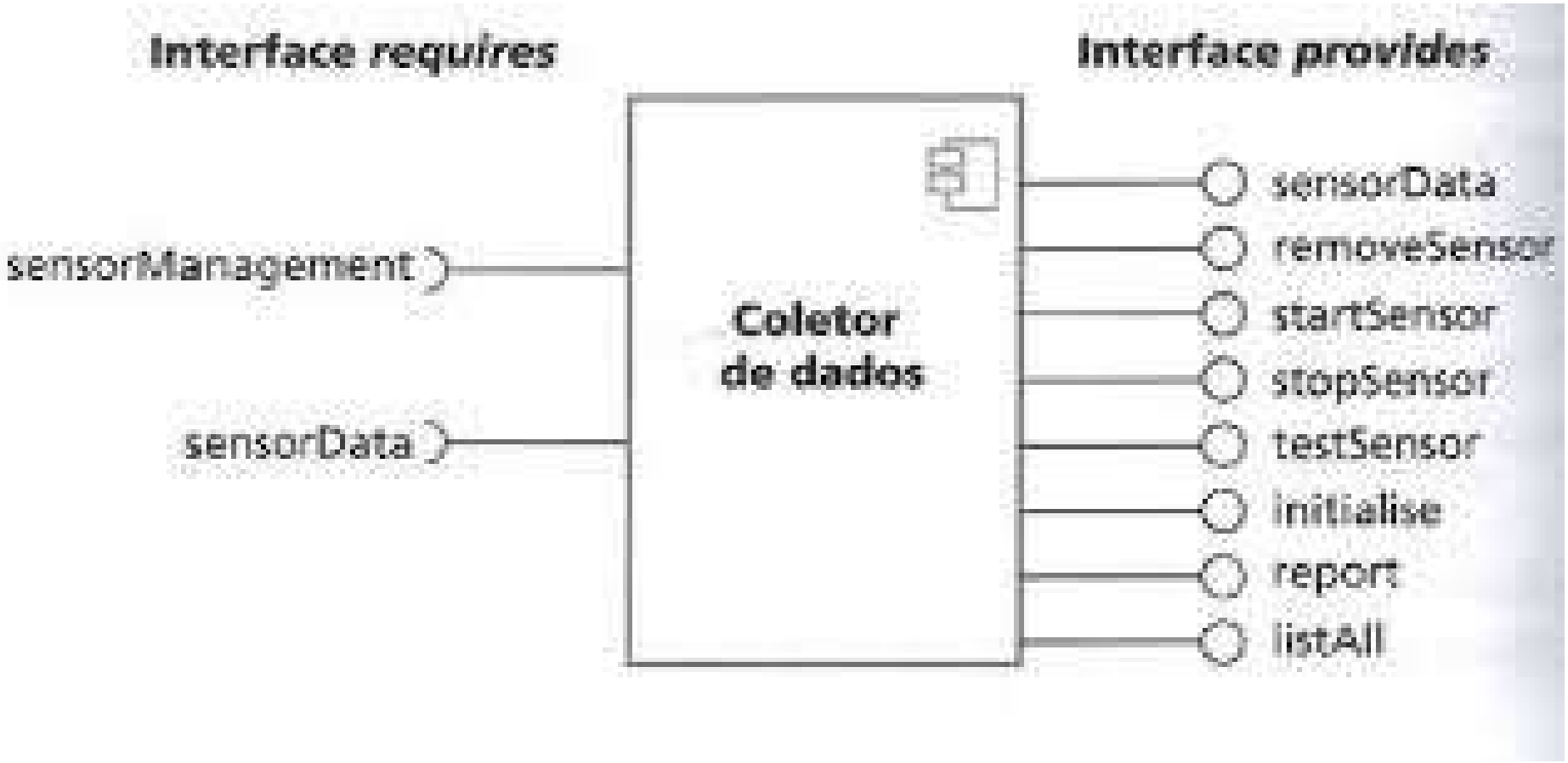
Desenvolvimento baseado em Componentes

- **Interfaces de componentes**



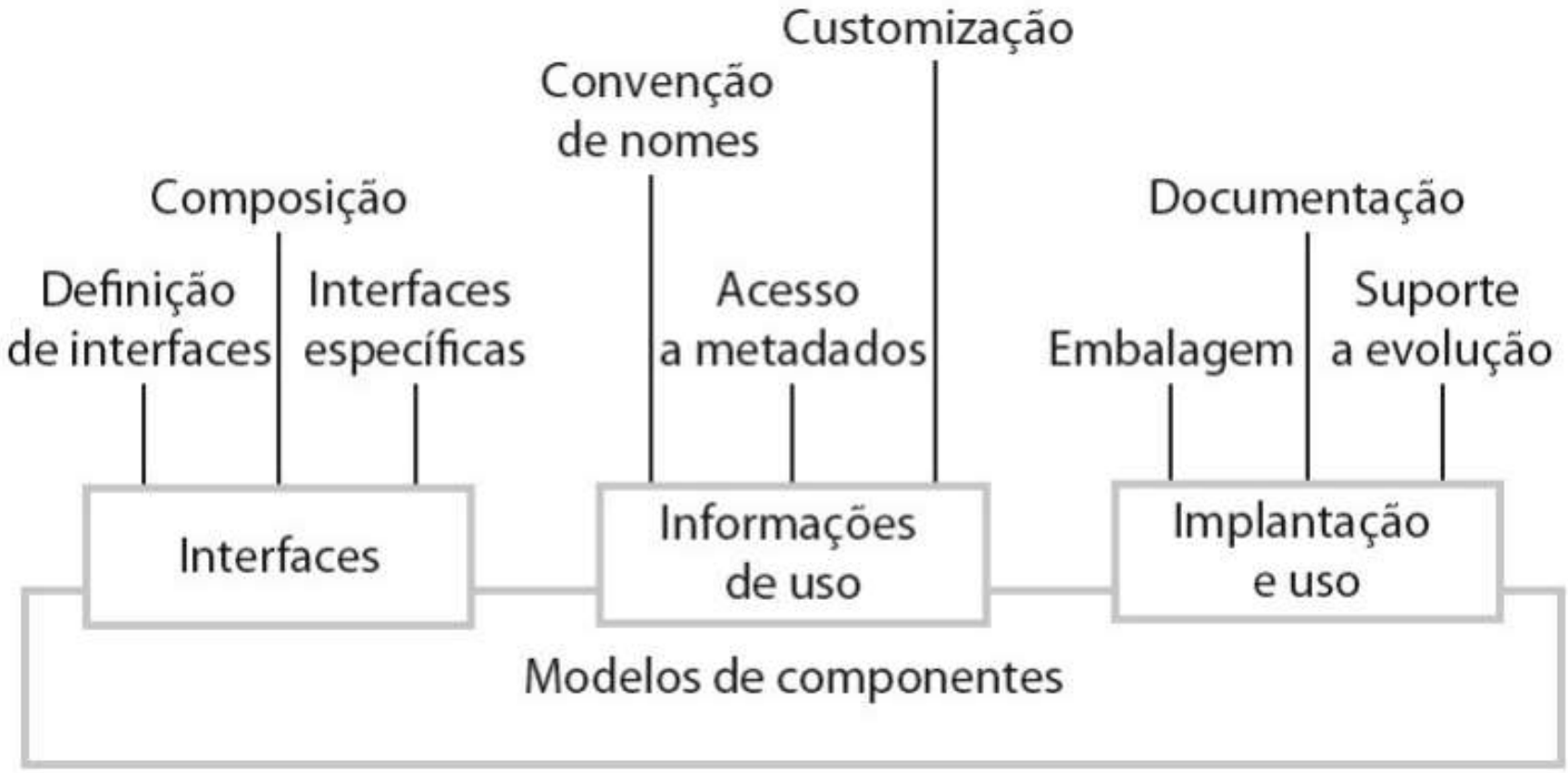
Desenvolvimento baseado em Componentes

- **Um modelo de componente sensor de dados**



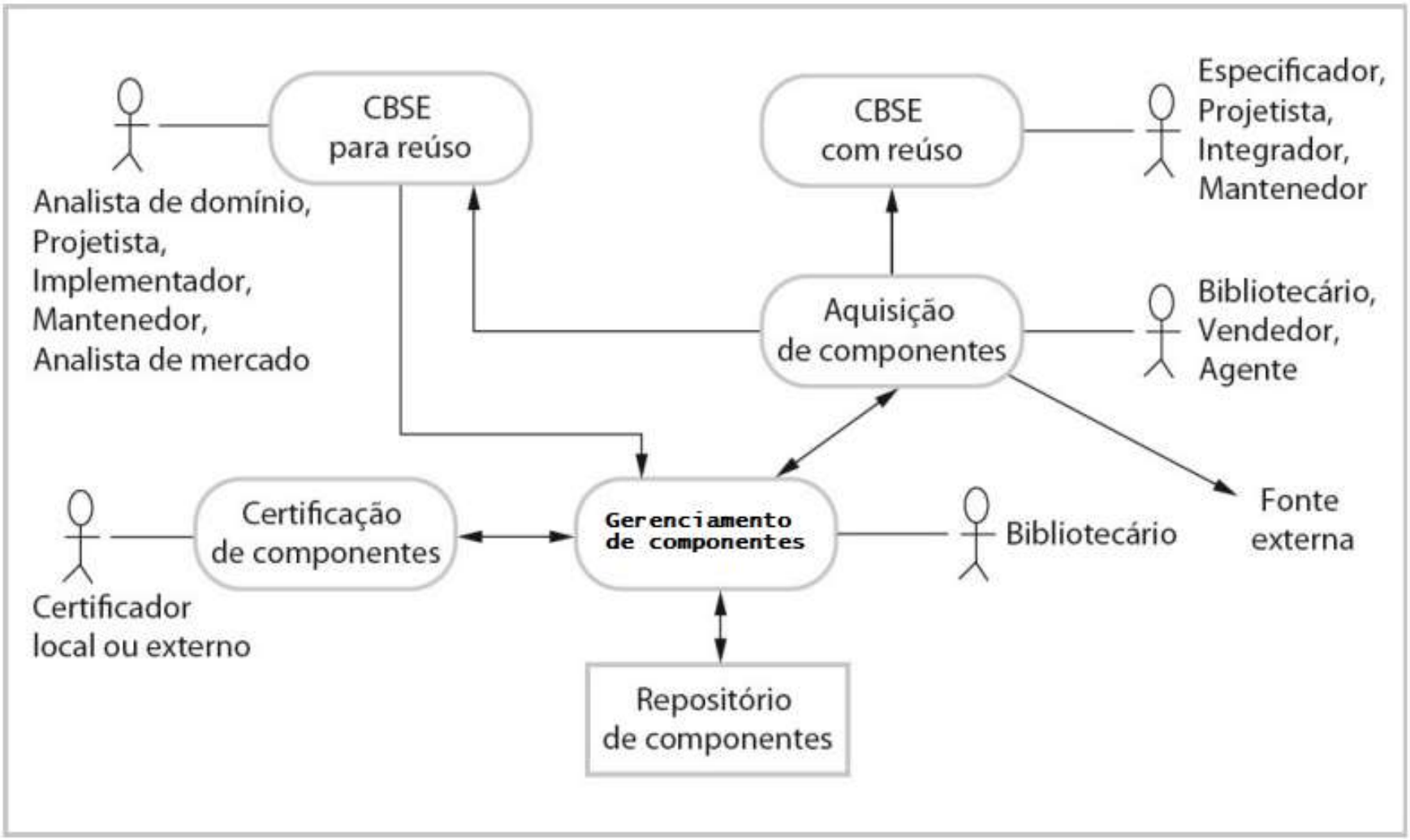
Desenvolvimento baseado em Componentes

- **Elementos básicos de um modelo de componentes**



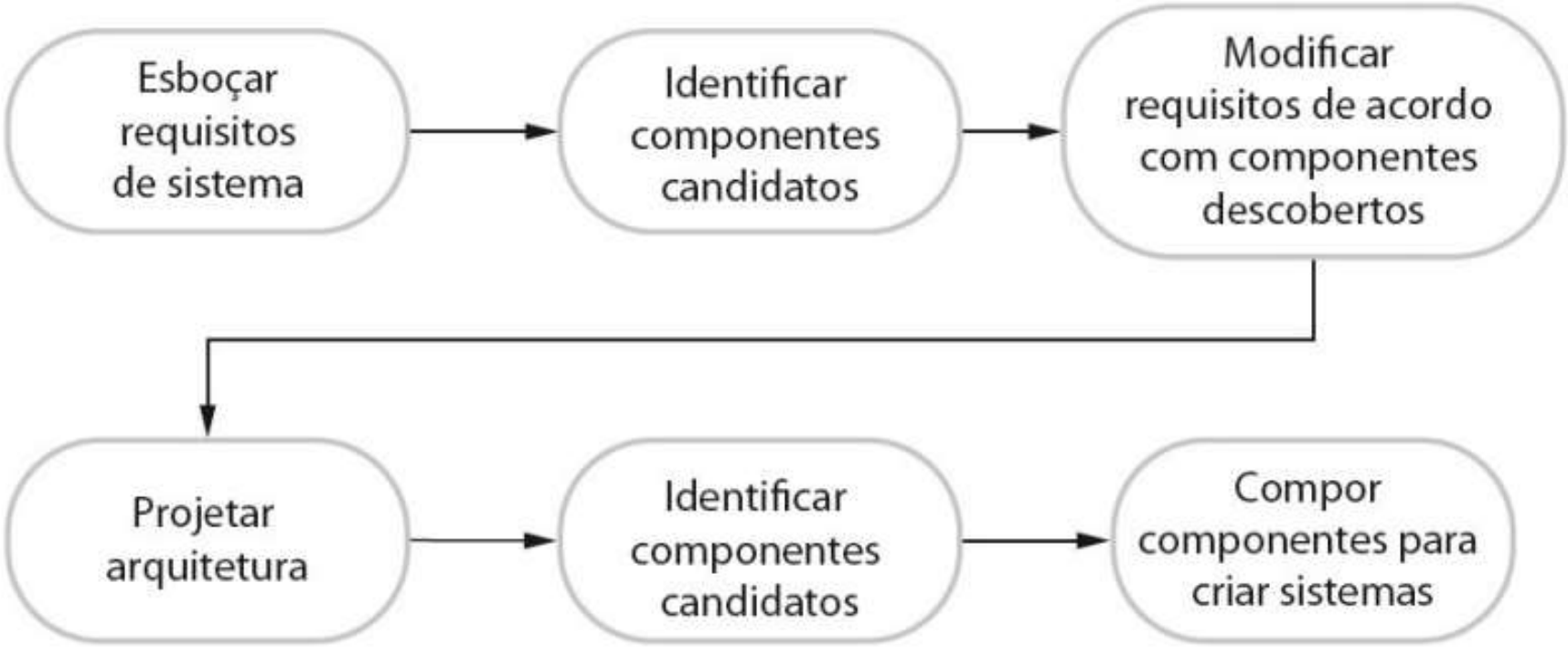
Desenvolvimento baseado em Componentes

- **Processos CBSE**



Desenvolvimento baseado em Componentes

- **CBSE com reúso**

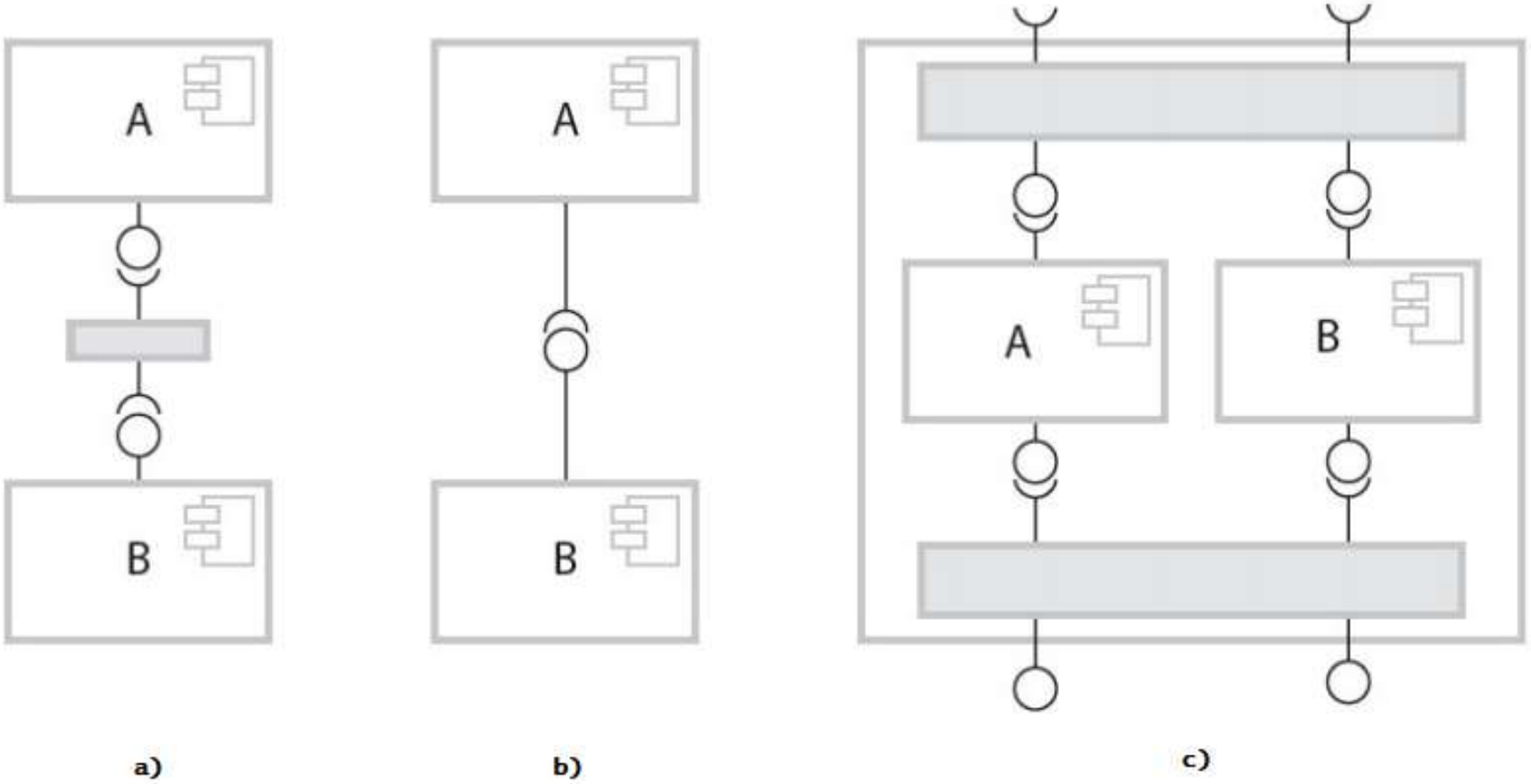


Desenvolvimento baseado em Componentes

- **Processo de identificação de componentes**

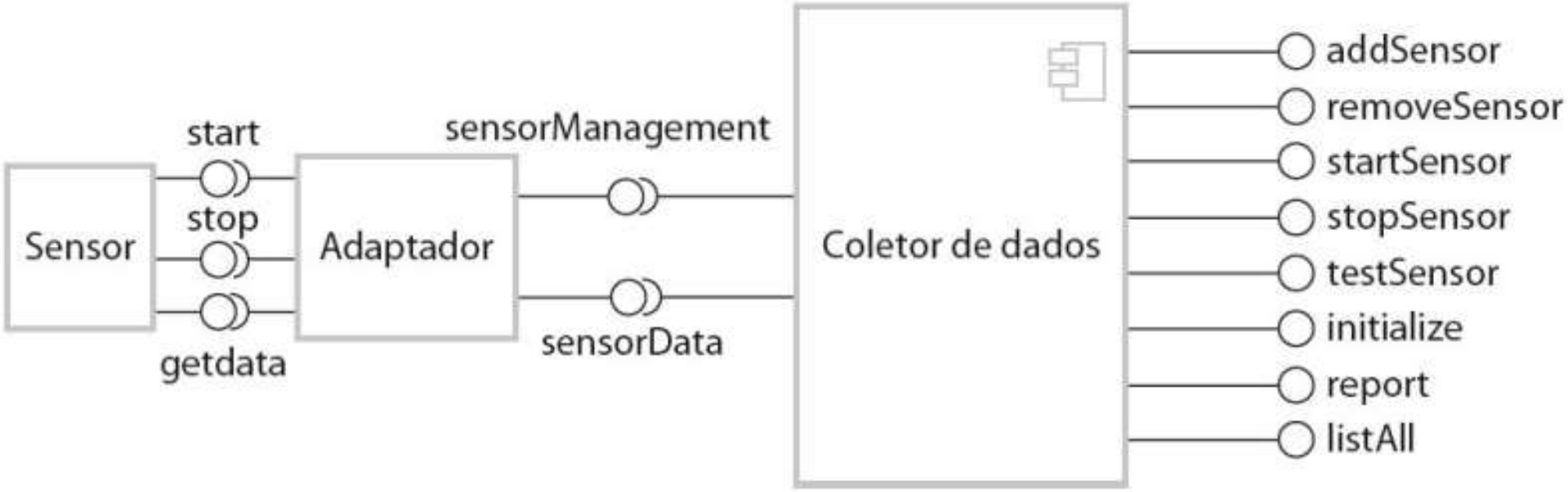
Desenvolvimento baseado em Componentes

- **Tipos de composição de componentes**



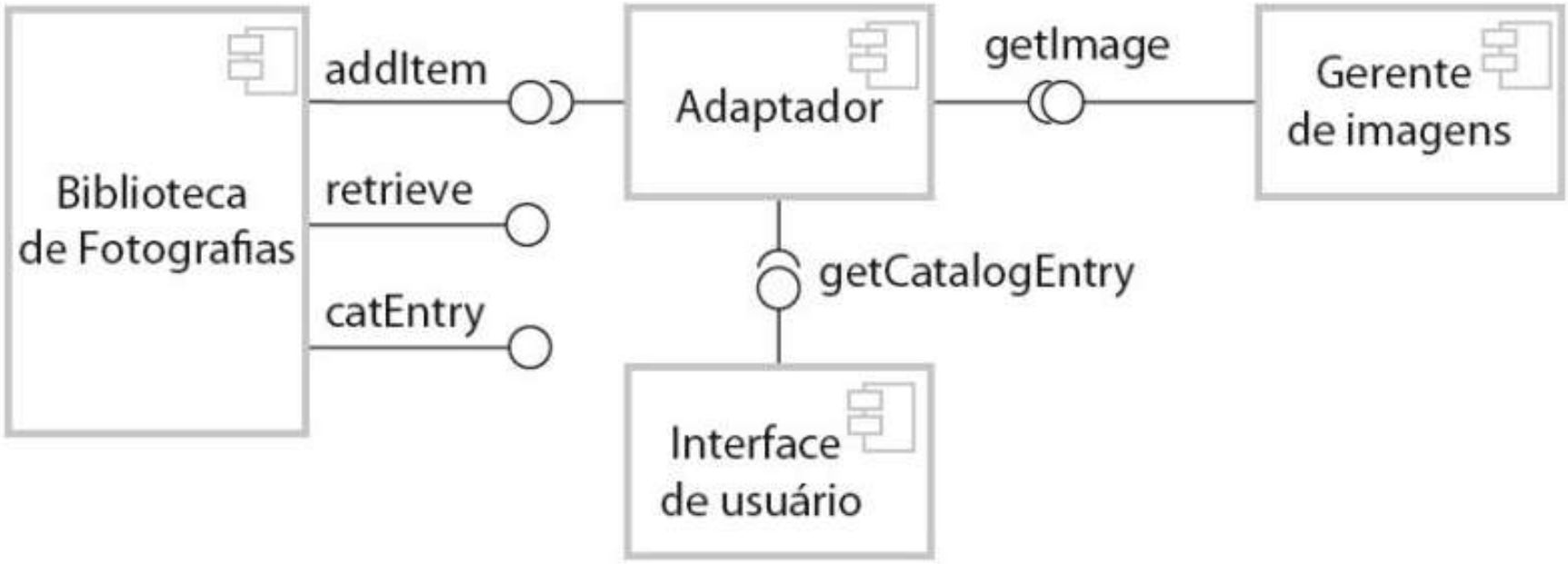
Desenvolvimento baseado em Componentes

- **Exemplo de um adaptador conectando o coletor de dados e um sensor.**



Desenvolvimento baseado em Componentes

- **Exemplo de um adaptador conectando com 3 componentes.**



Implementação

- Estágio crítico, onde é desenvolvido uma versão executável do sistema.
- Envolve desde a programação em alto ou baixo nível, assim como customização e adaptação de sistemas prateleiras.
- Aspectos de Implementação:
  - **Reuso**: desenvolver sistemas com o maior número de códigos já existentes.
  - **Gerenciamento de Configuração**: gerenciar as versões dos componentes criadas durante o desenvolvimento.
  - **Desenvolvimento host-target**: geralmente, o software de produção não é executado no mesmo computador que o ambiente de desenvolvimento de software. .

Aspectos de Implementação

- **Reuso**:
  - Em diferentes níveis: abstrato, objeto, componentes e sistemas.
  - Custos existentes:
    - Tempo de procura e analise do software para reúso.
    - Aquisição de um grande software de prateleira pode elevar o orçamento do projeto.
    - Custos de adaptação e configuração dos componentes para o software em desenvolvimento.
    - Custos de integração dos componentes reusável com o código que desenvolveu.

Aspectos de Implementação

- **Gerenciamento de Configuração:**
  - Importante quando uma equipe está colaborando no desenvolvimento de software.
  - Atividades:
    - Gerenciamento de Versões: suporte para manter o controle das diferentes versões de componentes.
    - Integração de Sistema: suporte que auxilia a definir quais versões dos componentes são de determinada versão do sistema.
    - Rastreamento de Problemas: suporte que permite os usuários reporta bugs e outros problemas.

Aspectos de Implementação

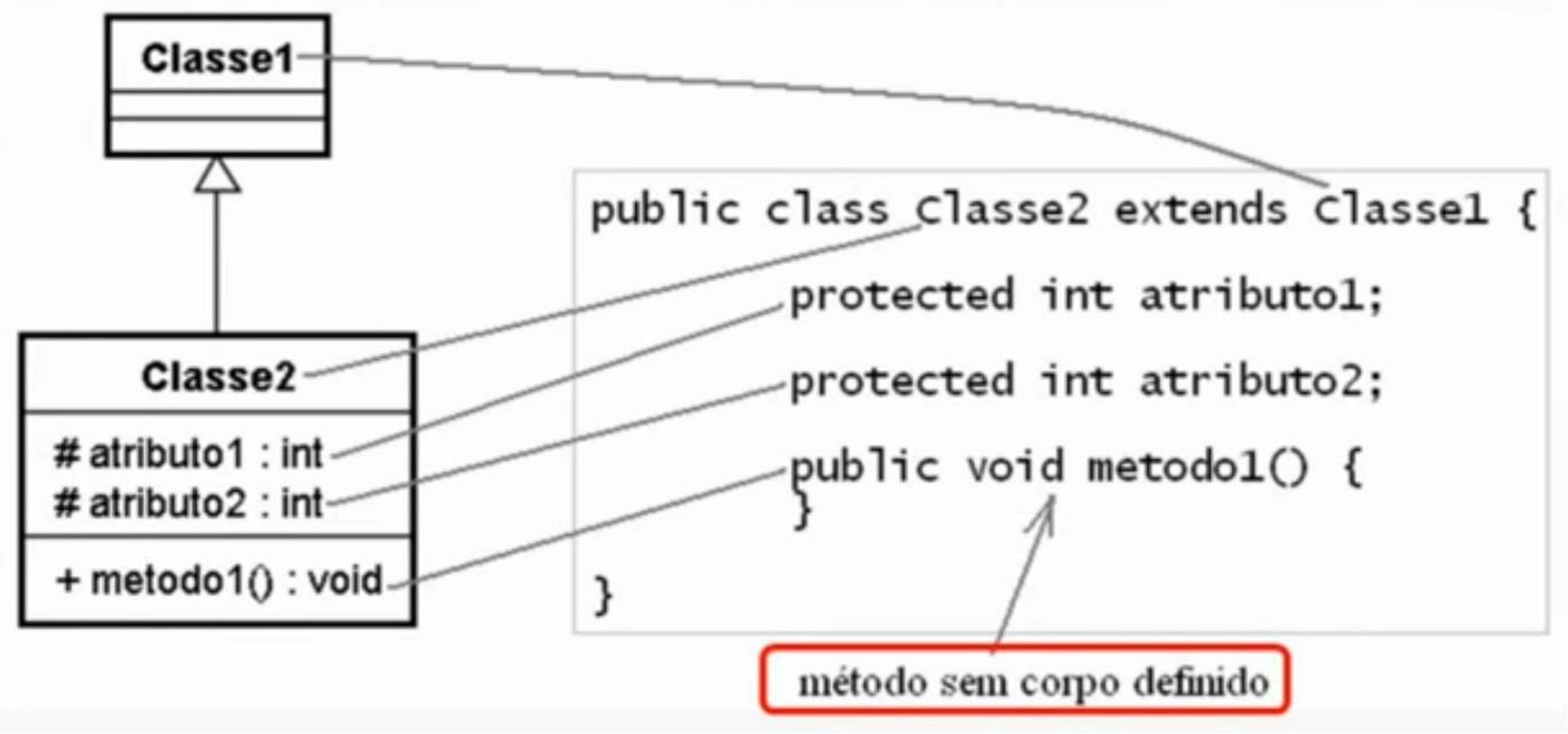
- **Desenvolvimento host-target**
  - Host: computador/plataforma onde o software é desenvolvido.
  - Target: computador/plataforma onde o software é executado.

  Outros aspectos:
  - Os requisitos do componente: a target deve fornecer o hardware requerido e o suporte de software para o componente.
  - Os requisitos de disponibilidade: em muitos casos uma implementação alternativa do componente deve ser disponível no caso de falha.
  - Comunicação de componentes: quando ocorre um volume de tráfego muito grande entre componentes, é importante mantê-los em mesma plataforma ou fisicamente próximas.

Codificação

- "...tem por objetivo **traduzir as especificações de software em códigos** de programa que possam ser processados por um sistema computacional." (Hirama, 2011)
- A qualidade e a manutenibilidade do software está diretamente relacionado com a linguagem de programação escolhida e o estilo de codificação.
  - A boa qualidade do programa é refletida nas seguintes características: ser modular, funcionar, estar de acordo com as especificações, ser flexível, bem estruturado, sem defeitos, ser documentado e de bom desempenho.
- Documento dessa atividade é a **Listagem de Programas(código-fonte)**.
  - Escrito pelos programadores em alguma linguagem de programação.

Codificação



Codificação

- Critérios para escolha da linguagem de programação:
  - Área de aplicação(industrial, comercial, financeira...)
  - Complexidade algorítmica(custo de tempo e memória)
  - Complexidade de estrutura de dados(filas, listas, pilhas...)
  - Requisitos de desempenho (tempo de resposta, *throughput*...)
  - Ambiente de execução (plataformas Java, Windows, .Net...)
  - Ferramentas CASE(abrange todas as ferramentas baseadas em computadores que auxiliam atividades da ES.

Codificação

- Classificação das Linguagens de programação:
  - 1ª Geração (década de 40-60): iniciou a partir do primeiro computador digital ENIAC (Electrical Numerical Integrator and Computer) e é constituída de código de máquina e linguagem Assembly.
  - 2ª Geração (década de 60-70): é constituída, entre outras, das linguagens Fortran, Cobol, Algol e Basic.
  - 3ª Geração (década de 70-80): é constituída, entre outras, das linguagens Pascal, C, C++, Smalltalk, Lisp e Prolog.
  - 4ª Geração (década de 80-presente): é constituída entre outras de linguagens de consulta(SQL), geradores de programas automáticos, linguagens de prototipação (Visual Basic e Delphi).

Codificação

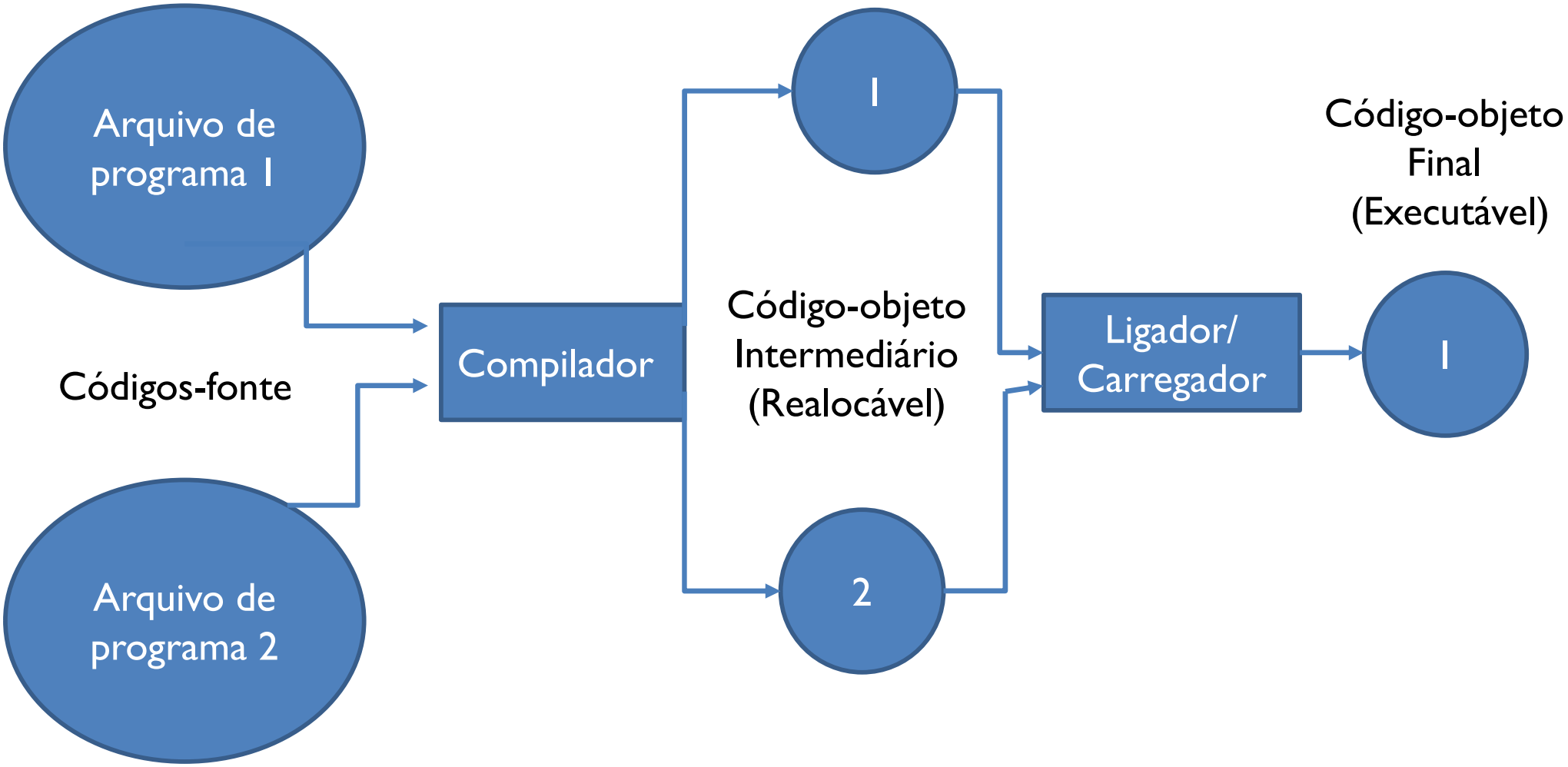
- Código-fonte:
  - Deve seguir as regras de sintaxe da linguagem.
  - Deve ser inteligível(favorecendo a clareza e a simplicidade).
  - Deve atender a um paradigma de programação.

Codificação

- Processo típico de geração de código.
  - Inicia-se com a tradução do código-fonte por um programa **compilador**, que gera um código-objeto intermediário.
  - Em seguida, os códigos-objeto intermediários são ligados através de um programa **ligador/carregador** que gera um código-objeto final que deve ser carregado na máquina-alvo.

Codificação

- Processo típico de geração de código.



Práticas de Codificação

- **Padrões de Codificação**: descreve várias convenções de codificação usadas para implementações inteligíveis, consistentes e com qualidade.
  - **Naming Conventions**: Utilize convenções de nomenclatura consistentes para variáveis, funções e classes.
  - **Indentação**: Mantenha a indentação consistente para melhorar a legibilidade do código.

Práticas de Codificação

- Boas Práticas:
  - **DRY (Don't Repeat Yourself):** Evite duplicação de código para reduzir erros e facilitar a manutenção.
  - **KISS (Keep It Simple, Stupid):** Mantenha o código o mais simples possível.
  - **SOLID Principles**: Siga os princípios SOLID para design orientado a objetos (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion).

Práticas de Codificação

- Boas Práticas:
  - Adicione **comentários claros e concisos** onde necessário para explicar **partes complexas** do código.
  - Escreva **código** que seja **autoexplicativo** sempre que possível.
  - **Pair Programming:** Dois desenvolvedores trabalham juntos no mesmo código para aumentar a qualidade e compartilhar conhecimento.
  - **Code Reviews:** Revisões de código ajudam a detectar erros e melhorar a qualidade geral antes da integração.

Controle de Versão

- Importância do Controle de Versão:
  - **Mantém o histórico** de mudanças no código.
  - **Facilita a colaboração** entre desenvolvedores.
  - **Permite reverter a versões anteriores** em caso de erros.
- Ferramentas de Controle de Versão:
  - **Git**: Amplamente utilizado, distribuído, poderoso.
  - **SVN** (Subversion): Centralizado, adequado para certos tipos de projetos.

Controle de Versão

- Fluxo de Trabalho com Git:
  - **Branches**: Criação de ramos para desenvolvimento paralelo.
  - **Commits**: Salvar mudanças de forma incremental e descritiva.
  - **Merges**: Unir mudanças de diferentes branches.
- Práticas Recomendadas:
  - **Commit Messages:** Escrever mensagens de commit claras e descritivas.
  - **Pull Requests:** Revisar mudanças antes de integrar no branch principal.
  - **Branches**: Usar branches para funcionalidades específicas, correções de bugs, e experimentos.

Testes

- "Tem por objetivo descobrir defeitos no software, considerando aspectos estruturais e lógicos do software." (Hirama, 2011)
  - Verificar a presença de defeitos.
- Mais qualidade quanto menos defeitos de software.
- Os conceitos, as estratégias, as técnicas e as métricas do teste devem ser integrados em um **processo** definido e controlado.
- Deve ser apoiada por um **Plano de Testes**.
  - A norma **IEEE 829-2008** descreve o que é necessário para uma boa documentação de teste.

Testes

- **Tipos de Testes:**
  - **Testes Unitários:** verificam se partes isoladas do código (unidades, como funções ou métodos) estão funcionando corretamente.
  - **Testes de Integração:** avaliam a interação entre diferentes módulos ou componentes do sistema.
  - **Testes de Sistema:** verificam o sistema completo para garantir que ele atenda aos requisitos especificados.

Testes

- **Tipos de Testes (exemplos):**
  - **Testes Unitários:** verificar se a função soma(a, b) retorna a soma correta dos dois números.

```
def test_soma():
    assert soma(2, 3) == 5
```

Testes

- **Tipos de Testes (exemplos):**
  - **Testes Integração:** Verificar se a **autenticação** funciona corretamente quando um usuário tenta fazer login pelo **frontend**.

```python
def test_login_frontend():
    resposta = frontend.login('usuario', 'senha123')
    assert resposta.status_code == 200
    assert 'token' in resposta.json()
```

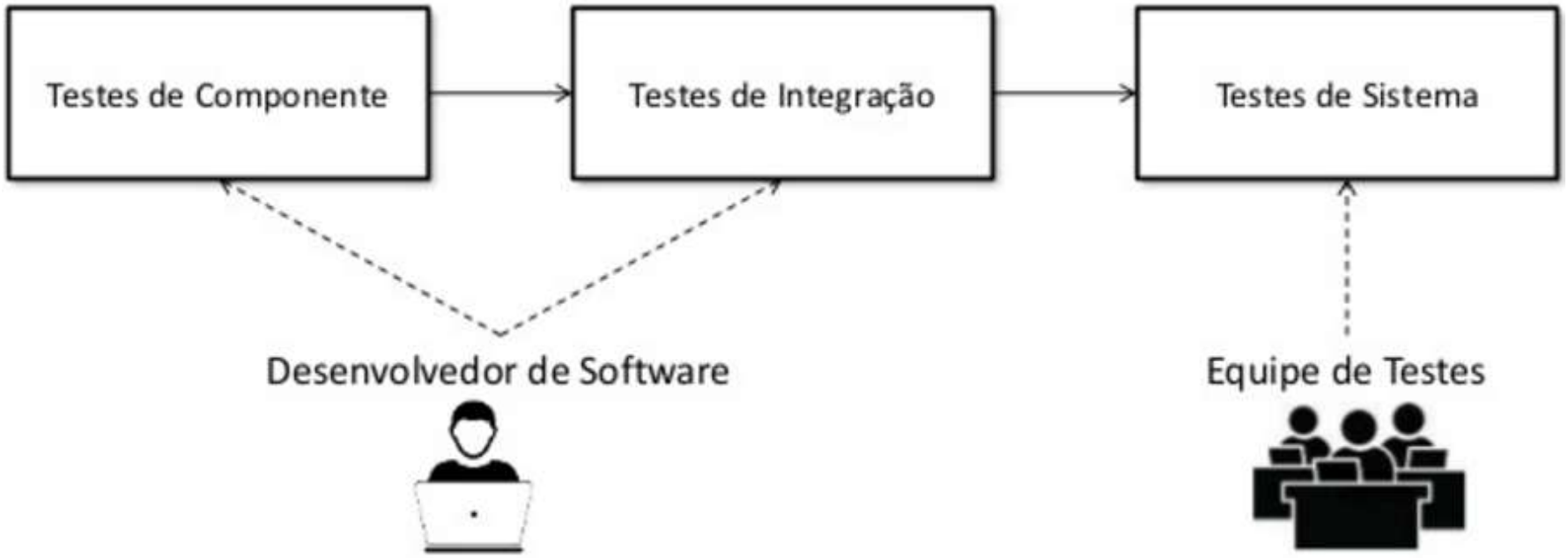
Testes

- **Tipos de Testes (exemplos):**
  - **Testes Sistema:** testar o agendamento de consultas, verificando se um médico pode ser agendado corretamente e se o paciente recebe a confirmação.

```python
def test_agendamento_consulta():
    resposta = sistema.agenda_consulta('paciente_id', 'medico_id', 'data_hora')
    assert resposta.status_code == 200
    assert resposta.json()['status'] == 'confirmado'
```

Testes

- Fases de um processo de teste.



Testes

- Processo de Teste:
  - Deve apoiar as atividades e fornecer um guia para as equipes de teste.
    - Desde o planejamento até a avaliação dos resultados.
  - Deve ser executado em fases, de forma a torná-lo mais eficaz.
  - Semelhante ao **Modelo V** de desenvolvimento de software.

Testes

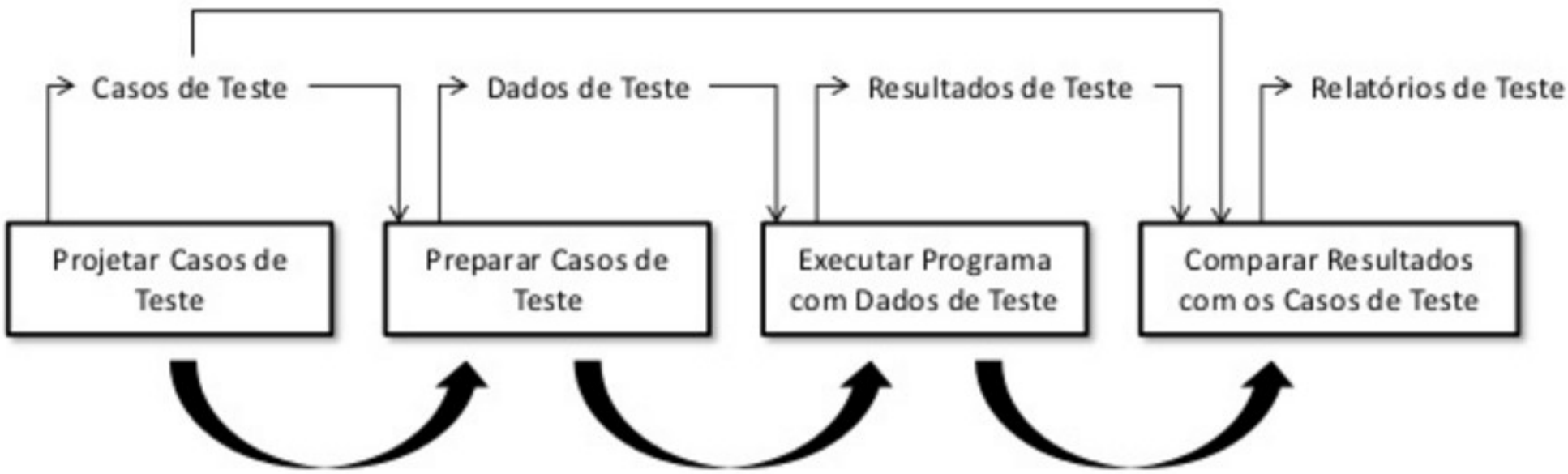
- Ferramentas de Teste:
  - **JUnit**: Uma popular framework de testes para Java.
  - **PyTest**: Uma framework de testes para Python que suporta testes unitários, funcionais e de integração.
  - **Selenium**: Uma ferramenta para testes automatizados de aplicações web.
- Técnicas de Depuração:
  - **Breakpoints**: Pontos de interrupção no código onde a execução pode ser pausada para inspeção.
  - **Logging**: Registro de eventos durante a execução do software para análise posterior.
  - **Debuggers**: Ferramentas que permitem a execução passo a passo do código para identificar problemas.

Testes

- Um procedimento de teste completo tem os seguintes passos:
  - Projetar casos de teste.
  - Preparar dados de teste.
  - Executar programa com dados de teste.
  - Comparar resultados com os casos de teste.

Testes

- Procedimento de teste.



Testes

- Procedimento de teste.
  - Exemplos de Casos de Teste.

ID	CT-001
Caso de Teste:	Efetuar login no sistema.
Funcionalidade:	Login
Pré - Condição:	1. Possuir usuário válido para efetuar o login.
Procedimento:	1. Acessar o sistema; 2. Inserir usuário e senha; 3. Clicar em "Login".
Resultado Esperado:	1. Login do usuário efetuado com sucesso.

ID	CT-002
Caso de Teste:	Cadastrar cliente no Sistema
Funcionalidade:	Cadastro de cliente
Pré - Condição:	Estar logado no sistema
Procedimento:	1. Entrar na tela "Cadastro" -> "Cliente" 2. Inserir dados do cliente. 3. Clicar em "Salvar".
Resultado Esperado:	1. Cliente é cadastrado com sucesso..

Testes

- Papéis dos envolvidos.

Gestor da Qualidade

Responsável pelas ações de controle de qualidade dos produtos

Líder do projeto de Teste

Responsável por liderar e planejar o projeto de teste

Arquiteto de Teste

Responsável por montar e disponibilizar o ambiente de teste

Analista de Teste

Responsável pela modelagem e elaboração dos casos de teste e scripts de teste

Testador

Responsável pela execução dos casos de teste e pela documentação dos resultados

Usuários

Responsável pela avaliação e aceitação do sistema

Testes

- **Plano de Testes**: define objetivos para cada tipo(ou fase) de teste, estabelece estratégias de teste, cronograma e responsabilidades, procedimentos e padrões a serem usados na execução e elaboração de relatórios de testes, e define critérios para a conclusão do teste, bem como o sucesso de cada teste.

Testes

- Norma IEEE 829:2008.
  - **Plano de teste:** apresenta o planejamento para a execução de teste incluindo abrangência, abordagem, recursos e cronograma. Identifica os itens e as funcionalidades a serem testadas, as tarefas a serem realizadas e os riscos relacionados a atividade de teste.
  - **Especificação do projeto de teste:** refina a abordagem apresentada no plano de teste, identifica as funcionalidades e características a serem testadas pelo projeto e seus testes associados. Também identifica os casos e procedimentos de testes e apresenta critérios de aprovação. Em alguns casos é incluído ou incorporado ao plano de testes.

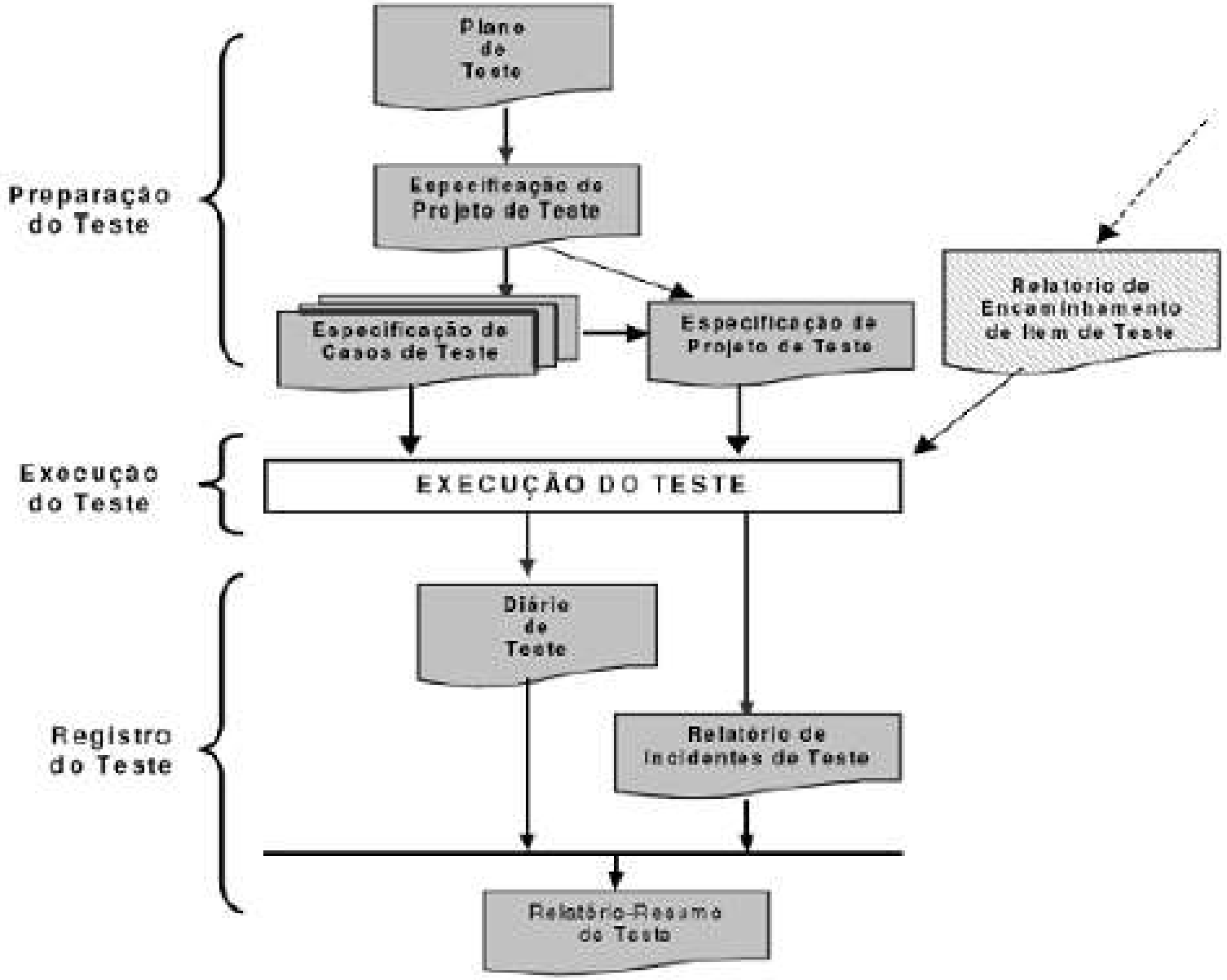
Testes

- Norma IEEE 829.
  - **Especificação do caso de teste:** define os casos de testes incluindo dados de entrada, resultados esperados, ações e condições gerais para os testes.
  - **Especificação de procedimento de teste:** especifica os passos para executar os procedimentos de casos de teste.
  - **Diário de teste:** documenta qualquer evento que ocorra durante a atividade de teste e que requeira analise posterior.

Testes

- Norma IEEE 829.
  - **Relatório de incidente de teste:** todos os defeitos encontrados durante o teste são registrados e passados para a equipe de desenvolvimento para as devidas correções.
  - **Relatório Resumo de Teste:** apresenta de forma resumida os conceitos das atividades de teste associados com uma ou mais especificações de projeto de testes e prove avaliações baseadas nesses resultados.
  - **Relatório de encaminhamento de item de teste:** identifica os itens encaminhados para teste no caso de equipes distintas de desenvolvimento e teste.

A Norma IEEE 829



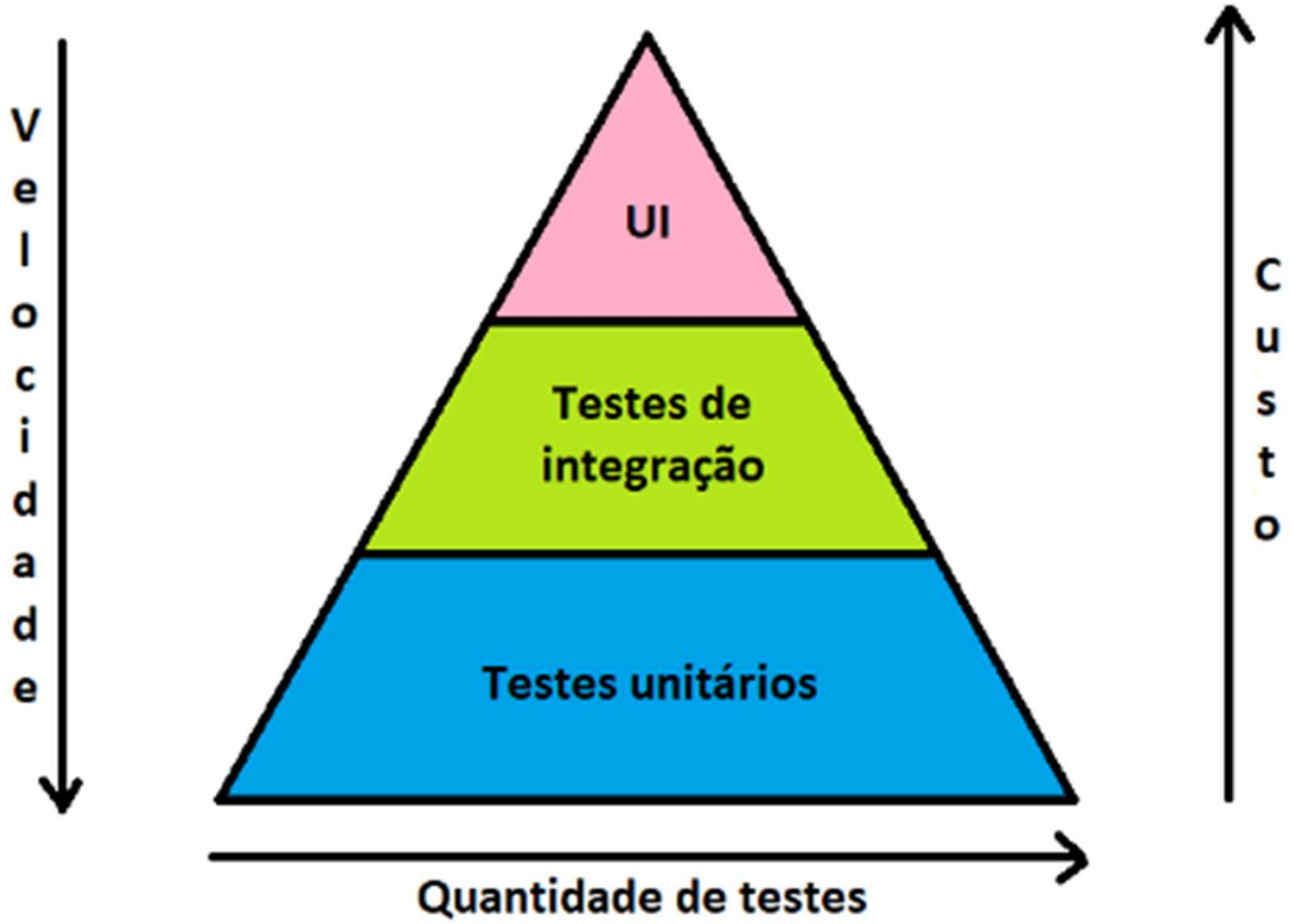
Testes

Estratégia de Teste de Software:

https://www.youtube.com/watch?v=bmzeuk3zVS4&t=34s

Testes

Pirâmide de Teste de Software:

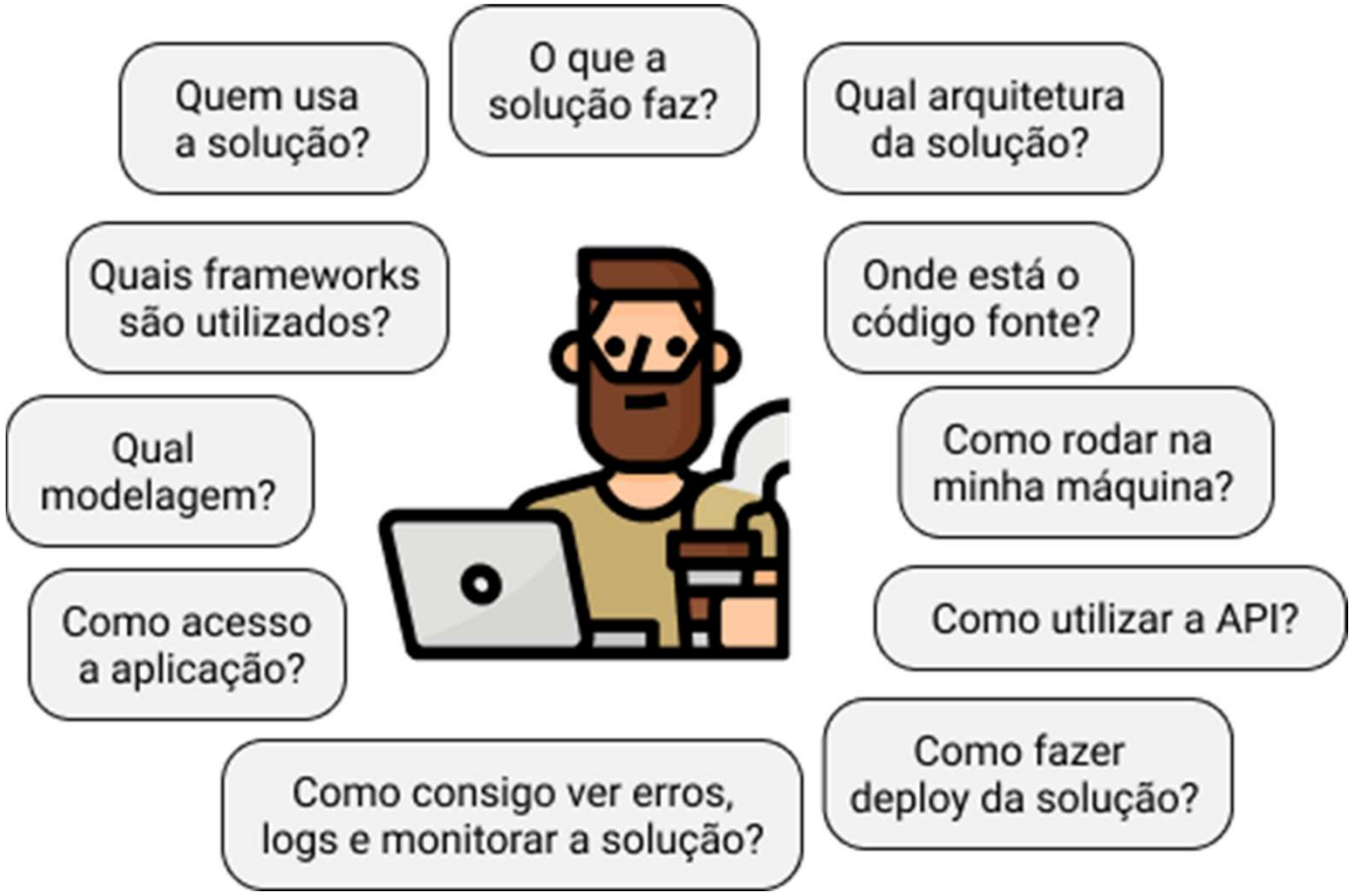


Testes

- Automação de Testes e CI/CD:
  - **Automação de Testes:** Uso de scripts e ferramentas para executar testes automaticamente.
  - **Integração Contínua** (CI): Prática de mesclar todas as alterações de código em um repositório central várias vezes ao dia.
  - **Entrega Contínua** (CD): Extensão da CI que automatiza o processo de entrega do software em produção.

Documentação do Código

- Facilita a compreensão do código por outros desenvolvedores.
- Ajuda na manutenção e evolução do software.
- Serve como referência durante a depuração e testes.



Documentação do Código

- Tipos de Documentação:
  - **Comentários**: Explicações inseridas diretamente no código.
    - Comentários de linha única e de bloco.
  - **README**: Arquivo que fornece uma visão geral do projeto, como instalar e usar.
  - **Wiki**: Documentação mais detalhada, frequentemente hospedada em repositórios de código.
  - **Manual do Usuário**: Guia para os usuários finais sobre como utilizar o software.

Documentação do Código

- Ferramentas de Documentação:
  - **Sphinx**: Ferramenta para documentação de projetos em Python.
  - **Javadoc**: Utilizado para gerar documentação API em Java.
  - **Doxygen**: Ferramenta de documentação para várias linguagens de programação.
- Manutenção e Atualização da Documentação:
  - **Documentação Contínua:** Atualize a documentação conforme o código evolui.
  - **Revisões Regulares:** Revise a documentação periodicamente para garantir que está atualizada.
  - **Automação**: Utilize ferramentas que integrem a documentação com o processo de desenvolvimento (CI/CD).

Exemplos práticos

- Exemplo 1: Implementação de uma Função Simples em Python

```
def soma(a, b):
    """
    Esta função retorna a soma de dois números.
    :param a: Primeiro número
    :param b: Segundo número
    :return: Soma de a e b
    """
    return a + b
```

Exemplos práticos

- Exemplo 2: Uso de Controle de Versão com Git
  - Criação de um novo repositório, commit inicial e push para um repositório remoto.

```
git init
git add .
git commit -m "Commit inicial"
git remote add origin https://github.com/usuario/repo.git
git push -u origin master
```

Conclusão

- As atividades de Codificação e Teste fazem parte da Implementação.
- A atividade de Codificação é fortemente influenciada pela atividade de Projeto e a escolha da linguagem de programação.
- A atividade de codificação é trabalhosa e, portanto, deve se incorporar ferramentas CASE e reuso de programas.
- O planejamento antecipado influencia diretamente a atividade de Testes.
- Os teste muitas vezes são negligenciados por vários motivos, tais quais: projeto atrasado, testes não planejados, sem ferramentas adequadas e pessoal qualificado.
- A atividade de Testes é parte dos conceitos de **Verificação e Validação**.

Atividade

- Crie 2 componentes e realize uma integração entre eles.
  - Faça o diagrama de componentes
  - Defina as interfaces provida e requerida, as informações de uso e de implantação.
  - Sem codificação.
  - https://www.youtube.com/watch?v=m41V13kqP_s

Extra

- https://www.youtube.com/watch?v=BLFxInj08Is
- https://www.youtube.com/watch?v=rkSNCEuxXKw
- https://www.youtube.com/watch?v=7nUsxS2wq58
- https://www.youtube.com/watch?v=xL52q8mLkMI

Referências

- PRESSMAN, R. S. *Engenharia de Software. 7ª* Edição. McGraw Hill, 2010.
- JUNIOR, H. E. *Engenharia de Software na Prática.* 2ª Edição. São Paulo: Novatec, 2010.
- SOMMERVILLE, I. *Engenharia de Software*. 9ª Edição. São Paulo: Pearson Prentice Hall, 2011.