Detalhamento de um Framework Horizontal: JUNIT

- Material baseado no artigo A Cook's Tour
- Ver também o artigo Test Infected: Programmers Love Writing Tests
- Um excelente exemplo da aplicação de vários Design Patterns
  - Command, Template Method, Collecting Parameter, Factory Method, Iterator, Adapter, Pluggable Selector, Composite

Objetivos

- Desenvolver um framework que programadores de fato usarão para escrever testes de unidade
  - Testes de unidade são testes de classes individuais
- Exigir o mínimo do programador
- Evitar duplicação de esforços ao escrever testes
- Permitir escrever testes que retenham seu valor ao longo do tempo

Um exemplo do uso do framework

- Para testar uma classe, criamos uma classe de testes correspondente
- O framework JUNIT nos ajuda a criar essa classe de testes
- O mínimo que se deve fazer é o seguinte:
  - Herdar da classe TestCase do framework
  - Fornecer um construtor recebendo como parâmetro o nome do teste a fazer
  - Fornecer um ou mais métodos com o nome test...() sem parâmetros e sem valor de retorno
    - Cada método de testes testa algo usando o método assertTrue(msg, condição)
  - Fornecer um método estático suite() que informa quais são os métodos de testes

- O framework fornece uma interface gráfica para executar os testes
  - Basta informar o nome da classe de testes
- O que foi mencionado acima é o mínimo necessário para usar o framework
- Para ter um pouco mais de controle, podemos fazer override dos seguintes métodos, se necessário:
  - setUp()
    - É um hook method chamado antes de cada método de teste
      - Para manter testes independentes
    - Usado para construir um contexto para um teste
      - Exemplo: abrir uma conexão de banco de dados
  - tearDown()
    - É um hook method chamado depois de cada método de teste
    - Usado para desfazer o que setUp() faz
      - Exemplo: fechar uma conexão de banco de dados
  - runTest()
    - Para controlar a execução de um teste particular
    - É raro fazer override disso

Exemplo do uso de JUNIT

- Queremos testar a seguinte classe:

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal da Paraíba
 *
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 * Não redistribuir sem permissão.
 */
```

```
package p1.aplic.cartas;

import java.util.*;
import java.lang.Math.*;

/**
 * Um baralho comum de cartas.
 * Num baralho comum, tem 52 cartas:
 * 13 valores (AS, 2, 3, ..., 10, valete, dama, rei)
 * de 4 naipes (ouros, espadas, copas, paus).
 *
 * @author Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.1
 * <br>
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 */
public class Baralho {
  /**
   * O baralho é armazenado aqui.
   * É protected porque alguns baralhos subclasses dessa cla
   * poderão talvez ter que mexer diretamente aqui
   * para construir baralhos especiais.
   */
  protected Vector baralho;

  /**
   * Construtor de um baralho comum.
   */
  public Baralho() {
    // Usa um Vector para ter um iterador facilmente
    baralho = new Vector();
    // enche o baralho
    for(int valor = menorValor(); valor <= maiorValor(); val
      for(int naipe = primeiroNaipe(); naipe <= últimoNaipe(
        // chama criaCarta e não "new" para poder fazer over
        // de criaCarta em baralhos de subclasses e
        // criar classes diferentes.
        baralho.add(criaCarta(valor, naipe));
      }
    }
  }

  /**
   * Cria uma carta para este baralho.
   * @param valor O valor da carta a criar.
   * @param naipe O naipe da carta a criar.
   * @return A carta criada.
   */
```

```
protected Carta criaCarta(int valor, int naipe) {
  return new Carta(valor, naipe);
}

/**
 * Recupera o valor da menor carta possível deste baralho.
 * É possível fazer um laço de menorValor() até maiorValor
 * para varrer todos os valores possíveis de cartas.
 * @return O menor valor.
 */
public int menorValor() {
  return Carta.menorValor();
}

/**
 * Recupera o valor da maior carta possível deste baralho.
 * É possível fazer um laço de menorValor() até maiorValor
 * para varrer todos os valores possíveis de cartas.
 * @return O maior valor.
 */
public int maiorValor() {
  return Carta.maiorValor();
}

/**
 * Recupera o "primeiro naipe" das cartas que podem estar
 * no baralho.
 * Ser "primeiro naipe" não significa muita coisa,
 * já que naipes não tem valor
 * (um naipe não é menor ou maior que o outro).
 * Fala-se de "primeiro naipe" e "último naipe" para poder
 * fazer um laço de primeiroNaipe() até últimoNaipe() para
 * todos os naipes possíveis de cartas.
 * @return O primeiro naipe.
 */
public int primeiroNaipe() {
  return Carta.primeiroNaipe();
}

/**
 * Recupera o "último naipe" das cartas que podem estar
 * no baralho.
 * Ser "último naipe" não significa muita coisa,
 * já que naipes não tem valor
 * (um naipe não é menor ou maior que o outro).
 * Fala-se de "primeiro naipe" e "último naipe" para poder
 * fazer um laço de primeiroNaipe() até últimoNaipe() para
 * todos os naipes possíveis de cartas.
```

```
 * @return O primeiro naipe.
 */
public int últimoNaipe() {
  return Carta.últimoNaipe();
}

/**
 * Recupera o número de cartas atualmente no baralho.
 * @return O número de cartas no baralho.
 */
public int númeroDeCartas() {
  return baralho.size();
}

/**
 * Recupera um iterador para poder varrer todas
 * as cartas do baralho.
 * @return Um iterador do baralho.
 */
public Iterator iterator() {
  return baralho.iterator();
}

/**
 * Baralha (traça) o baralho.
 */
public void baralhar() {
  int posição;
  for(posição = 0; posição < númeroDeCartas() - 1; posição
    // escolhe uma posição aleatória entre posição e númer
    int posAleatória = posição +
                       (int)((númeroDeCartas() - posição)
                             Math.random());
    // troca as cartas em posição e posAleatória
    Carta temp = (Carta)baralho.get(posição);
    baralho.set(posição, baralho.get(posAleatória));
    baralho.set(posAleatória, temp);
  }
}

  /**
   * Retira uma carta do topo do baralho e a retorna.
   * A carta é removida do baralho.
   * @return A carta retirada do baralho.
   */
public Carta pegaCarta() {
  if(númeroDeCartas() == 0) return null;
  return (Carta)baralho.remove(númeroDeCartas()-1);
```

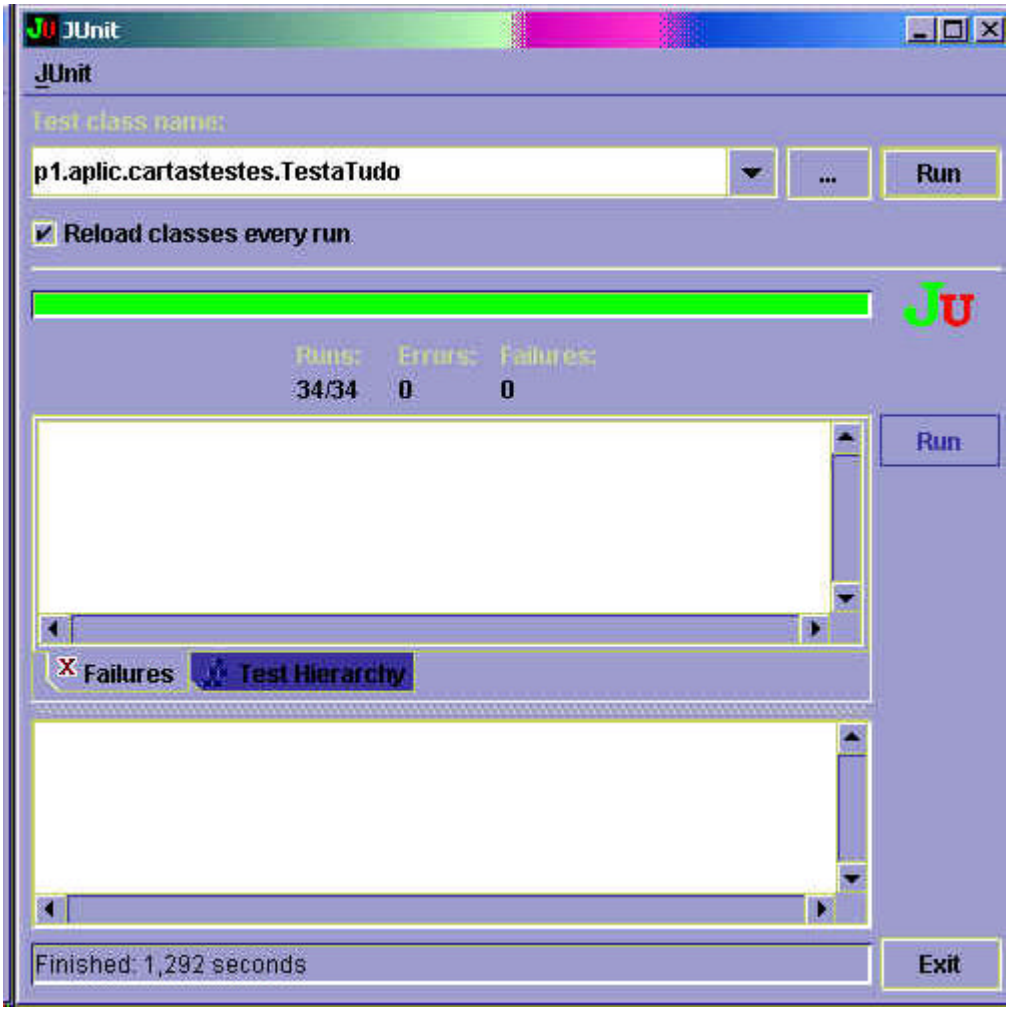
```
  }
}
```

- Uma possível classe de teste segue abaixo
  - A cor vermelha indica classes, métodos, etc. do framework

```
package p1.aplic.cartastestes;

import junit.framework.*;
import p1.aplic.cartas.*;
import java.util.*;

/**
 * Testes da classe Baralho.
 *
 */
public class TestaBaralho extends TestCase {
  protected Baralho  b1; // fica intacto

  public TestaBaralho(String name) {
    super(name);
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
  protected void setUp() {
    b1 = new Baralho();
  }
  public static Test suite() {
    return new TestSuite(TestaBaralho.class);
  }
  public void testNúmeroDeCartas() {
    assertEquals(1, b1.menorValor());
    assertEquals(13, b1.maiorValor());
    assertEquals(52, b1.númeroDeCartas());
  }
  public void testBaralhoNovo() {
    assertTrue(baralhoEstáCompleto(b1));
  }
  public void testBaralhar() {
    Baralho b2 = new Baralho();
    b2.baralhar();
    assertTrue(baralhoEstáCompleto(b2));
  }
  public boolean baralhoEstáCompleto(Baralho b) {
```

```
    Vector cartasJáVistas = new Vector();
    Iterator it = b.iterator();
    while(it.hasNext()) {
      Carta c = (Carta)it.next();
      // vê se carta está ok
      int v = c.getValor();
      int n = c.getNaipe();
      assertTrue("Valor não ok",
                 v >= c.menorValor() && v <= c.maiorValor())
      assertTrue("Naipe não ok",
                 n >= c.primeiroNaipe() && n <= c.últimoNaip
      assertTrue("Carta já vista",
                 !cartasJáVistas.contains(c));
      cartasJáVistas.add(c);
    }
    return cartasJáVistas.size() == 52;
  }
  public void testPegaCarta() {
    Vector cartasJáVistas = new Vector();
    Baralho b3 = new Baralho();
    Carta c;
    while((c = b3.pegaCarta()) != null) {
      // vê se carta está ok
      int v = c.getValor();
      int n = c.getNaipe();
      assertTrue("Valor não ok",
                 v >= c.menorValor() && v <= c.maiorValor())
      assertTrue("Naipe não ok",
                 n >= c.primeiroNaipe() && n <= c.últimoNaip
      assertTrue("Carta já vista",
                 !cartasJáVistas.contains(c));
      cartasJáVistas.add(c);
    }
    assertEquals("Baralho não vazio", 0, b3.númeroDeCartas()
  }
}
```

- Execute o JUNIT via applet aqui
  - Sorry! Não funciona porque o JUNIT tenta ler um arquivo local (de preferências) e o mecanismo de segurança do applet não deixa. Um dia, vou ver se conserto essas coisas ...
- Alternativamente, execute o teste da classe (comando testa.bat do diretório deste arquivo)
  - O resultado deve ser algo assim:



O projeto de JUNIT

- Começaremos do zero e usaremos patterns para explicar o framework até montar a arquitetura final

O início: TestCase

- Para manipular testes facilmente, eles devem ser objetos e não:
  - Comandos de impressão
  - Expressões de depurador
  - Scripts de testes
- Isso *concretiza* os testes para que possam reter seu valor ao longo do tempo
- O pattern "Command" é útil aqui
  - O padrão Command permite encapsular um pedido (de teste) como objeto
  - Command pede para criar um objeto e fornecer um

método execute()
- Nós chamaremos este método de run()

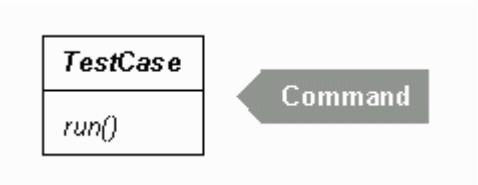
- A definição da classe TestCase segue:

```
public abstract class TestCase implements Test {
  ...
}
```

- É declarada "public abstract" porque será reusada através de herança
- Por enquanto, esqueça da implementação da interface Test
- Cada teste é criado com um nome para que você possa identificar o teste quando falha

```
public abstract class TestCase implements Test {
  private final String fName;
  public TestCase(String name) {
    fName= name;
  }

  public abstract void run();
  ...
}
```

- A arquitetura até agora é a seguinte (com indicação do design pattern usado):



Completando a informação: run()

- Precisamos de um lugar conveniente para colocar o código de testes
- A parte comum para todos os testes deve ser fornecida pelo framework
- O que cada teste faz:
  - Cria um contexto para o teste
  - Executa código usando o contexto

- Verifica algum resultado
- Limpa o contexto

- Testes terão seu valor maximizado se um teste não afetar o outro
- Portanto, os passos acima devem ser feito para cada teste
- Podemos assegurar que essas etapas serão sempre feitas usando o design pattern chamado Template Method
  - Template Method define o esqueleto de um algoritmo em uma operação, deixando certos passos para subclasses. A estrutura do algoritmo não muda
- A execução dos passos não muda mas a forma de fazer cada passo será definida pelas subclasses
- O template method segue abaixo:

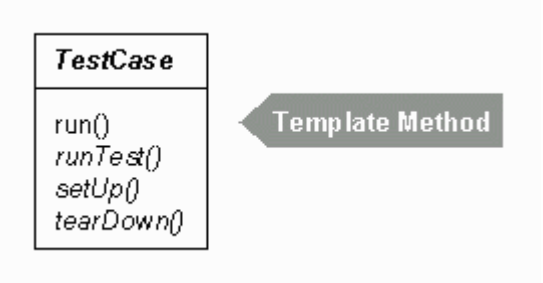
```
public abstract class TestCase implements Test {
  // ...
  public void run() {
    setUp();
    runTest();
    tearDown();
  }
}
```

- A implementação default dos métodos faz nada:

```
public abstract class TestCase implements Test {
  // ...
  protected void runTest() {
    // na realidade, o framework oferece uma implementação d
    // que veremos abaixo
  }

  protected void setUp() { // hook method
  }

  protected void tearDown() { // hook method
  }
}
```

- Já que setUp() e tearDown() sofrerão override nas

subclasses, elas são declaradas como protected
- A arquitetura até agora:



Obtendo resultados dos testes: TestResult

- Para cada teste, queremos:
  - Um resumo supercondensado sobre o que funcionou
  - Detalhes do que não funcionou
- Usaremos o pattern Collecting Parameter da coleção "Smalltalk Best Practice Patterns" do Guru Kent Beck
  - Padrões quase tão famosos quanto os padrões do Gang of Four
- O padrão sugere que, quando você precisa colecionar resultados obtidos com vários métodos, adicione um parâmetro ao método e passe um objeto que vai colecionar os resultados para você
- Criamos um novo objeto, da classe TestResult, para juntar os resultados dos testes executados
  - Esta versão simplificada de TestResult apenas conta os testes executados

```
public class TestResult {
  protected int fRunTests;
  public TestResult() {
    fRunTests= 0;
  }
}
```

- Para usar o objeto:
  - Adicionamos um parâmetro à chamada TestCase.run() para coletar resultados
  - Aqui, o "resultado" é só contar testes

```
public abstract class TestCase implements Test {
  // ...
  public void run(TestResult result) {
    result.startTest(this);
    setUp();
    runTest();
    tearDown();
  }
}
```

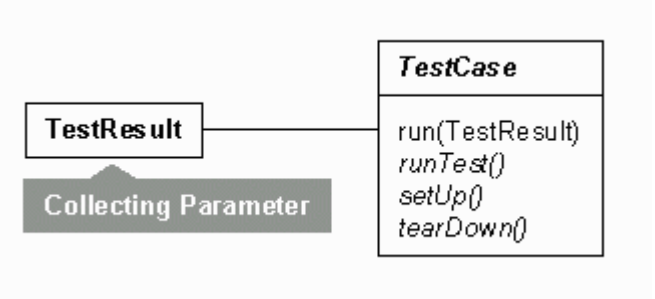
- Assim, TestResult pode contar os testes executados:

```
public class TestResult {
  public synchronized void startTest(Test test) {
    fRunTests++;
  }
}
```

- O método startTest() é synchronized para que TestResult possa colecionar resultados de vários testes, mesmo que executem em threads diferentes
- Para manter a interface externa simples de TestCase, criamos uma versão de run() sem parâmetros que cria seu próprio TestResult

```
public abstract class TestCase implements Test {
  // ...
  public TestResult run() {
    TestResult result= createResult();
    run(result);
    return result;
  }
  protected TestResult createResult() {
    return new TestResult();
  }
}
```

- Observe o uso do pattern Factory Method
- A arquitetura até agora:



- Testes podem falhar de forma prevista e de forma não prevista (freqüentemente espetacular!)
- Portanto, JUNIT distingue entre *falhas* e *erros*
  - Falhas são descobertas com uma exceção AssertionFailedError
  - Erros são resultados de quaisquer outras exceções

```
public abstract class TestCase implements Test {
  // ...
  public void run(TestResult result) {
    result.startTest(this);
    setUp();
    try {
      runTest();
    } catch (AssertionFailedError e) {
      result.addFailure(this, e);
    } catch (Throwable e) {
      result.addError(this, e);
    } finally {
      tearDown();
    }
  }
}
```

- A exceção AssertionFailedError é lançada pelo método assertTrue() da classe TestCase
- Há várias versões de assertXXX()
- Um exemplo simples segue:

```
public abstract class TestCase implements Test {
  // ...
  protected void assertTrue(boolean condition) {
    if(!condition) {
      throw new AssertionFailedError();
    }
  }
}
```

- Os métodos de TestResult que colecionam os erros são mostrados abaixo:

```
public class TestResult {
  // ...
  public synchronized void addError(Test test, Throwable t)
    fErrors.addElement(new TestFailure(test, t));
  }

  public synchronized void addFailure(Test test, Throwable t
    fFailures.addElement(new TestFailure(test, t));
  }
}
```

- A classe TestFailure é uma classe interna que mantém o teste e a exceção juntos num objeto:

```
public class TestFailure {
  protected Test fFailedTest;
  protected Throwable fThrownException;
}
```

- O uso canônico do pattern Collecting Parameter exigiria que cada método de teste da classe de testes passasse um parâmetro com o TestResult
- Já que estamos usando exceções para indicar problemas, podemos evitar essa passagem de parâmetros (que poluiriam as interfaces) da seguinte forma:
  - run() chama o próprio TestResult para executar o teste e TestResult chama os métodos de testes e captura as exceções
  - Desta forma, o programador que bola os testes não tem que se preocupar com TestResult, que fica completamente escondido
- O teste abaixo mostra como o método de teste nada sabe sobre TestResult

```
public abstract class MeuTeste extends TestCase {
  // ...
  public void runTest() {
    assertEquals(1, b1.menorValor());
    assertEquals(13, b1.maiorValor());
```

```
    assertEquals(52, b1.númeroDeCartas());
  }
}
```

- JUNIT provê várias versões de TestResult:
  - A implementação default conta falhas e erros e junta os resultados
  - TextTestResult junta os resultados e os apresenta em forma textual
  - UITestResult junta os resultados e os apresenta em forma gráfica
  - O programador poderia fornecer um HTMLTestResult para reportar os resultados em HTML
  - Porém, eu [Jacques] acho que poderíamos melhorar isso usando um único TestResult e usar o padrão Observer para conectá-lo à interface do usuário
    - Talvez isso tenha sido feito numa versão mais recente de JUNIT

Não crie subclasses estúpidas: TestCase revisitado

- Usamos o pattern Command para representar um teste
- O pattern precisa de um método execute() (chamado run(), aqui)
- Podemos chamar diferentes implementações do comando através dessa interface simples
- Precisamos de uma interface genérica para executar os testes
- Porém, todos os testes são implementados como métodos diferentes de uma mesma classe de testes
  - Para evitar a proliferação de classes de testes
- Como chamar todos esses testes (com nomes diferentes) usando uma interface única?
- O próximo problema a resolver é portanto de fazer todos os testes parecerem iguais ao chamador do teste
- O pattern Adapter parece se aplicar

- Adapter converte a interface de uma classe para uma outra interface que o cliente espera

- Há várias formas de fazer isso
  - Uma classe adaptadora que usa herança para adaptar a interface
  - Exemplo: override de runTest() para adaptar a interface e chamar outro método

```
public class TesteBaralhar extends TesteBaralho {
  public TesteBaralhar() {
    super("testBaralhar");
  }
  protected void runTest() {
    testBaralhar();
  }
}
```

- A forma acima é ruim para o programador pois exige a criação de muitas classes
- Em Java, podemos usar classes anônimas para evitar a proliferação de classes:

```
TestCase test = new TesteBaralho("testBaralhar") {
  protected void runTest() {
    testBaralhar();
  }
};
```

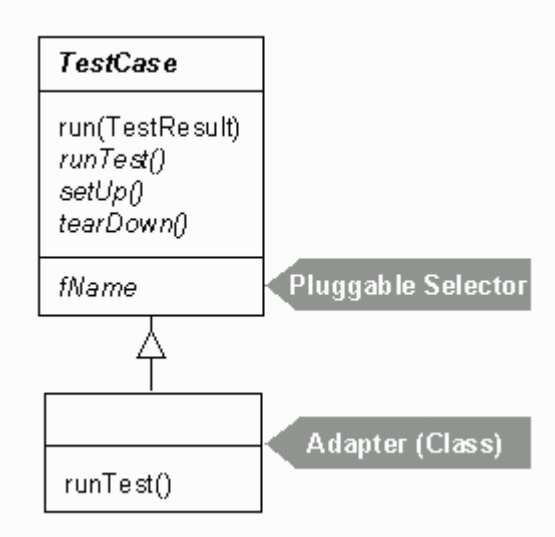
- Isso é muito mais conveniente que criar subclasses
- Uma outra forma se chama Pluggable Behavior
  - A idéia é usar uma classe única que pode ser parametrizada para executar lógica diferente sem subclasses
  - Uma forma de implementar Pluggable Behavior é de usar o Pluggable Selector
  - O Pluggable Selector é um atributo da classe que indica o método a chamar
  - Em Java, podemos usar o String representando o nome do método e usar a API de reflexão para fazer a chamada
    - Isso é chamado Very Late Binding

- Reflexão deve ser usada com cuidado! Não use sempre! É feio! (difícil de entender)

- JUNIT permite usar Pluggable Selector ou classes anônimas para implementar a adaptação: o programador escolhe
- Por default, o Pluggable Selector com reflexão já está implementado pelo framework

```
public abstract class TestCase implements Test {
  // ...
  protected void runTest() throws Throwable {
    Method runMethod = null;
    try {
      runMethod = getClass().getMethod(fName, new Class[0]);
    } catch (NoSuchMethodException e) {
      assertTrue("Method \"" + fName + "\" not found", false
    }
    try {
      runMethod.invoke(this, new Class[0]);
    }
    // catch InvocationTargetException and IllegalAccessExce
  }
}
```

- Todos os métodos de testes devem ser públicos, já que a API de reflexão do Java só permite achar métodos públicos
- A arquitetura até agora:



Um teste ou vários testes? TestSuite

- Muitos testes devem ser executados, não apenas um
- Até agora, JUNIT roda um único teste (com vários métodos de testes) e registra os resultados num TestResult
- Agora, queremos executar vários testes
- O problema é fácil de resolver se o chamador de um teste não souber se está executando um teste ou vários
- Qual o pattern a usar? Composite!
- Composite permite tratar objetos individuais e composições de objetos de forma uniforme
- Queremos testes individuais, suites de testes, suites de suites de testes, suites de suites de suites de testes, etc.
- O pattern Composite requer os seguintes participantes:
  - Component: declara a interface a usar para interagir com os testes
  - Composite: implementa essa interface e mantém uma coleção de testes
  - Folha: representa um caso de teste que obedece a interface Component
- Em Java, definimos uma interface para capturar o comportamento comum

```
public interface Test {
  public abstract void run(TestResult result);
}
```

- TestCase é a folha
- O Composite é a classe TestSuite

```
public class TestSuite implements Test {
  private Vector fTests= new Vector();
}
```

- O método run() de TestSuite delega o trabalho para seus filhos:

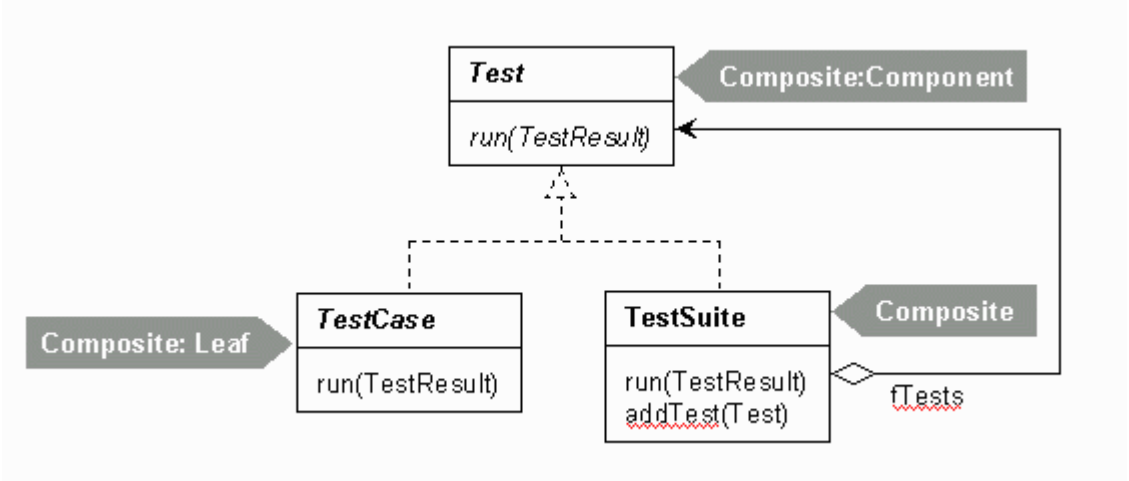
```
public class TestSuite implements Test {
  // ...
```

```
  public void run(TestResult result) {
    for(Iterator it = fTests.iterator(); it.hasNext(); ) {
      Test test = (Test)it.next();
      test.run(result);
    }
  }
}
```

- Observe o uso do padrão Iterator
- Finalmente, clientes podem adicionar testes a uma suite:

```
public class TestSuite implements Test {
  // ...
  public void addTest(Test test) {
    fTests.add(test);
  }
}
```

- Todo o código acima só depende da interface Test
- Já que TestCase e TestSuite implementam a interface Test, podemos fazer composições recursivas arbitrárias
  - Cada programador cria suas suites
  - Podemos rodar todas as suites dos programadores, criando uma nova TestSuite e adicionando as suites individuais
  - No fim, basta chamar run() da suite do topo para executar todos os testes
- A arquitetura com Composite:



- Segue um exemplo da criação de uma TestSuite:

```
public static Test suite() {
  TestSuite suite= new TestSuite();
  suite.addTest(new TestaBaralho("testNúmeroDeCartas"));
  suite.addTest(new TestaBaralho("testBaralhoNovo"));
  suite.addTest(new TestaBaralho("testBaralhar"));
  suite.addTest(new TestaBaralho("testPegaCarta"));
}
```

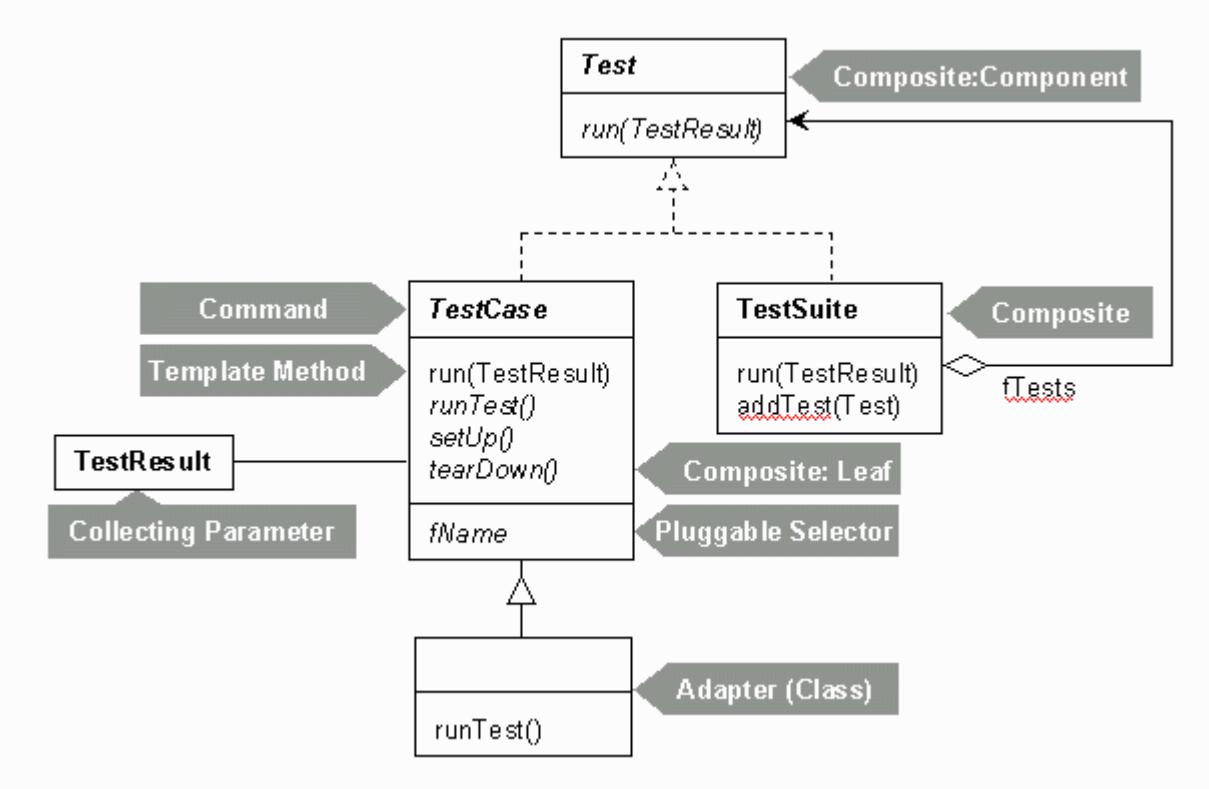
- Porém, isso ainda é muito trabalho para o programador
- Ao criar um novo teste, ele não vai rodar se você esquecer de colocá-lo na suite
- JUNIT provê um método de conveniência que recebe a classe como parâmetro e extrai todos os métodos de testes, criando uma suite para eles
  - O nome dos métodos de testes inicia com test... e eles não têm parâmetros
- O código acima pode ser simplificado para:

```
public static Test suite() {
  return new TestSuite(TestaBaralho.class);
}
```

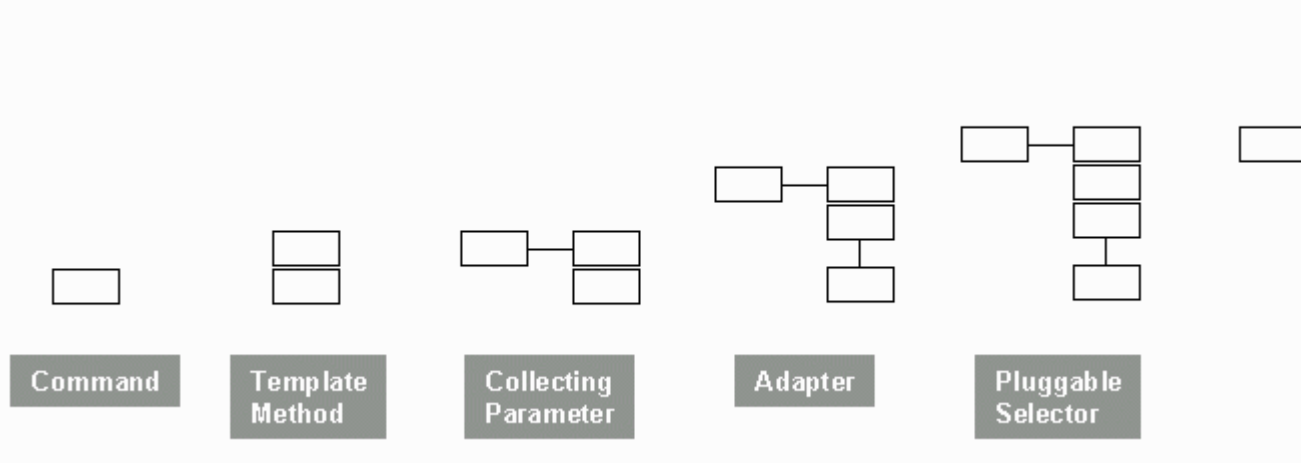
- Se desejado, a forma original pode ser usada para executar apenas um conjunto dos testes
- Dá para ver que os autores foram muito longe no sentido de dar pouco trabalho ao pobre programador que bola os testes

Resumo

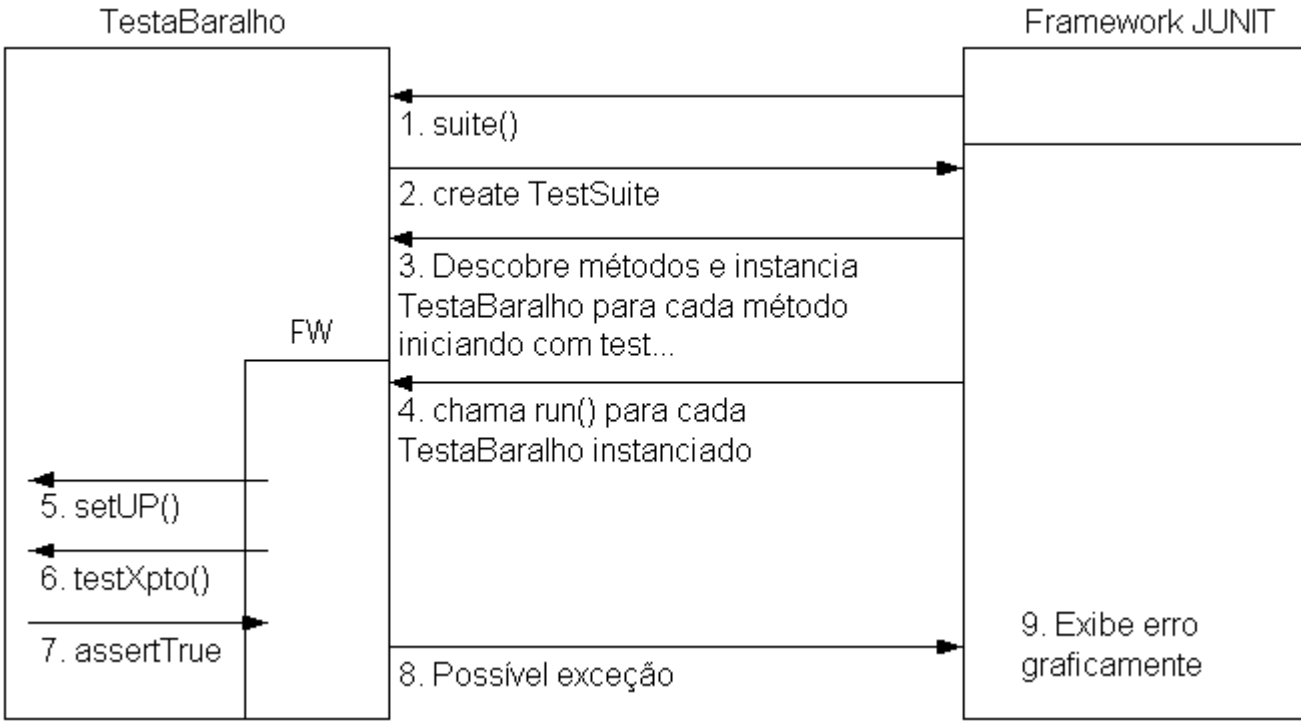
- A arquitetura final:



- Observe que a classe TestCase está envolvida em vários padrões
- Essa "densidade de padrões" é comum em bons projetos, depois de maduros (isto é, depois de bastante uso e refactoring!)
  - Os autores (Beck e Gamma) admitem que demorou um bocado até chegar ao design presente
  - Não saiu assim de primeira
- A figura abaixo mostra a evolução da arquitetura com a aplicação de padrões



- A figura abaixo mostra iterações entre classes de testes e o framework

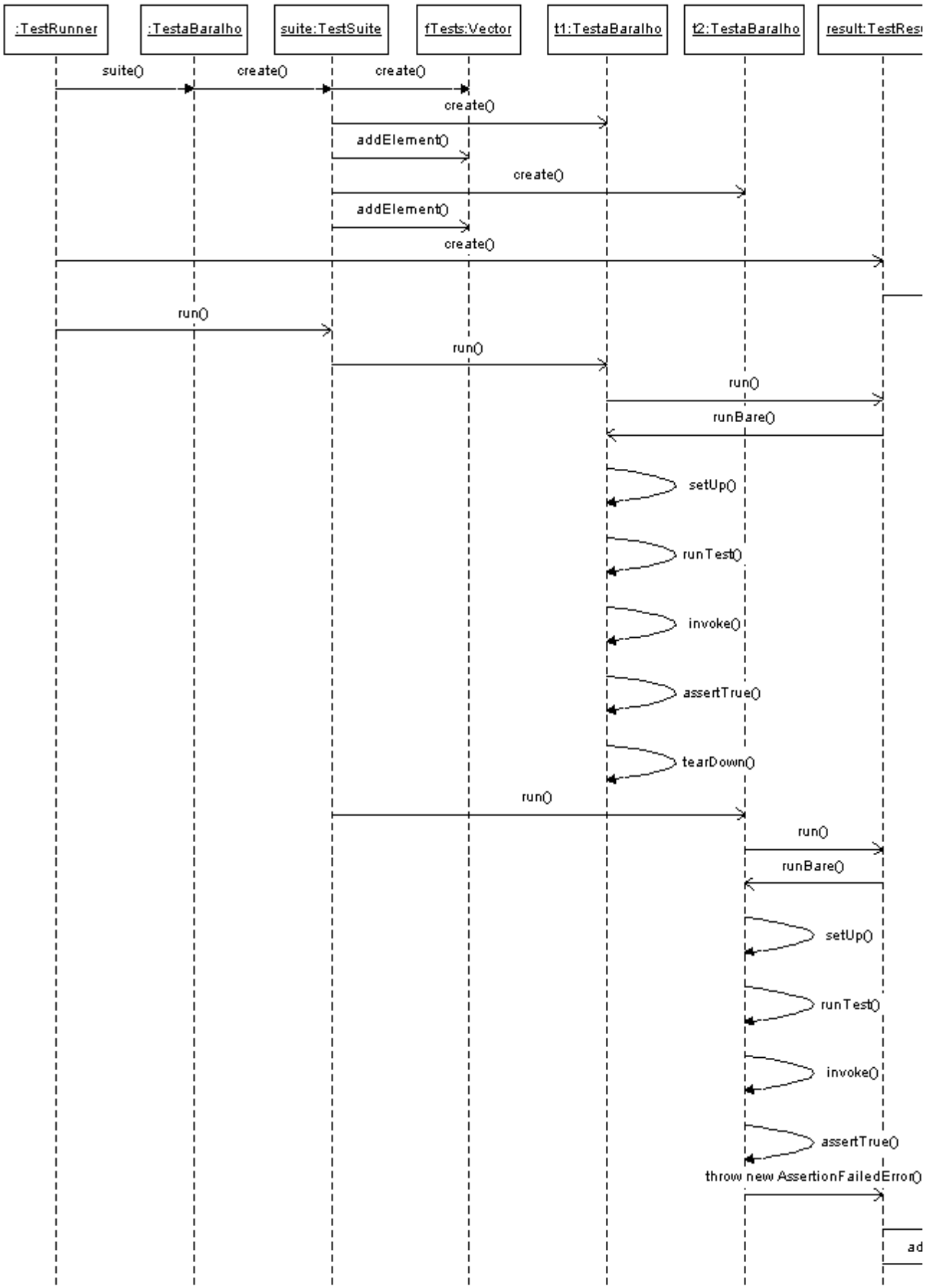


- Observações sobre cada passo (após ter executado o framework indicando que deve testar TestaBaralho)

1. O framework chama o método estático suite() da classe de teste
2. A classe de testes cria um novo TestSuite, passando a classe de testes como parâmetro
3. O construtor de TestSuite usa reflexão para descobrir os métodos de testes da classe de testes e instancia um objeto da classe de testes para cada método de teste
4. O framework chama run() para cada objeto instanciado
5. setUp() é chamado para criar um contexto para o teste
6. O método particular testXpto() é chamado
7. O método de teste pode fazer uma ou mais asserções

8. Uma asserção poderá estar falsa, em cujo caso uma exceção é lançada ...
9. ... e é capturada pelo framework, que então exibe o erro na interface do usuário (gráfica ou textual)

- Um diagrama de seqüência mais detalhado segue abaixo:


:TestRunner
:TestaBaralho
suite:TestSuite
fTests:Vector
t1:TestaBaralho
t2:TestaBaralho
result:TestRes
suite()
create()
create()
create()
addElement()
create()
addElement()
create()
run()
run()
run()
runBare()
setUp()
runTest()
invoke()
assertTrue()
tearDown()
run()
run()
runBare()
setUp()
runTest()
invoke()
assertTrue()
throw new AssertionFailedError()

Observações gerais

- Projetar com padrões é muito útil
- Explicar um projeto para outras pessoas usando padrões é também muito útil
- Frameworks maduros têm uma alta densidade de padrões
  - Frameworks maduros são mais fáceis de usar, mas podem ser mais difíceis de mudar
- JUNIT foi testado usando o próprio JUNIT, claro!
- JUNIT é um framework muito simples: ele faz o trabalho mínimo necessário
  - Extensões estão disponíveis em outro package (junit.extensions)
  - Uma das extensões é um TestDecorator que permite executar código adicional antes e depois de um teste

Por que JUNIT é um framework?

- Ele faz quase tudo que é necessário para testar
- Força o programador a dar apenas o que falta:
  - Através de hook methods (setUp, tearDown, runTest, construtor da classe de testes)
  - Fornece defaults para todos os hook methods, menos o construtor da classe de testes
  - Fornece um padrão de nomes a seguir para os métodos de testes para que sejam descobertos automaticamente pelo framework
- O Hollywood Principle está em ação:
  - A lógica básica está pronta
  - O framework chama seu código de testes
  - Porém observe que seu código também chama o framework (com os métodos assertTrue, ...)