Laboratório de Engenharia de Software

Testes de software

Prof. Marco Antonio Furlan de Souza

Problemas famosos

- **F-16 : cruzar o equador com piloto automático**
  - Resultado: capotamento do avião
  - Motivo?
    - Reutilização do software de piloto automático

- **Acidentes do Therac-25 (1985-1987) – máquina de terapia nuclear – pelo menos cinco mortes**
  - Motivo: manipulação de eventos problemática
- **NASA Mars Climate Orbiter (September 23, 1999) – entrada incorreta na órbita**
  - Motivo: problemas na unidade de conversão.

Terminologia

- **Falha**: Qualquer desvio do comportamento observado em relação ao comportamento especificado
- **Estado de erro (erro)**: O sistema está num estado tal que o processamento pelo sistema pode levar a uma falha
- **Defeito**: A causa mecânica ou algorítmica de um erro ("bug")
- **Validação**: Atividade de verificação de desvios entre o comportamento observado de um sistema e sua especificação.

Exemplos de defeitos e erros

Defeitos na especificação de interface

- Incompatibilidade entre o que um cliente precisa e o que o servidor oferece
- Incompatibilidade entre requisitos e implementação

Defeitos algorítmicos

- Ausência de inicialização
- Condição de desvio incorreta
- Teste de nulo ausente

Falhas mecânicas (mais difíceis de encontrar)

- Temperatura fora da especificação do equipamento

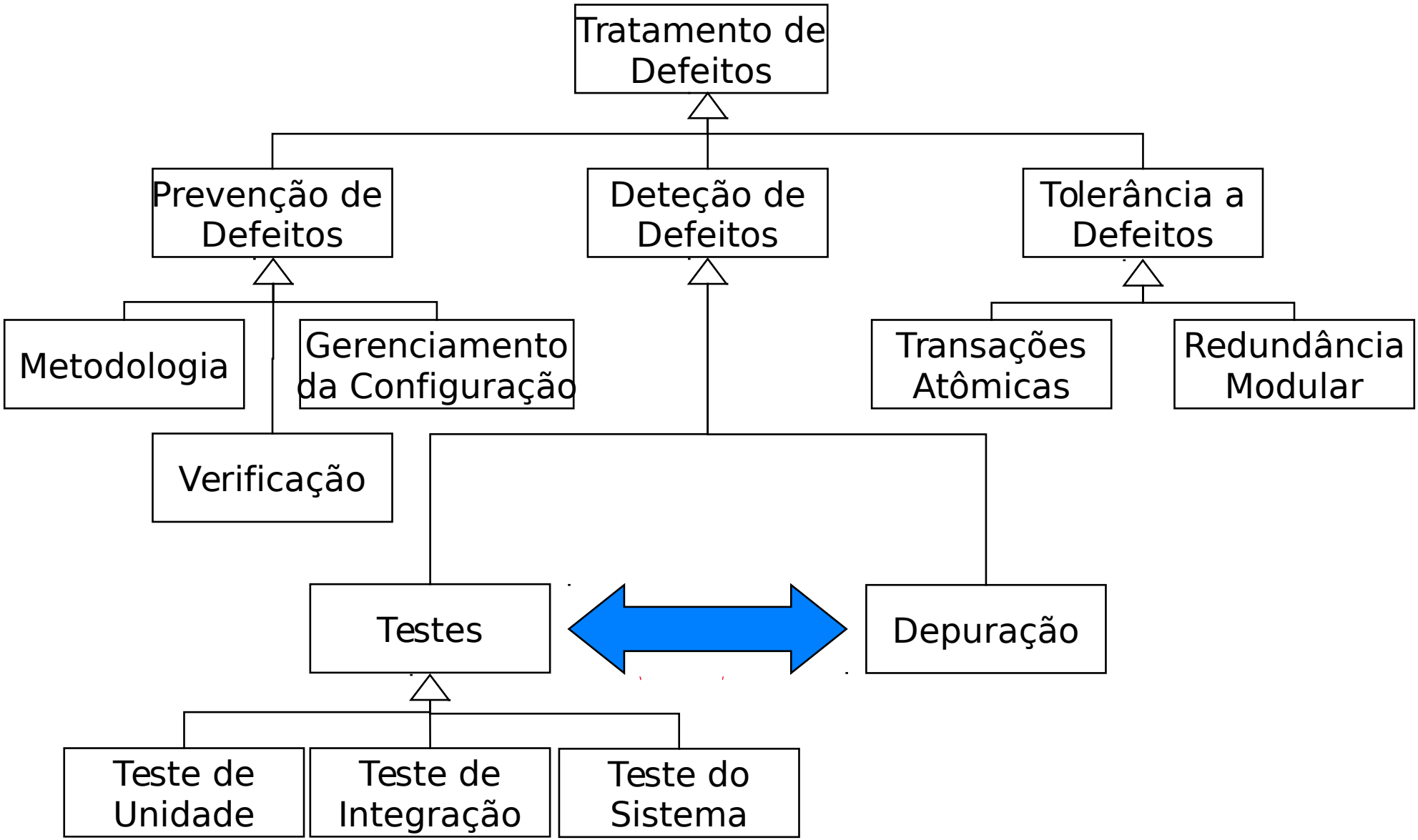
Erros

- Erros de referência a nulo
- Erros de concorrência
- Exceções

Formas de lidar com defeitos

- **Prevenção de defeitos**
  - Utilizar metodologia para reduzir complexidade
  - Utilizar gerenciamento da configuração para prevenir inconsistências
  - Aplicar verificação para prevenir defeitos algorítmicos
  - Empregar revisões
- **Detecção de defeitos**
  - Testes: atividade para provocar falhas de modo planejado
  - Depuração (*debugging*): descobrir e remover a causa (defeito) de uma falha observada
  - Monitoração: enquanto depurando, exibir informações sobre o estado do sistema
- **Tolerância a defeitos**
  - Tratamento de exceção
  - Redundância modular

Técnicas para tratamento de defeitos



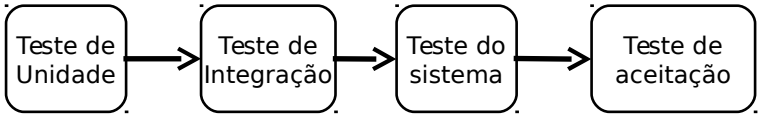
Notas

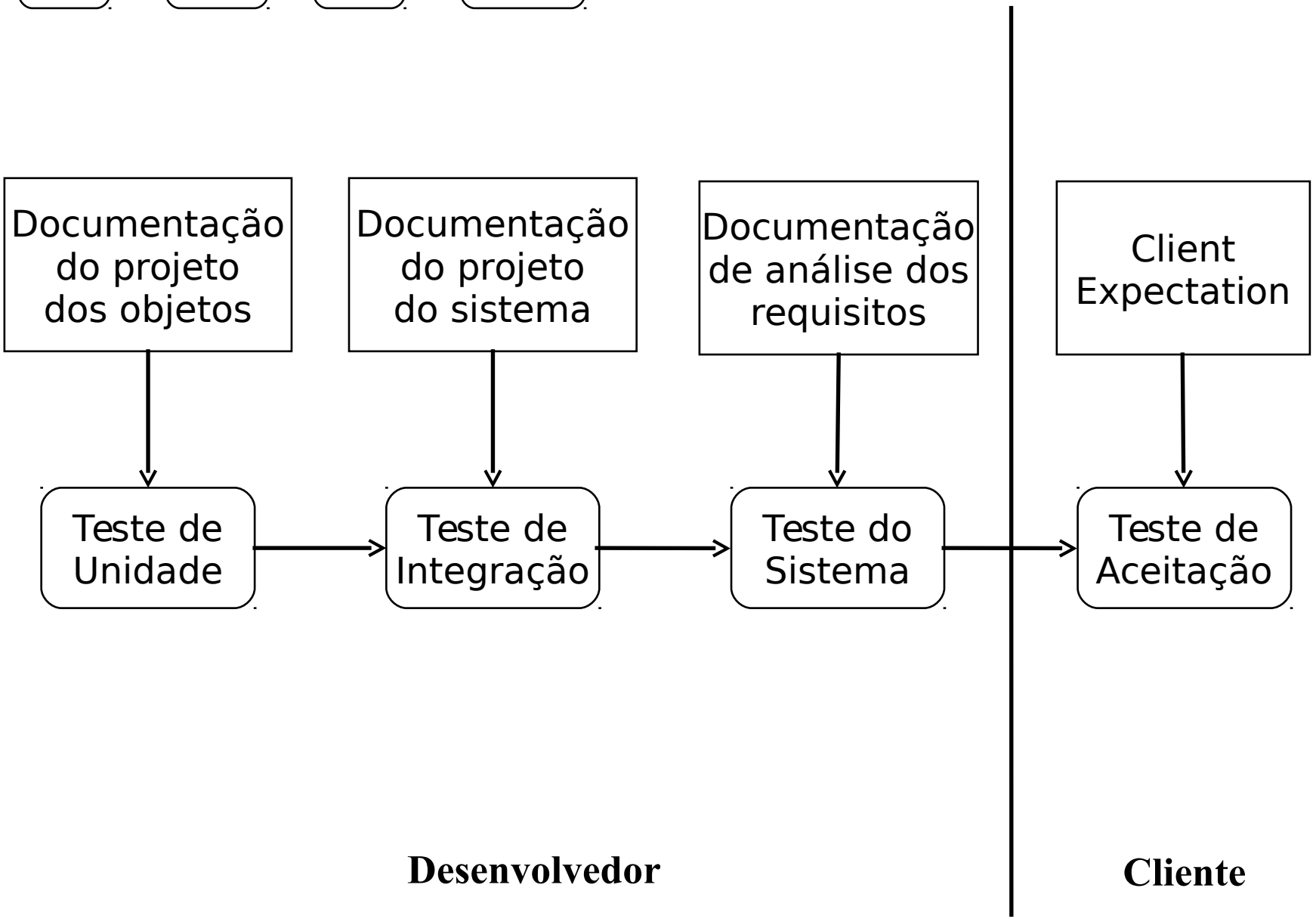
- **É impossível testar completamente qualquer módulo ou sistema não trivial**
  - Limitações práticas: testes exaustivos são proibitivos em tempo e custo
  - Limitações teóricas: por exemplo, o Problema da Parada (Turing)
- **“Testes apontam apenas a presença de erros, não sua ausência” (Dijkstra).**
- **Testar não é gratuito!**

Testar é uma atividade criativa

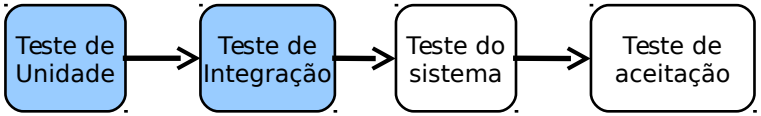
- **Testes eficazes necessitam de**
  - Entendimento detalhado do sistema
  - Conhecimento do domínio da aplicação e da solução
  - Conhecimento de técnicas de teste
  - Habilidades para aplicar as técnicas
- **Testes são melhor realizados por testadores independentes**
  - Nós desenvolvemos uma atitude mental que o programa deveria se comportar de um certo modo quando na realidade ele não o faz
  - Programadores tendem a utilizar dados que permite o programa funcionar
  - Um programa em geral não funciona quando testado por alguém mais

Atividades de testes





Tipos de testes



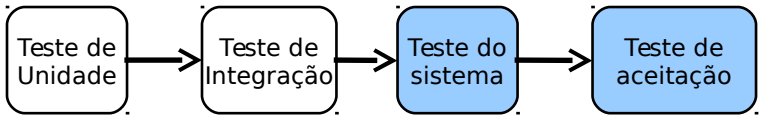
Teste de unidade

- Componentes individuais (classe ou subsistemas)
- Executado pelos desenvolvedores
- Meta: confirmar que o componente ou subsistema está codificado corretamente e executa a funcionalidade intencionada

Teste de integração

- Testes realizados em grupos de subsistemas (ou coleção de subsistemas) e eventualmente o sistema inteiro
- Executado pelos desenvolvedores
- Meta: Testar as interfaces entre os subsistemas.

Tipos de testes



- **Teste do sistema**
  - O sistema inteiro
  - Executado pelos desenvolvedores
  - Meta: Determinar se o sistema cumpre os requisitos (funcionais e não funcionais)
- **Teste de aceitação**
  - Avalia o sistema desenvolvido pelos desenvolvedores
  - Executado pelo cliente
  - Meta: Demonstrar que o sistema cumpre os requisitos e que está pronto para ser usado.

Quando se escrevem testes?

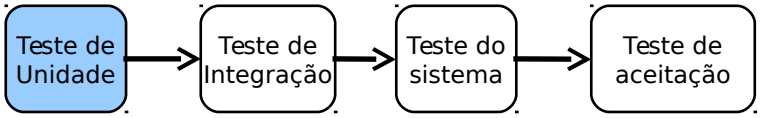
- **Tradicionalmente, após o código-fonte ser escrito**
- **Em XP (eXtreme Programming) é antes do código-fonte ser escrito**
  - Ciclo de desenvolvimento dirigido por testes
    - Adicionar um teste
    - Executar testes automatizados

      => especificar condições para falhas
    - Escrever código
    - Executar testes automatizados

      => verificar se passaram nos testes
    - Refatorar o código.

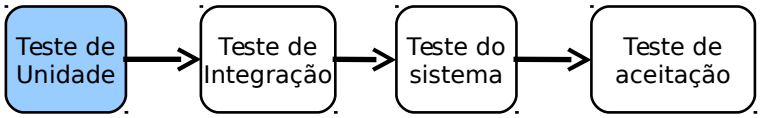
Teste de unidade

- **Teste estático (tempo de compilação)**
  - Análise estática
  - Revisões
    - Walk-through (informal)
    - Inspeção de código (formal)

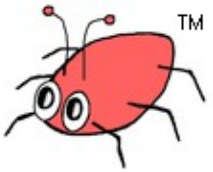


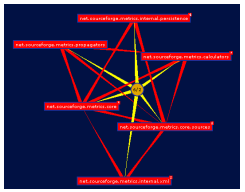
- **Testes dinâmicos (tempo de execução)**
  - Testes “caixa-preta”
  - Testes “caixa-branca”

Análise estática com Eclipse



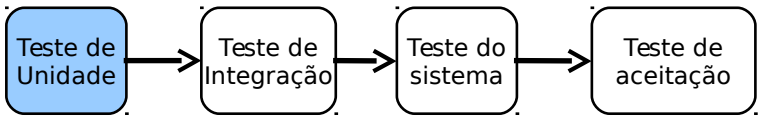
- **Avisos e erros do compilador**
  - Variáveis que podem não ter sido inicializadas
  - Blocos vazios não documentados
  - Atribuições sem efeito
- **Checkstyle**
  - Checa padronizações de código
  - http://checkstyle.sourceforge.net
- **FindBugs**
  - Verifica anomalias no código
  - http://findbugs.sourceforge.net
- **Metrics**
  - Verifica anomalias estruturais
  - http://metrics.sourceforge.net





Teste caixa-preta

- **Foco: Comportamento Entrada/Saída**
  - Se, para alguma entrada, pode-se predizer a saída, então o componente é aprovado no teste
  - Requer: oráculo de teste (*test oracle*)
- **Meta: Reduzir o número de casos de testes pelo particionamento de equivalência:**
  - Dividir condições de entrada em classes de equivalência
  - Escolher casos de teste para cada classe de equivalência.



Caixa-Preta: Seleção de casos de teste

a) Quando uma entrada é válida em um intervalo de valores

- O desenvolvedor seleciona casos de teste a partir de três classes de equivalência:
  - Abaixo do intervalo
  - Dentro do intervalo
  - Acima do intervalo

b) Quando uma entrada é válida apenas se é membro de um conjunto discreto

- O desenvolvedor seleciona caos de teste a partir de duas classes de equivalência:
  - Valores discretos válidos
  - Valores discretos inválidos

Exemplo de teste caixa-preta

```
public class MyCalendar {
    public int getNumDaysInMonth(int month, int year)
        throws InvalidMonthException
        { … }
}
```

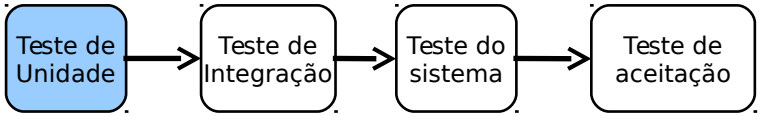
Representação do mês:
1: Janeiro, 2: Fevereiro, …., 12: Dezembro

Representação do ano:
1904, … 1999, 2000,…, 2006, …

Quantos casos de teste para os testes tipo caixa-preta para a função `getNumDaysInMonth()`?

Teste caixa-branca

- Cobertura de código
- Cobertura de ramos
- Cobertura de condições
- Cobertura de caminhos



Heurísticas para testes de unidades

1. **Criar unidades de teste quando o projeto dos objetos estiver completado**

   Teste caixa-preta: testar o modelo funcional

   Teste caixa-branca: testar o modelo dinâmico

2. **Desenvolver casos de teste**

   Meta: Determinar um número efetivo de casos de teste

3. **Remover duplicatas**

4. **Analisar o código-fonte**

   Pode reduzir o tempo de teste

5. **Criar estrutura para teste**

   Drivers e stubs de teste são necessários para teste de integração

6. **Descrever o oráculo de teste**

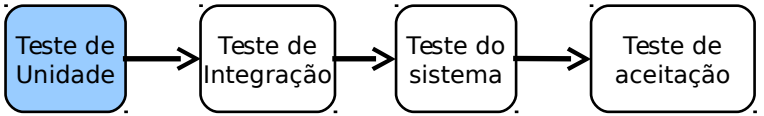
7. **Executar os casos de teste**

   Reeexecutar o teste sempre que uma alteração for realizada – **testes de regressão**

8. **Comparar os resultados dos testes com o oráculo**

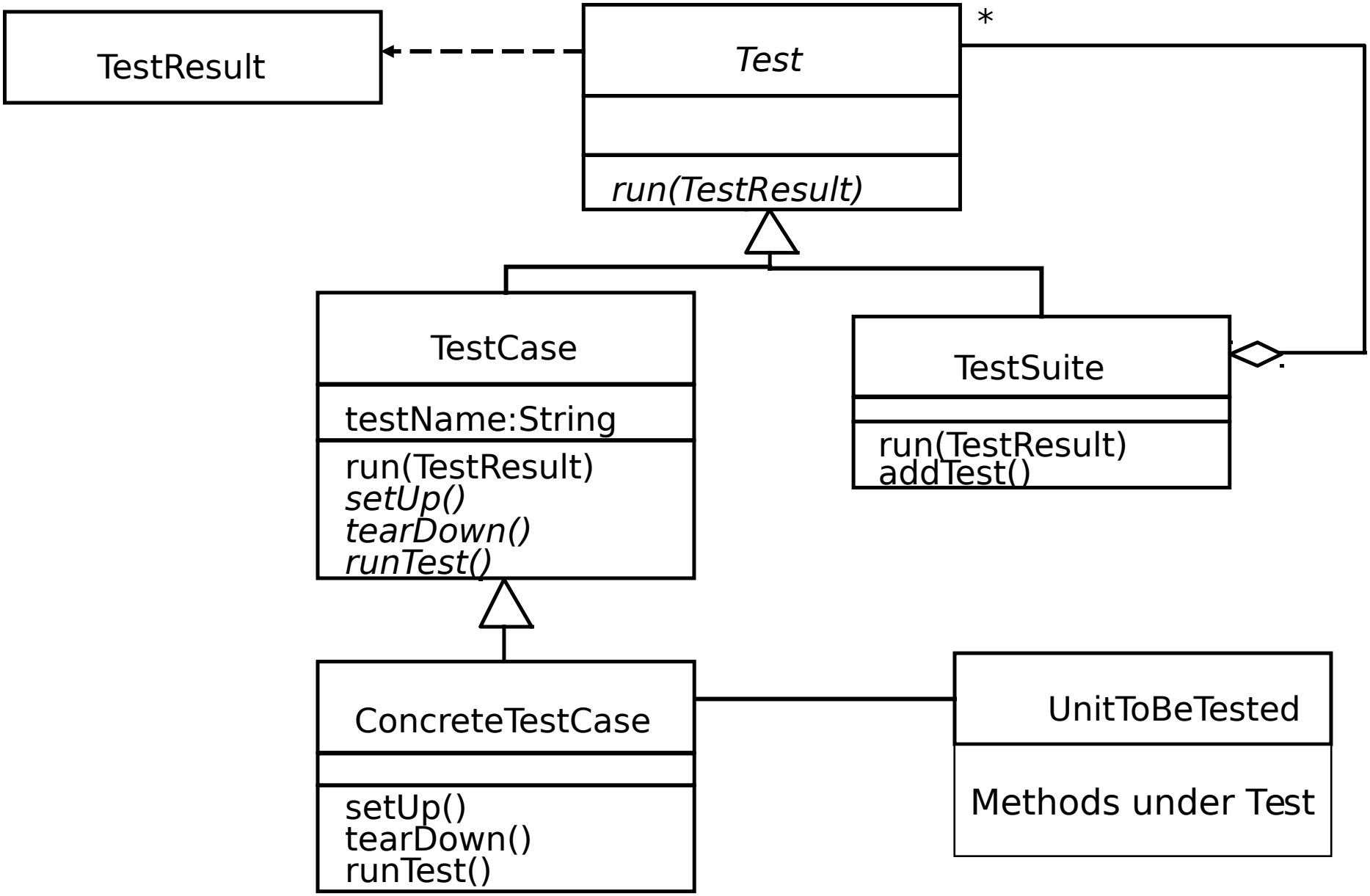
   Automatizar, se possível, esta tarefa

JUnit



- **É um *framework* Java para escrever e executar unidades de teste**
  - Casos e ensaios de teste
  - Conjuntos de teste
  - Executores de teste
- **Elaborado para o estilo de desenvolvimento "testar primeiro" e desenvolvimento baseado em padrões**
  - Testes escritos antes do código
  - Permite testes de regressão
  - Facilita refatoração
- **Código aberto**
  - www.junit.org

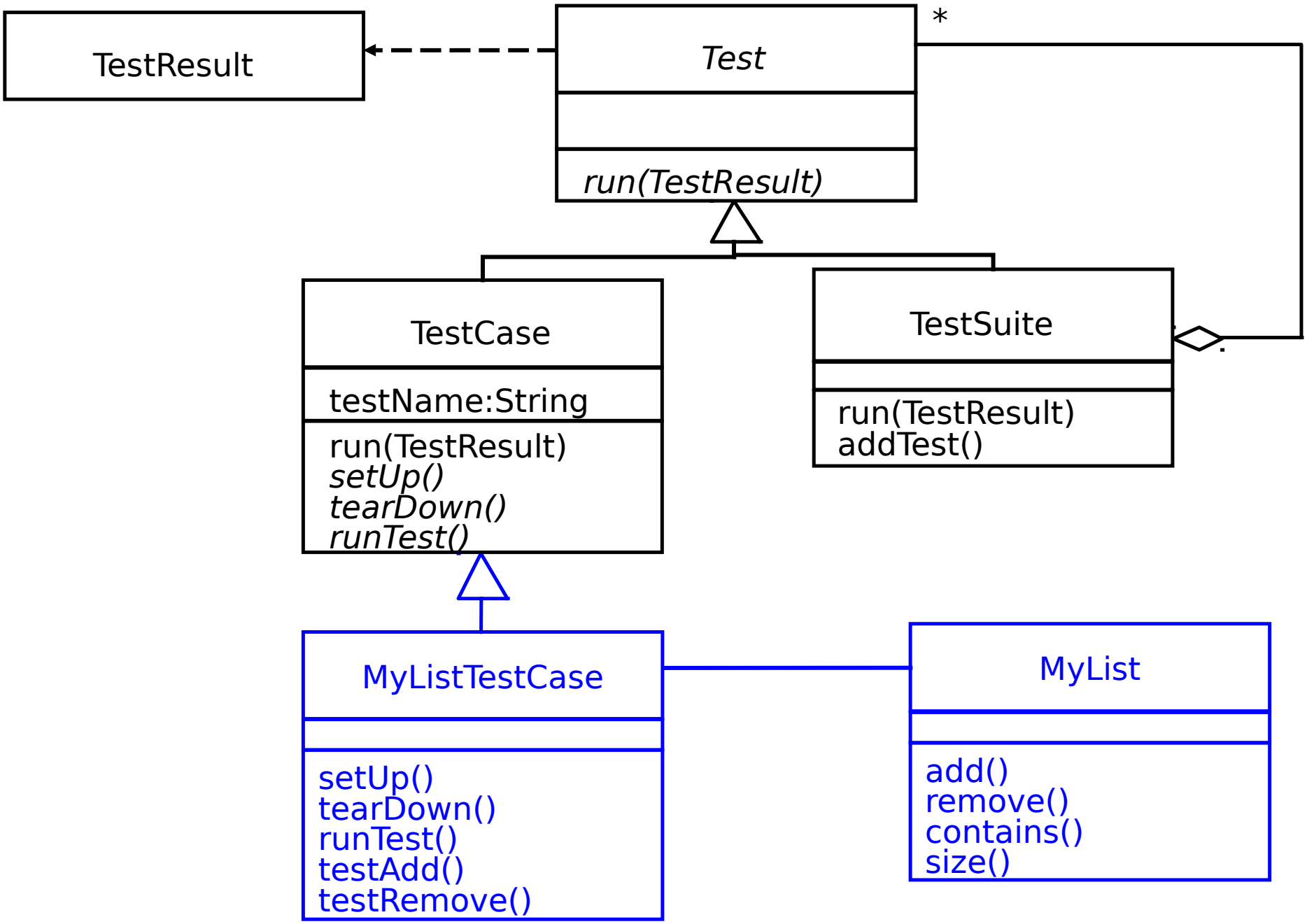
Classes do JUnit



Exemplo: teste da classe MyList

- **Unidade a ser testada**
  - MyList
- **Métodos sob teste**
  - add()
  - remove()
  - contains()
  - size()
- **Caso de teste concreto**
  - MyListTestCase

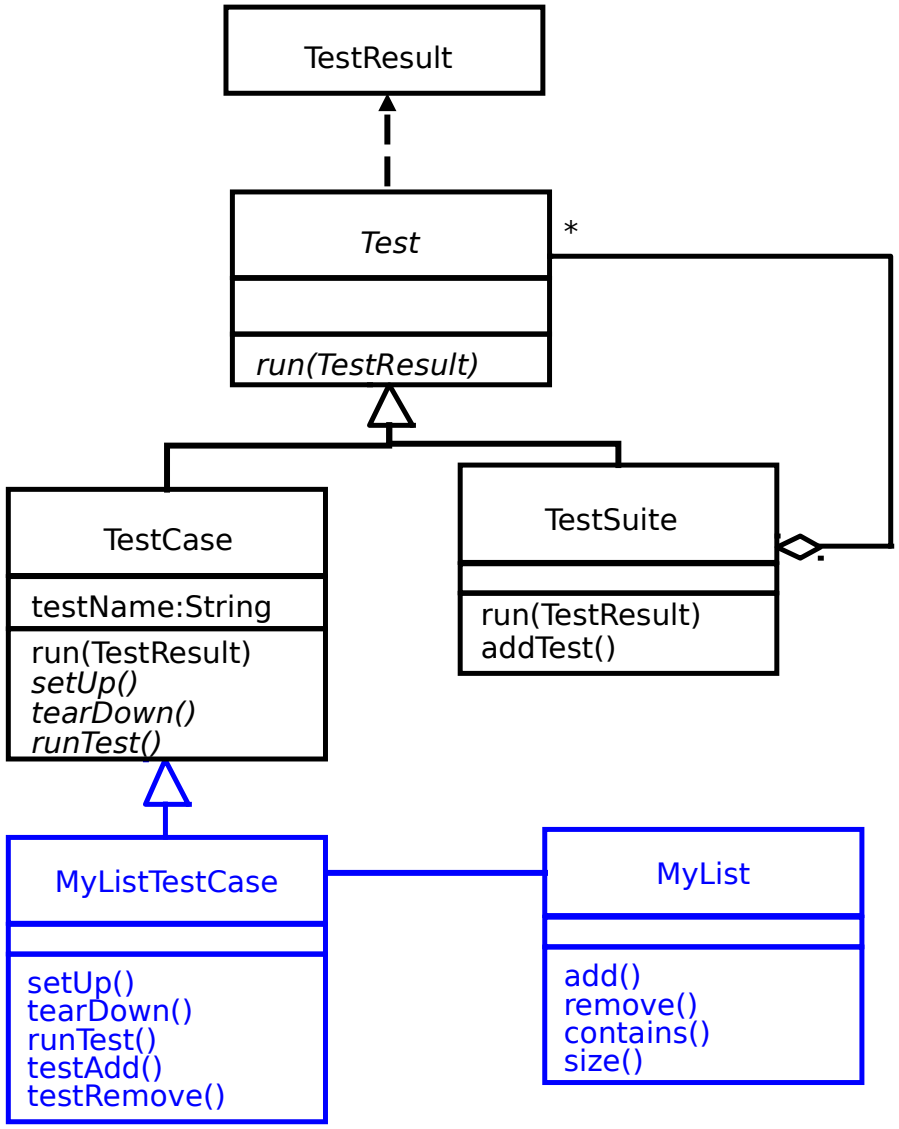
Exemplo: teste da classe MyList



Escrevendo casos de teste no JUnit

```
public class MyListTestCase extends TestCase {
    public MyListTestCase(String name) {
        super(name);
    }
    public void testAdd() {
        // Set up the test
        List aList = new MyList();
        String anElement = "a string";
        // Perform the test
        aList.add(anElement);
        // Check if test succeeded
        assertTrue(aList.size() == 1);
        assertTrue(aList.contains(anElement));
    }
    protected void runTest() {
        testAdd();
    }
}
```

Caso de teste



Ensaios e casos de teste

```
public class MyListTestCase extends TestCase {
// …
    private MyList aList;
    private String anElement;
    public void setUp() {
        aList = new MyList();
        anElement = "a string";
    }
    public void testAdd() {
        aList.add(anElement);
        assertTrue(aList.size() == 1);
        assertTrue(aList.contains(anElement));
    }
    public void testRemove() {
        aList.add(anElement);
        aList.remove(anElement);
        assertTrue(aList.size() == 0);
        assertFalse(aList.contains(anElement));
    }
// …
}
```

Ensaio de teste

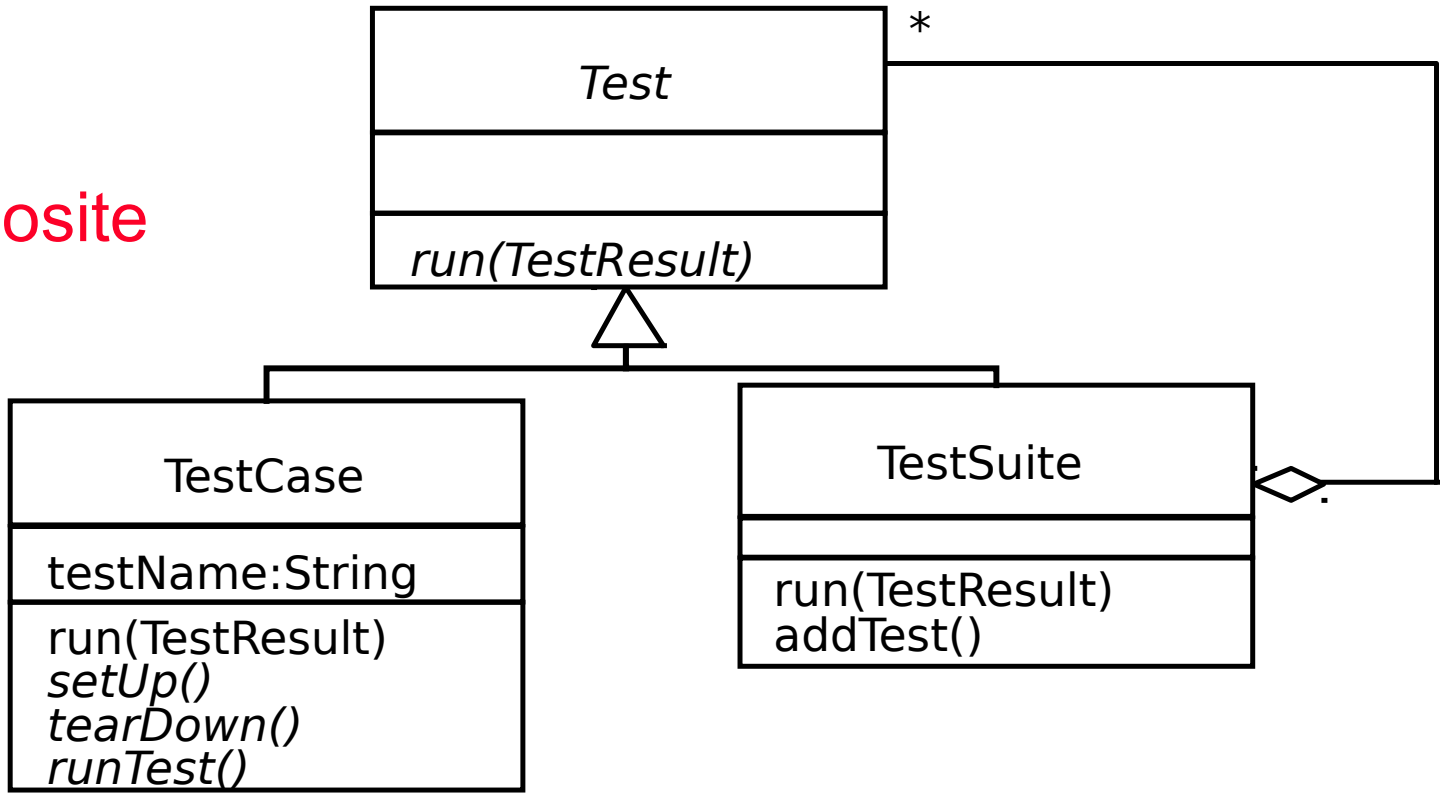
Caso de teste

Caso de teste

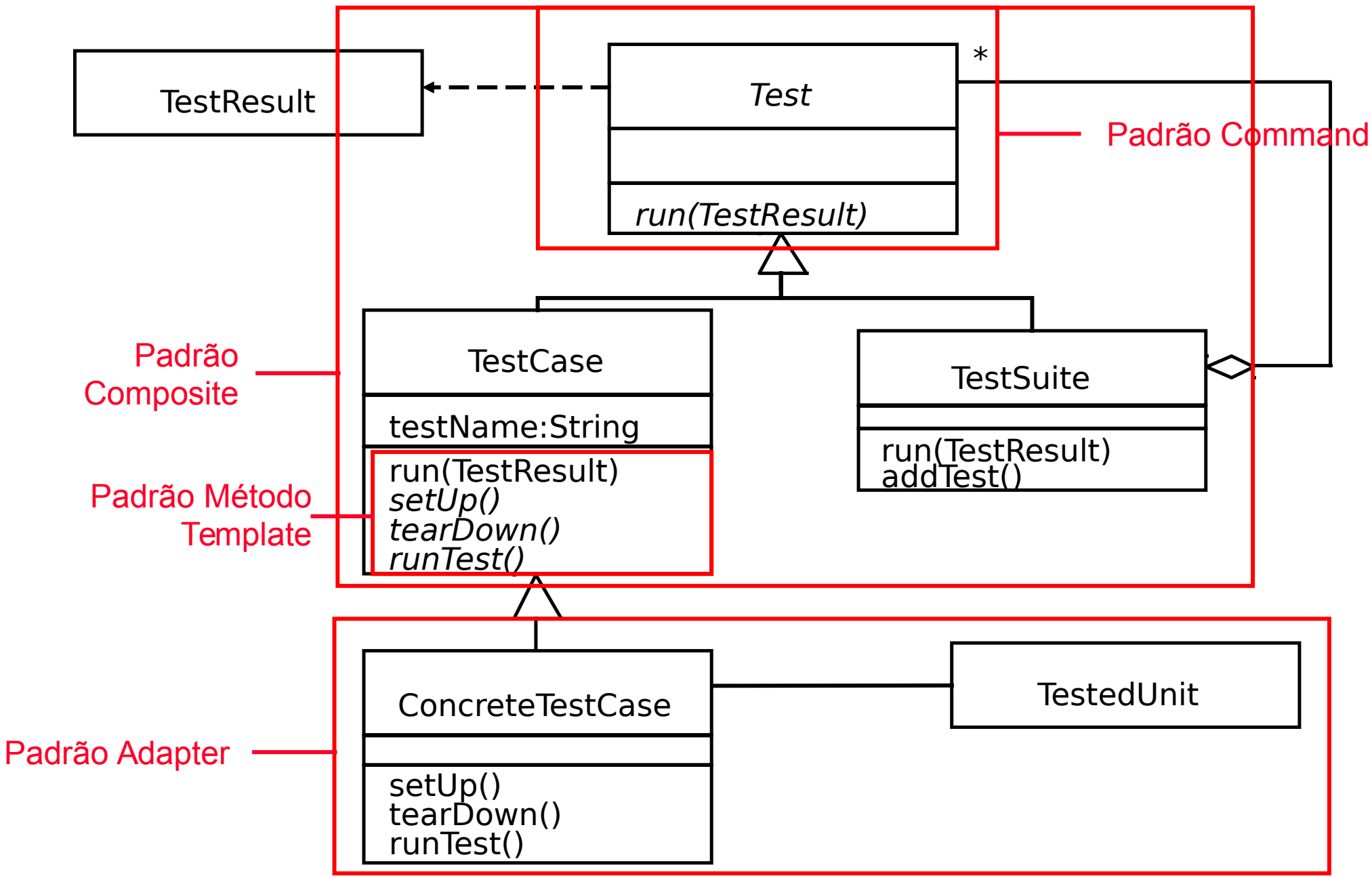
Conjuntos de teste

```
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new MyListTest("testAdd"));
    suite.addTest(new MyListTest("testRemove"));
    return suite;
}
```

Padrão Composite

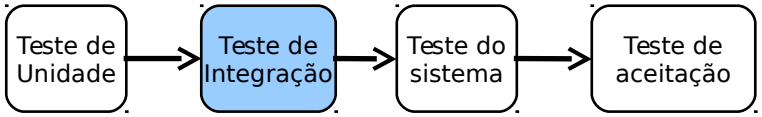


Padrões de projeto no JUnit



Testes de integração

- Todo o sistema é visto como uma coleção de subsistemas (conjuntos de classes) elaborada durante o projeto do sistema e dos objetos
- Objetivo: Testar todas as interfaces entre os subsistemas e a interação dos subsistemas
- A estratégia de teste de integração determina a ordem em que os subsistemas são selecionados para testes e integração.



Por que testes de integração?

- Os testes de unidade só testam unidades isoladas
- Muitas falhas resultam de defeitos na interação de subsistemas
- Muitas vezes, muitos componentes reutilizáveis que não são testados como unidades são usados
- Sem testes de integração, o teste do sistema será muito demorado
- Falhas que não são descobertas em testes de integração serão descobertas depois que o sistema for implantado e podem ser muito caras.

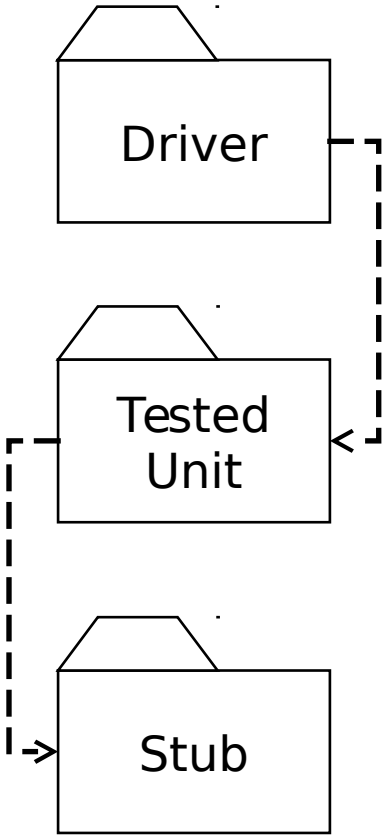
Stubs e drivers

Driver:

- Um componente que chama a `TestedUnit`
- Controla os casos de teste

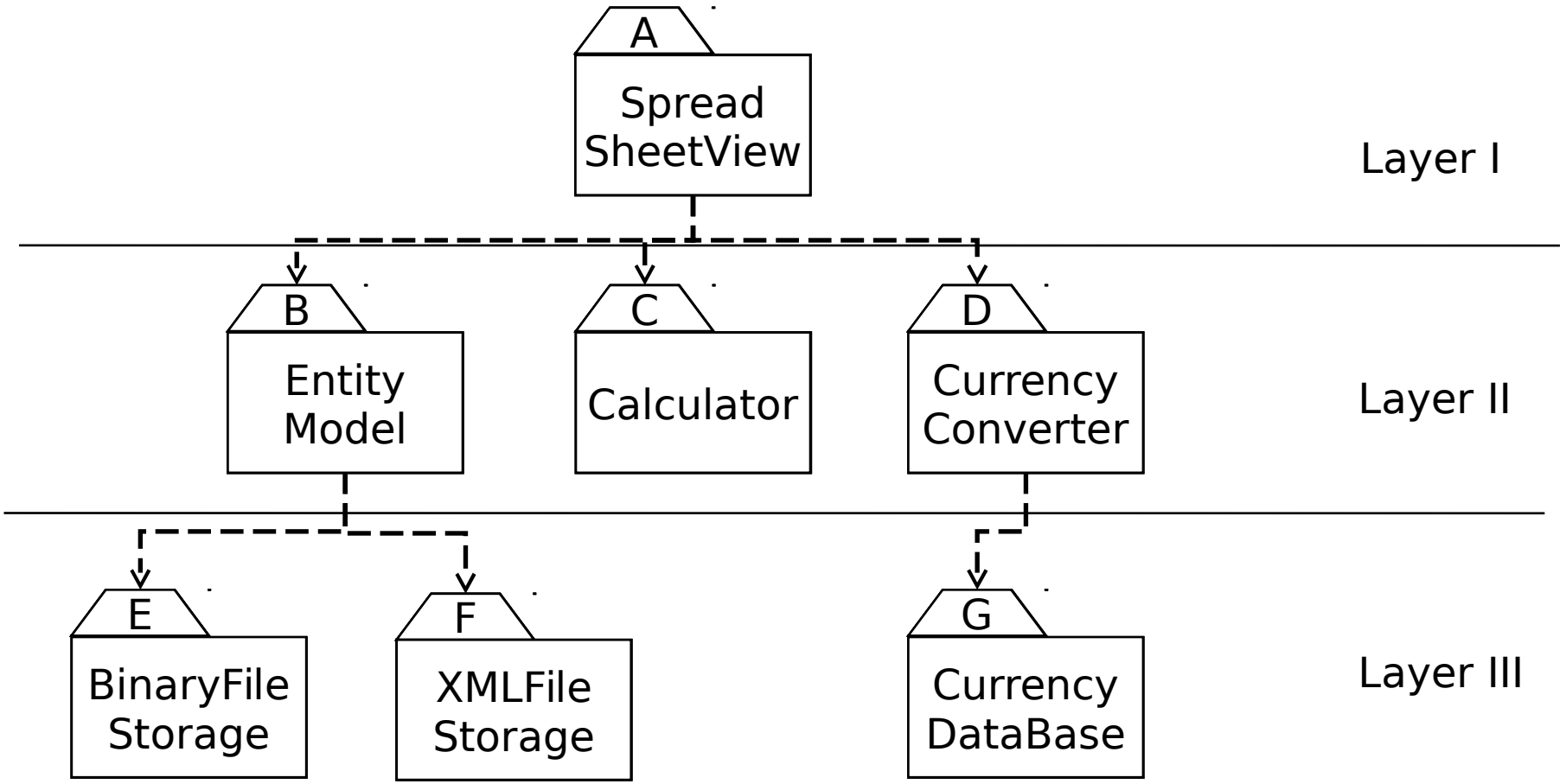
Stub:

- Um componente de teste, `TestedUnit`
- Implementação parcial
- Retorna valores fictícios

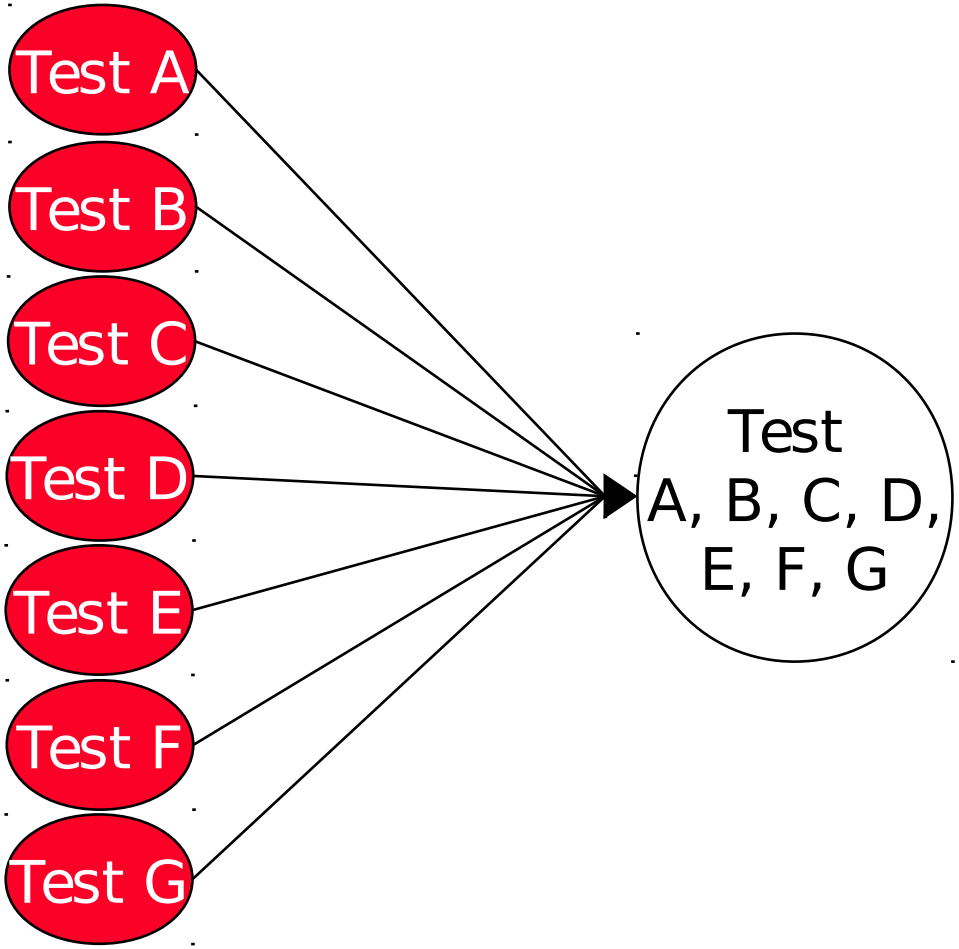


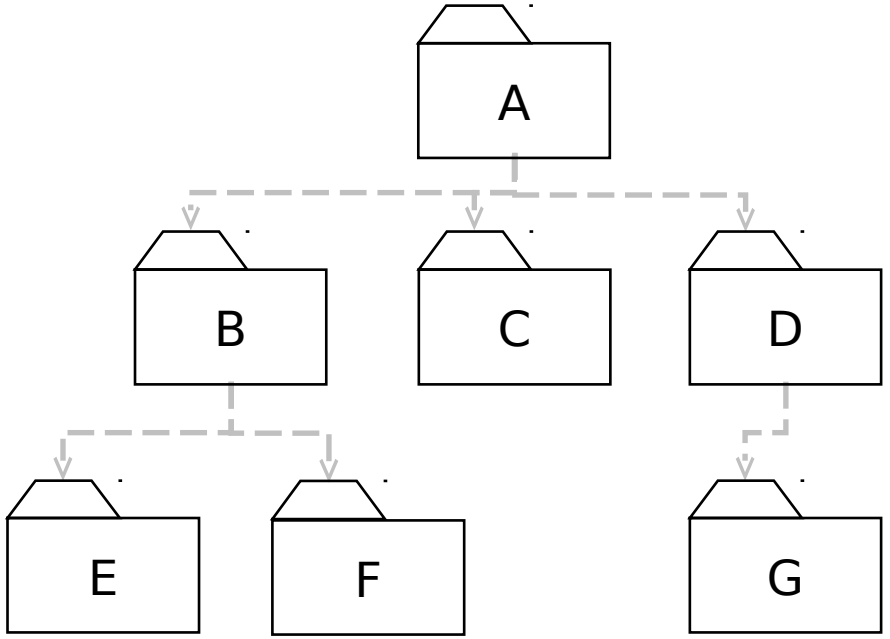
Exemplo: Projeto com 3 camadas

(Planilha)



Abordagem Big-Bang

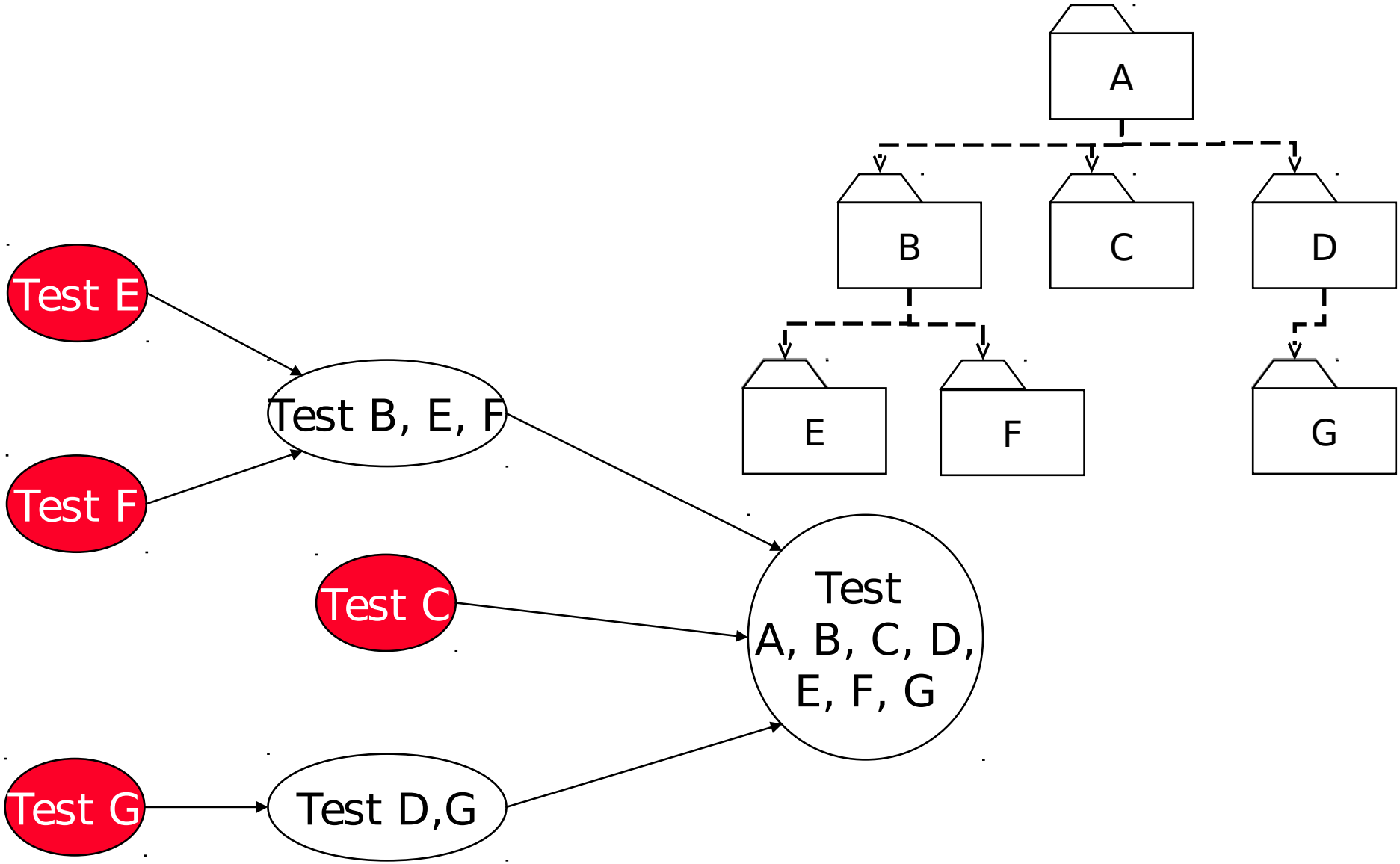




Estratégia de testes bottom-up

- Os subsistemas na camada mais baixa da hierarquia de chamada são testados individualmente
- Em seguida, os próximos subsistemas são testados que então chamam os subsistemas previamente testados
- Este processo é repetido até que todos os subsistemas estejam incluídos
- Drivers são necessários (para simular as camadas mais altas enquanto não se chega lá).

Integração bottom-up



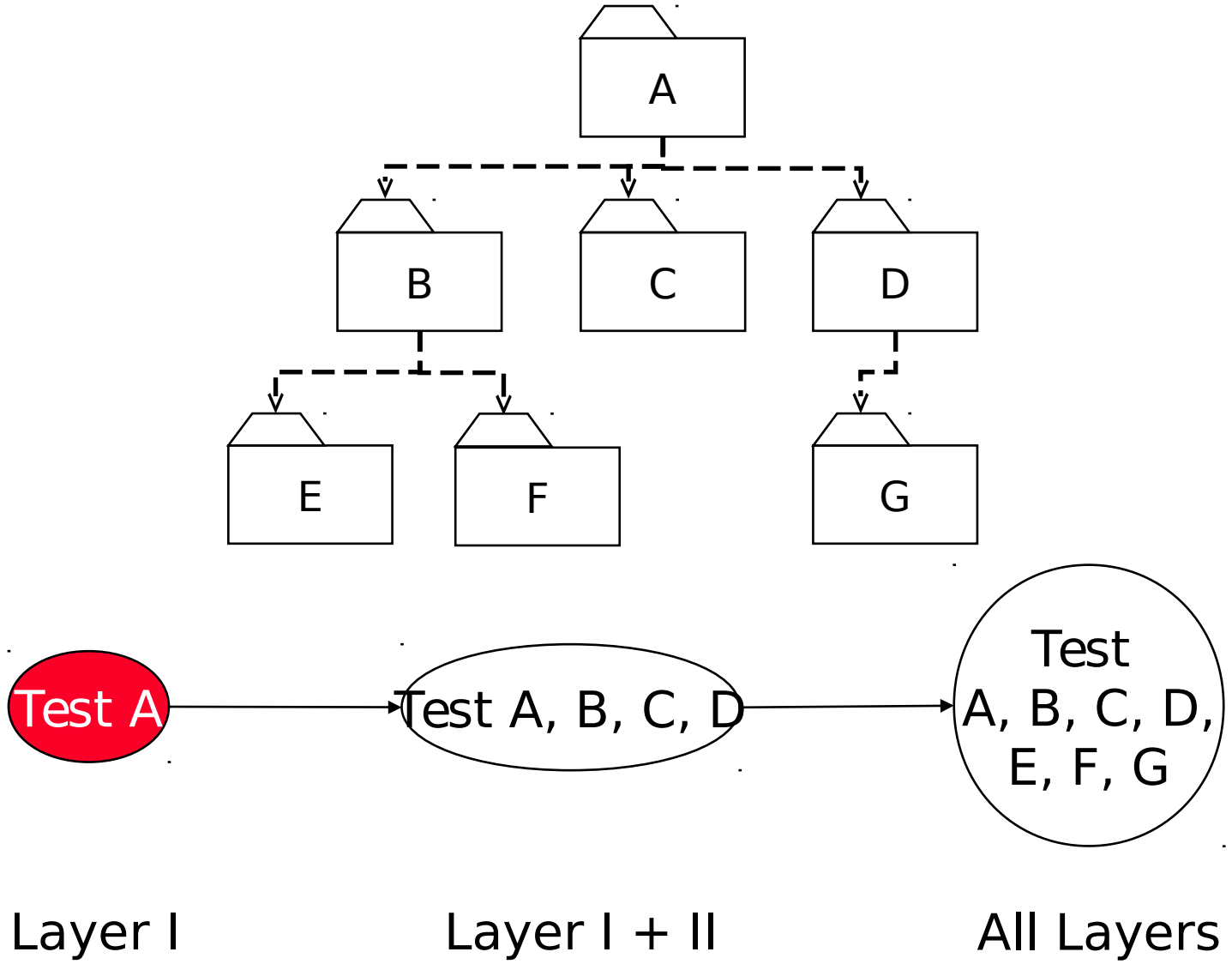
Integração bottom-up

- **Contra:**
  - Testa o subsistema mais importante (interface de usuário) por último!
  - Drivers são necessários
- **Pró**
  - Sem necessidade de stubs
  - Útil para testes de integração dos seguintes sistemas
    - Sistemas orientados a objetos
    - Sistemas de tempo real
    - Sistemas com requisitos rigorosos de desempenho.

Estratégia de testes top-down

- Teste a camada superior ou o subsistema de controle primeiro
- Em seguida, combina todos os subsistemas que são chamados pelos subsistemas testados e testa a coleção resultante de subsistemas
- Isso é feito até que todos os subsistemas são incorporadas ao teste
- Stubs são necessários para fazer o teste.

Integração top-down



Integração top-down

Pró

- Os casos de teste pode ser definidos em termos da funcionalidade do sistema (requisitos funcionais)
- Não há necessidade de drivers

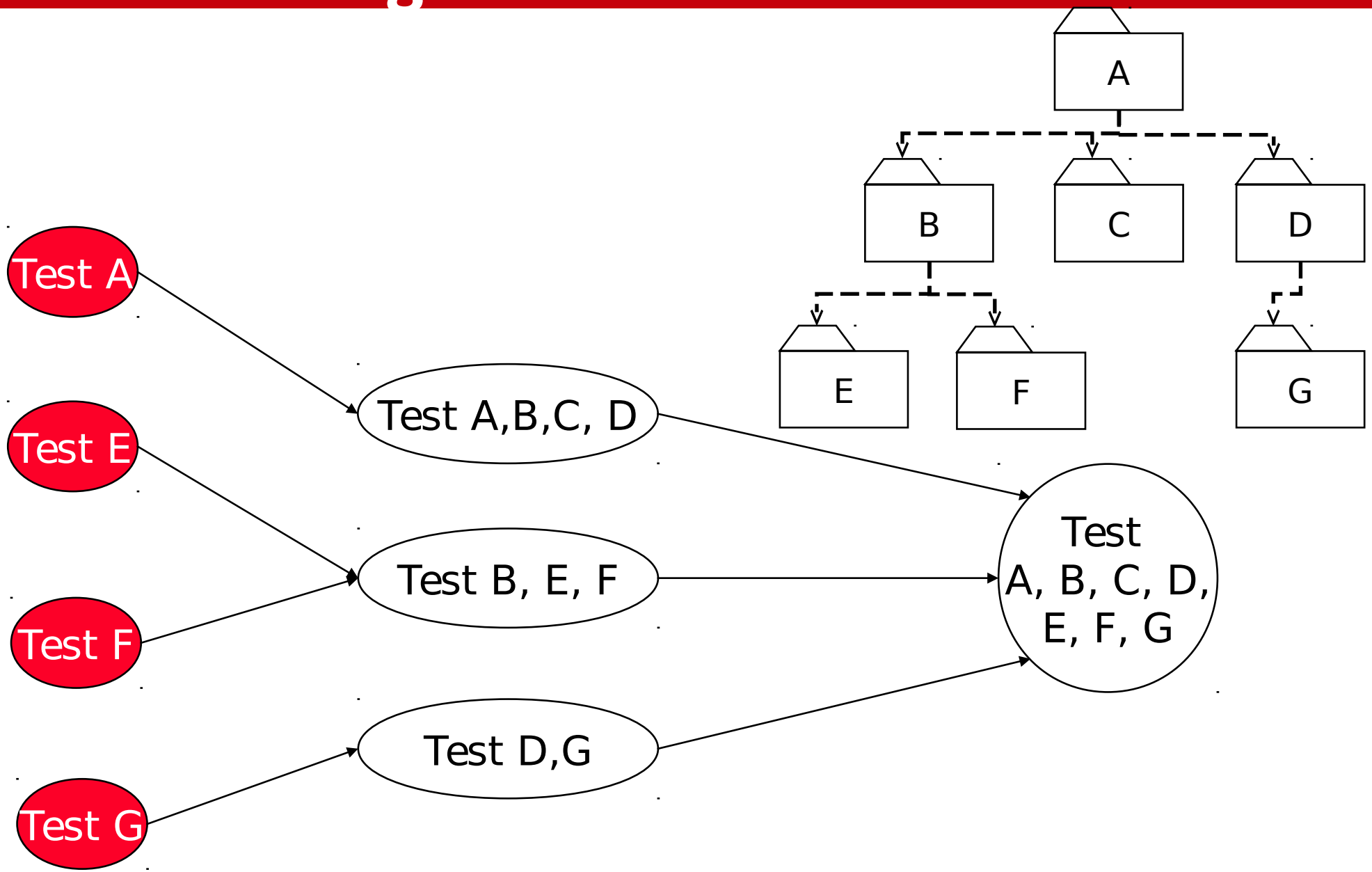
Contra

- Escrever stubs é difícil: stubs devem permitir todas as condições possíveis para ser testada.
- Pode ser necessário grande um grande número de stubs, especialmente se o nível mais baixo do sistema contém muitos métodos.
- Algumas interfaces não são testadas separadamente....

Estratégia de testes sanduíche

- **Combina as estratégias top-down e bottom-up**
- **O sistema é visto como tendo três camadas**
  - Uma camada de alvo no meio
  - A camada acima do alvo
  - Uma camada abaixo do alvo
- **O teste converge na camada alvo.**

Estratégia de testes sanduíche



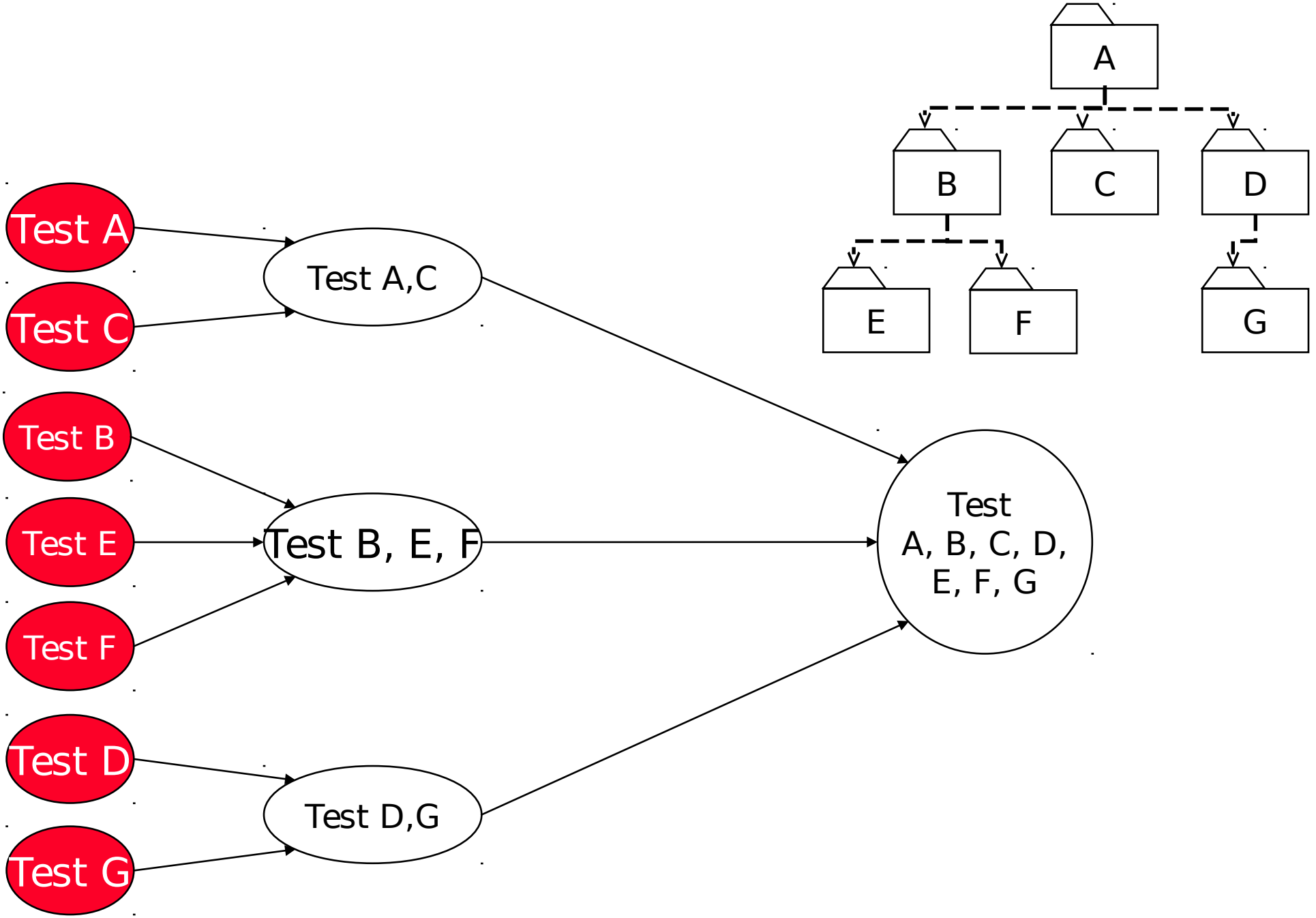
Estratégia de testes sanduíche

- Testes superior e inferior da camada podem ser feitos em paralelo
- Problema: Não testar os subsistemas individuais e suas interfaces completamente antes de integração
- Solução: Modificado estratégia de ensaio sanduíche

Estratégia de testes sanduíche modificada

- **Testar em paralelo:**
  - Camada média com drivers e stubs
  - Camada superior com stubs
  - Camada inferior com drivers
- **Testar em paralelo:**
  - Camada superior acessando camada média (camada superior substitui drivers)
  - Inferior acessada pela camada média (camada inferior substitui stubs).

Estratégia de testes sanduíche modificada

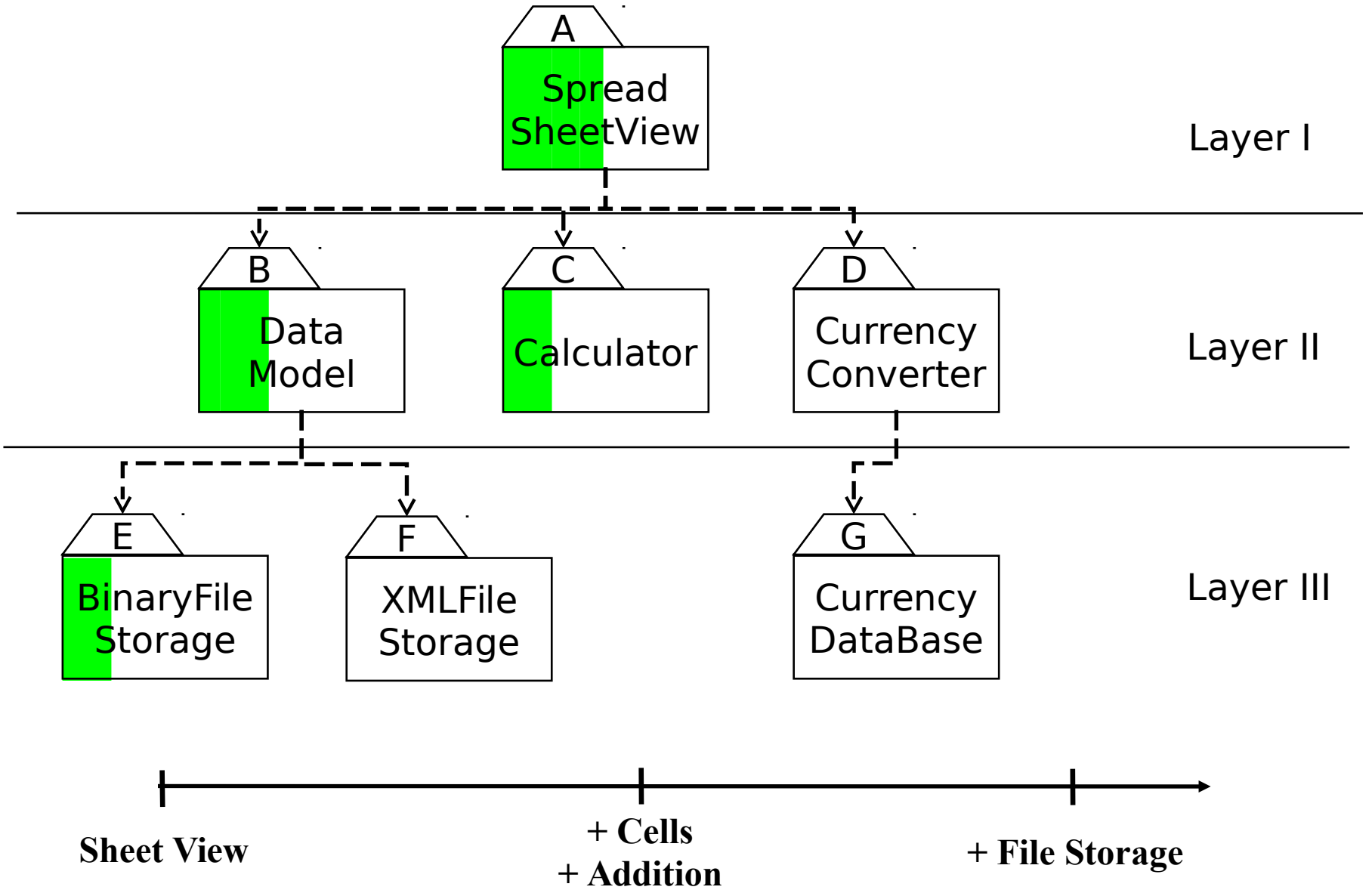


Estratégia de testes contínuos

- **Construção contínua:**
  - Construir desde o primeiro dia
  - Testar desde o primeiro dia
  - Integrar desde o primeiro dia

  ⇒ O sistema sempre está executável
- **Requer suporte integrado de ferramentas:**
  - Servidor de construção contínua
  - Testes automatizados com alta cobertura
  - Ferramenta de apoio para refatoração
  - Gerenciamento de configuração de software
  - Rastreamento de problemas.

Estratégia de testes contínuos

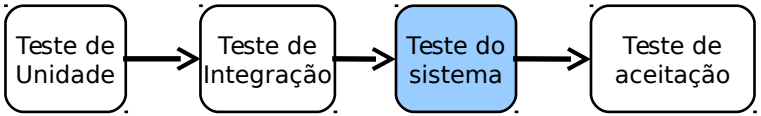


Passos em testes de integração

1. Com base na estratégia de integração, selecionar um componente a ser testado. Realizar teste de unidade de todas as classes do componente.
2. Coloque o componente selecionado junto; fazer qualquer correção preliminar necessária para fazer o teste de integração operacional (drivers, stubs)
3. Testar requisitos funcionais: Definir casos de teste que exercitam todos os usos casos com o componente selecionado
4. Testar a decomposição do subsistema: definir casos de teste que exercitam todas as dependências
5. Testar requisitos não-funcionais: Executar testes de desempenho
6. Manter registros de casos de testes e atividades de teste.
7. Repetir os passos 1-7 até que o sistema completo seja testado.

O principal objetivo do teste de integração é identificar falhas com a configuração do componente (atual).

Teste do sistema



- **Testes funcionais**
  - Valida os requisitos funcionais
- **Testes de desempenho**
  - Valida os requisitos não funcionais
- **Testes de aceitação**
  - Valida as expectativas do cliente

Testes funcionais

Meta: Testar a funcionalidade do sistemas

- **Os casos de teste são projetados a partir do documento de análise de requisitos (melhor: manual) e centrado em torno de requisitos e funções-chave (casos de uso)**
- **O sistema é tratado como uma caixa preta**
- **Casos de teste de unidade podem ser reutilizados, mas novos casos de teste devem ser desenvolvidos também.**

Testes de desempenho

Meta: Tentar violar requisitos não funcionais

- **Testar como o sistema se comporta quando sobrecarregado**
  - Pode-se identificar gargalos?
- **Experimentar ordens de execução não usuais:**
  - Chamar `receive()` antes de `send()`
- **Verificar a resposta do sistema a grandes volumes de dados**
  - Se o sistema é projetado para manipular 1000 itens, tente com 1001
- **Qual é a quantidade de tempo gasto em diferentes casos de uso?**
  - Os casos mais comuns são executados mais a tempo?

Tipos de teste de desempenho

Testes de estresse

Estressa os limites do sistema

Teste de volume

Testa o que acontece se grandes quantidades de dados são manipuladas

Teste de configuração

Testa diversas configurações de software e hardware

Teste de compatibilidade

Testa compatibilidade retroativa com sistemas existentes

Teste de temporização

Avalia tempos de resposta e tempos para executar funções

Testes de segurança

Tenta violar requisitos de segurança

Testes de ambiente

Testar tolerâncias ao calor, umidade, movimento etc

Testes de qualidade

Testar confiabilidade, manutenabilidade e disponibilidade

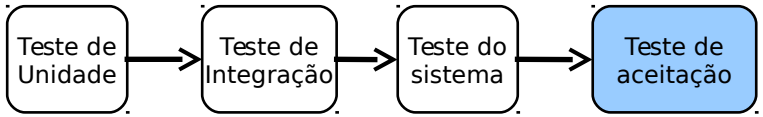
Testes de recuperação

Testar a resposta do sistema em resposta à presença de erros ou perda de dados

Teste de fatores humanos

Teste com usuários finais.

Testes de aceitação



Meta: demonstrar que o sistema está pronto para uso operacional

- A escolha dos testes é feita pelo cliente
- Vários testes podem ser obtidos a partir dos testes de integração
- Os testes de aceitação são realizados pelo cliente e não pelo desenvolvedor

Teste alfa:

- O cliente utiliza o software no ambiente de desenvolvedor.
- Software utilizado com ajuste controlado, com desenvolvedor sempre pronto para consertar bugs.

Teste beta:

- Conduzido no ambiente do cliente, sem a presença do desenvolvedor.
- Exercita-se o software em um ambiente realístico

Atividades de teste

Estabelecer os objetivos do teste

Projetar os casos de teste

Escrever os casos de teste

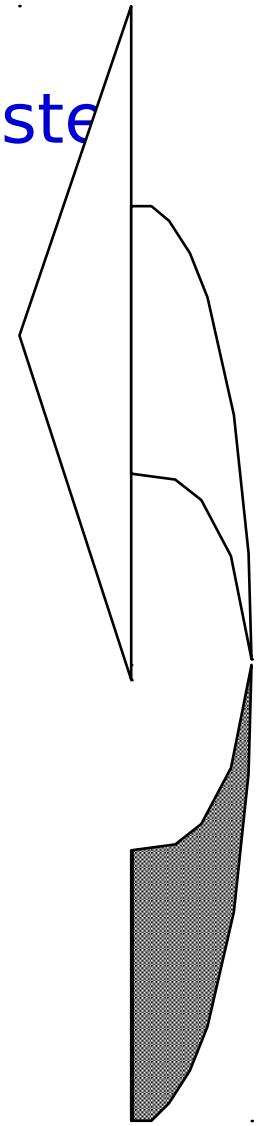
Testar os casos de teste

Executar os testes

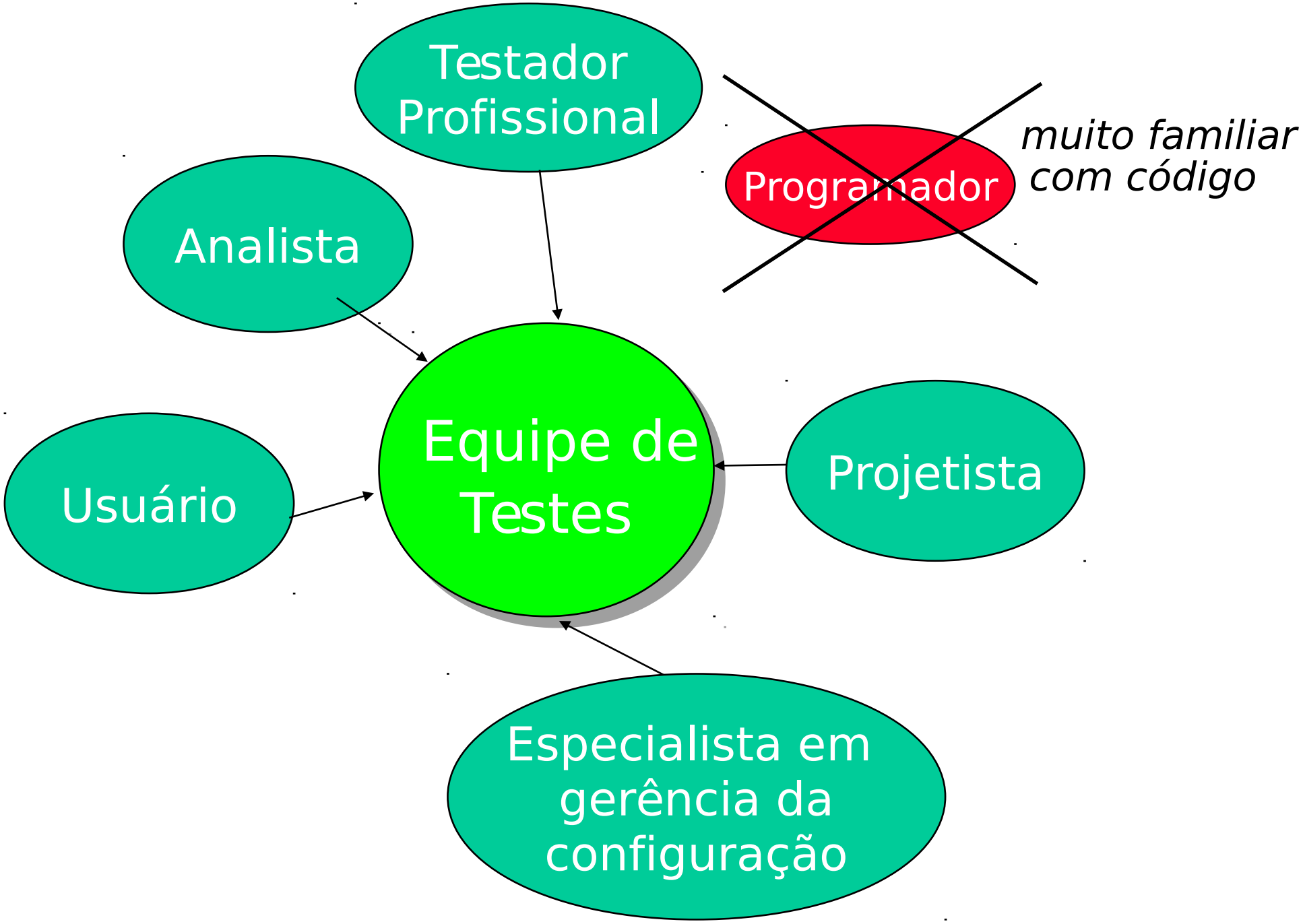
Avaliar os resultados de teste

Alterar o sistema

Executar testes de regressão



Equipe de testes



Quatro passos para testes

1. Selecionar o que precisa ser testado

- Análise: Completude dos requisitos
- Projeto: Coesão
- Implementação: Código fonte

2. Decidir como o teste será realizado

- Revisão ou inspeção de código
- Provas (Projeto por contrato)
- Caixa-preta, caixa-branca,
- Selecionar estratégia de intergração de testes (big bang, bottom up, top down, sanduíche)

3. Desenvolver casos de teste

- Um caso de teste é um conjunto de dados de teste ou situações que serão utilizadas para exercitar a unidade (classe, subsistema, sistema) sendo testado ou sobre o atributo sendo mensurado.

4. Criar o oráculo de teste

- Um oráculo contém resultados previstos para um conjunto de casos de teste
- O oráculo de teste tem que ser escrito antes do teste ser iniciado!

Guia para a seleção de dados de teste

Utilizar *conhecimento de análise* sobre requisitos funcionais (teste caixa-preta):

- Casos de uso
- Dados de entrada esperados
- Dados de entrada inválidos

Utilizar *conhecimento de projeto* sobre a estrutura do sistema - algoritmos, estruturas de dados (teste caixa-branca):

- Estruturas de controle
  - Ramificações, laços ...
- Estruturas de dados
  - Testar registros, arranjo, ...

Utilizar *conhecimento de implementação* sobre algoritmos e estruturas de dados:

- Forçar divisão por zero
- Tentar ultrapassar espaço de índices de variáveis indexadas.

Referências bibliográficas

- BRUEGGE, B.; DUTOIT, A. H. **Object-Oriented Software Engineering Using UML, Patterns and Java**. Upper Saddle River, N.J.: Pearson Education, 2010.