JayUnit

Projeto de Software para a Disciplina Técnicas de Programação Avançada (2012.1)

Clovis Pinheiro
Daniel Tamaki Batista
Marcos Rodrigues

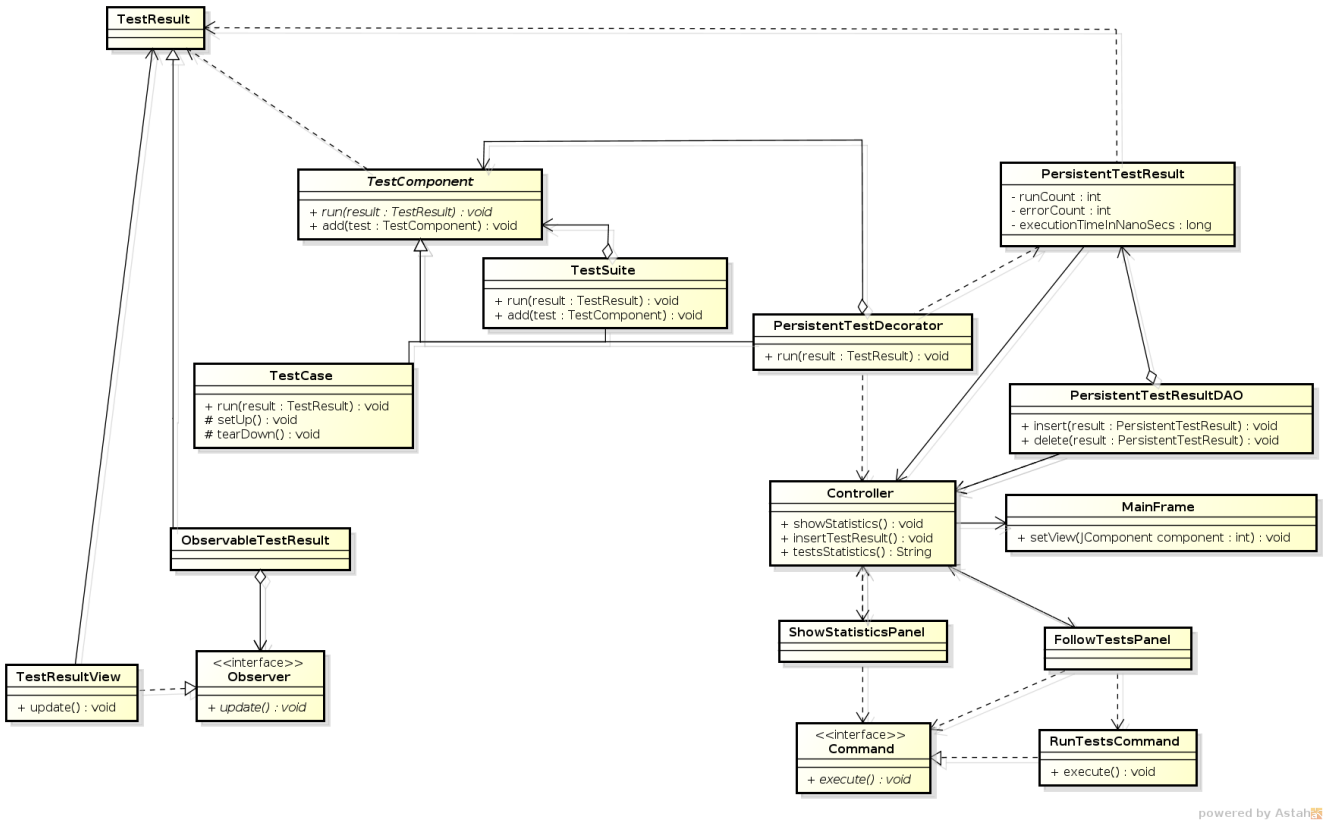
1. Descrição do problema

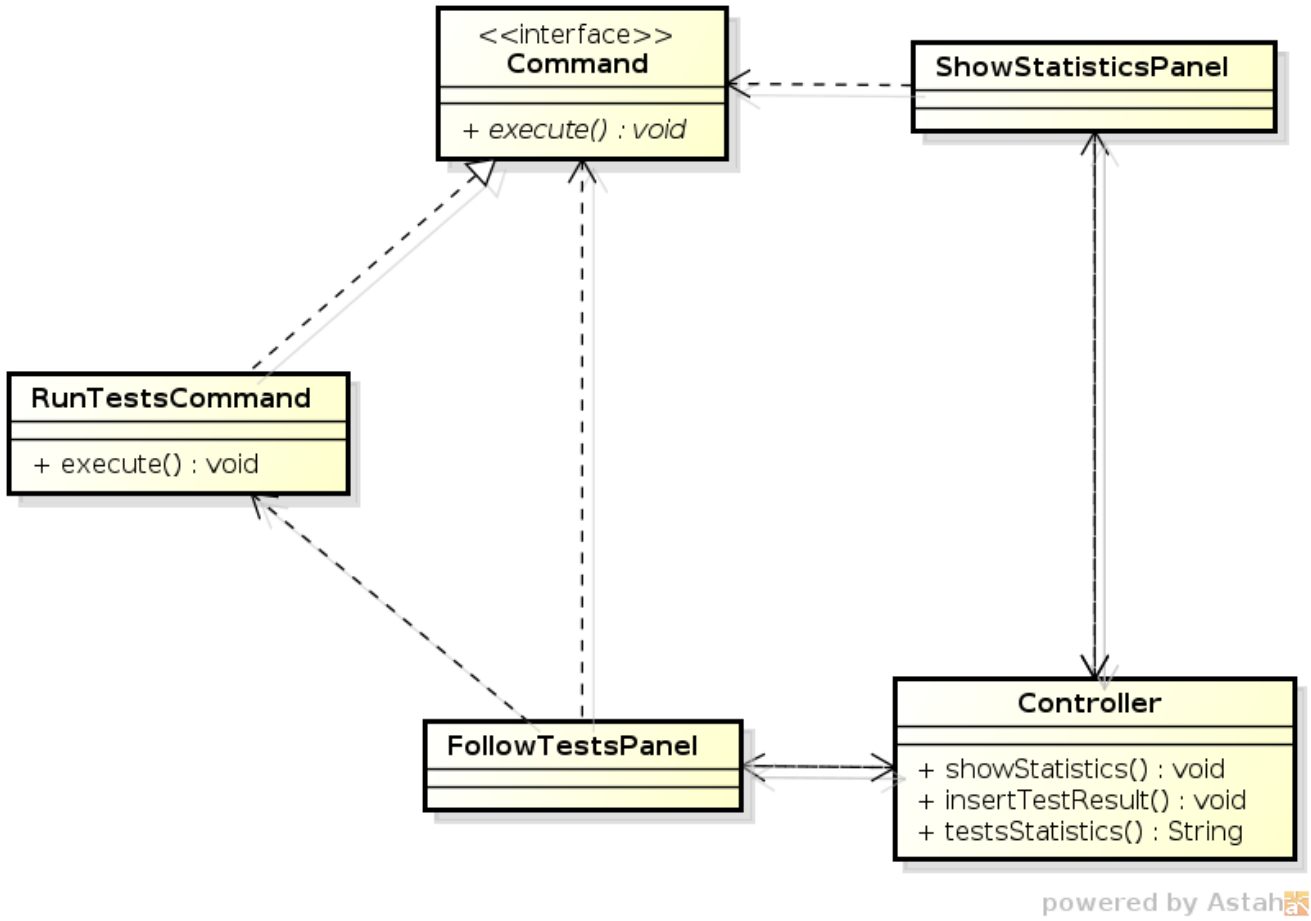
A motivação para este projeto foi criar um framework para testes, baseado no JUnit, de forma que seja fácil escrever e executar testes unitários automatizados. Estes são escritos de modo a garantir o funcionamento de pedaços de código, mesmo muito tempo depois de sua implementação, sendo assim fácil identificar quando e onde um erro é introduzido durante a evolução do software. Também é possível utilizar os testes como um guia para a programação, prática conhecida como TDD (Test-Driven Development).

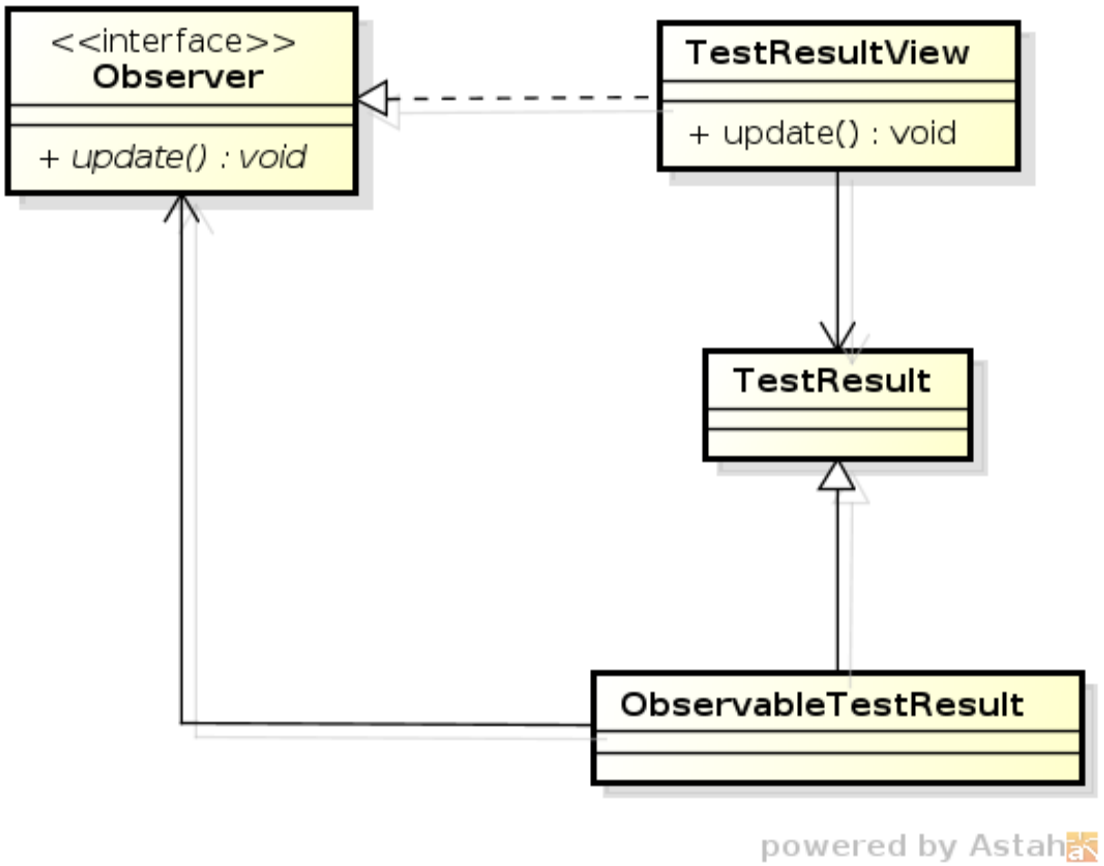
O framework deve possuir uma classe base que deve ser extendida pelo cliente. Os métodos que realizarão os testes não devem receber quaisquer parâmetros, e seus nomes devem sempre iniciar por “test”. Esta classe deve possuir um método de inicialização e um de finalização, que são chamados antes e depois de cada teste respectivamente, para que se possa minimizar a repetição na preparação para os testes. Além disso, devem existir métodos utilitários para determinar as condições de validade do teste (asserções), tais que informem claramente o estado dos dados testados em caso de falha. Também deve poder ser executada uma suíte (coleção) de testes.

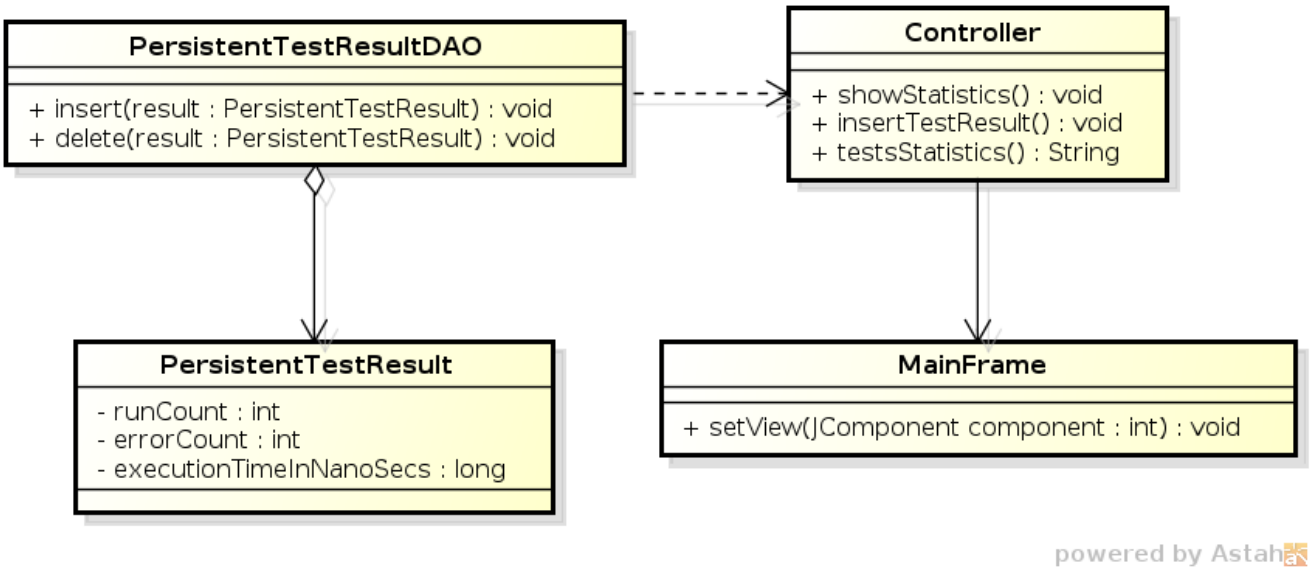
Deve existir uma interface gráfica de onde seja possível acompanhar a execução dos testes em tempo real, e que persista os dados desta execução. Deve ser possível analisar o histórico de testes quanto ao número de execuções, quantidade média de testes, de falhas, e tempo levado na execução.

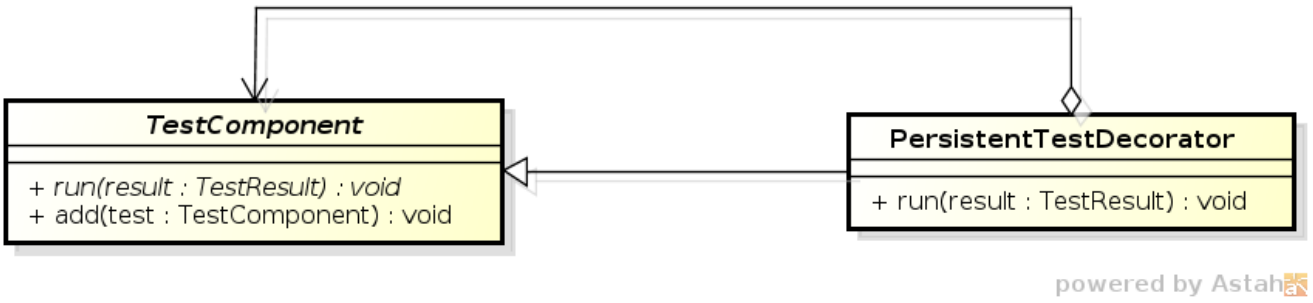
2. Projeto de Classes

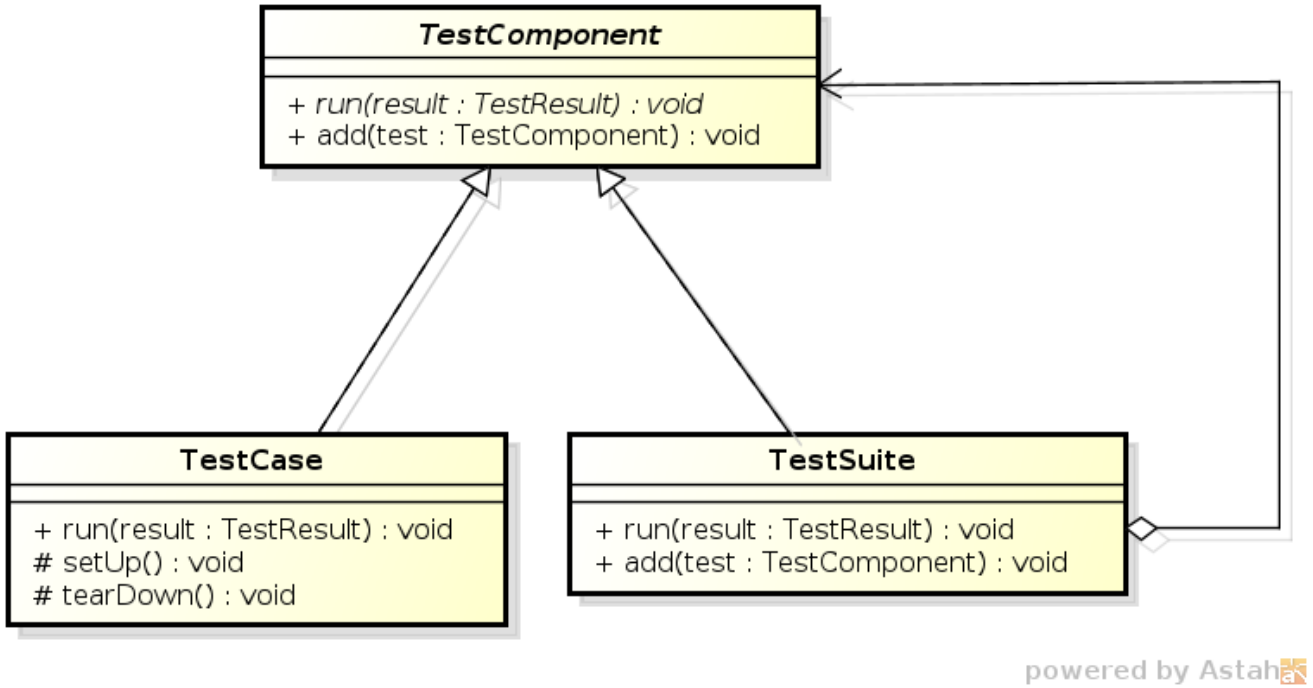


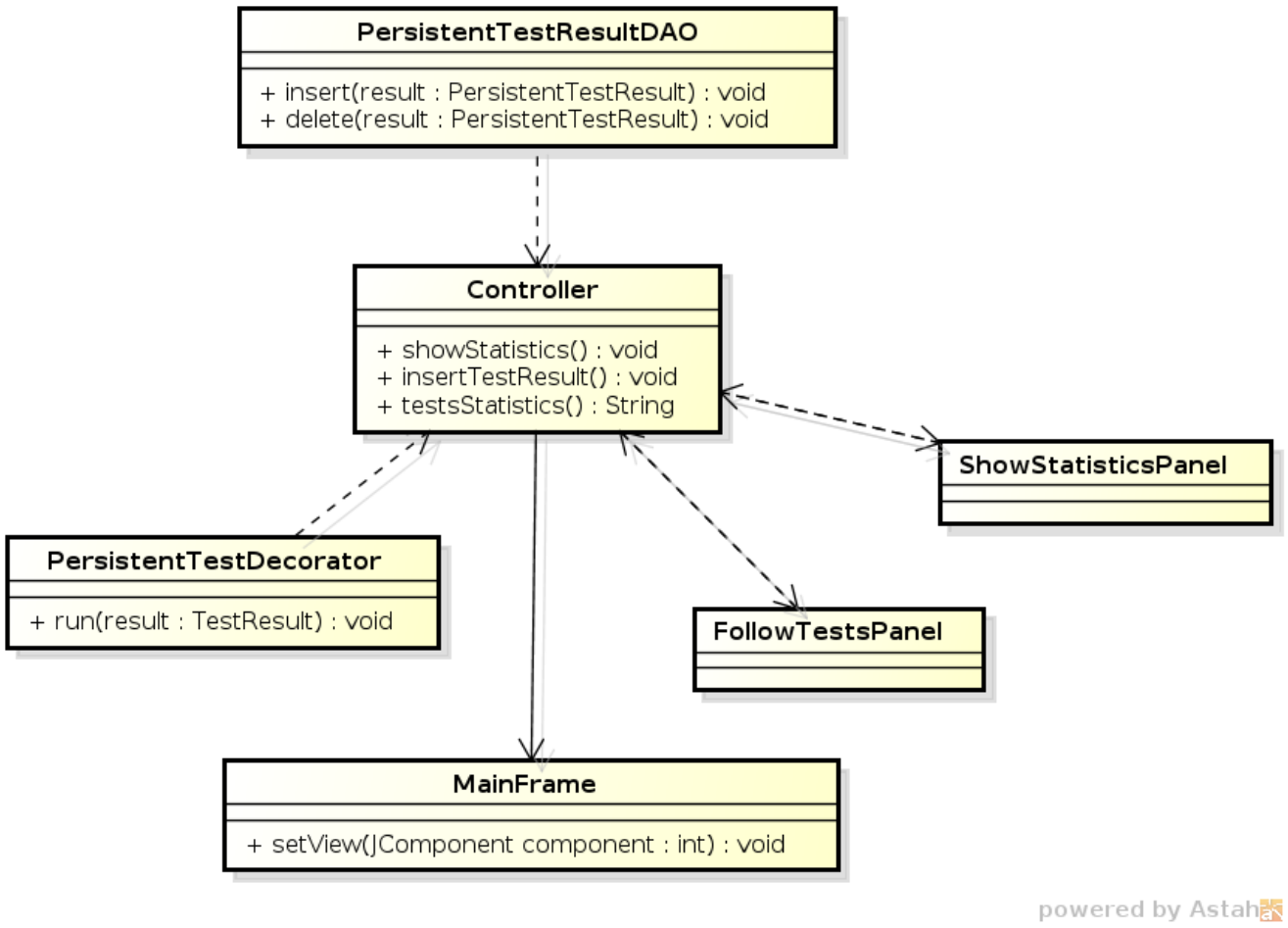

<<interface>>
Command
+ execute() : void
ShowStatisticsPanel
RunTestsCommand
+ execute() : void
FollowTestsPanel
Controller
+ showStatistics() : void
+ insertTestResult() : void
+ testsStatistics() : String
powered by Astah


<<interface>>
Observer
+ update() : void
TestResultView
+ update() : void
TestResult
ObservableTestResult
powered by Astah


PersistentTestResultDAO
+ insert(result : PersistentTestResult) : void
+ delete(result : PersistentTestResult) : void
Controller
+ showStatistics() : void
+ insertTestResult() : void
+ testsStatistics() : String
PersistentTestResult
- runCount : int
- errorCount : int
- executionTimeInNanoSecs : long
MainFrame
+ setView(JComponent component : int) : void
powered by Astah


TestComponent
+ run(result : TestResult) : void
+ add(test : TestComponent) : void
PersistentTestDecorator
+ run(result : TestResult) : void
powered by Astah


TestComponent
+ run(result : TestResult) : void
+ add(test : TestComponent) : void
TestCase
+ run(result : TestResult) : void
# setUp() : void
# tearDown() : void
TestSuite
+ run(result : TestResult) : void
+ add(test : TestComponent) : void
powered by Astah



3. Padrões de projeto utilizados

- MVC

Vimos necessidade de separar a lógica de negócios, de persistência e de apresentação. Desse modo, o software pôde atinge alta ortogonalidade, sendo possível alterar livremente suas camadas sem interferir no resto dos componentes.

- Command

Foi necessário implementar de antemão ações modulares que deveriam ser executadas em um momento indeterminado. Em nosso caso, devido à eventos gerados pelo usuário ao clicar em um botão,

- Facade

O padrão MVC faz com que as classes de apresentação nunca se comuniquem diretamente com os modelos, ou mesmo entre si. Deste modo, o controlador age como um Facade, repassando os comandos para as classes adequadas.

- Observer

Por nossa aplicação exigir o acompanhamento em tempo real dos resultados dos testes, foi necessário utilizar um Observer.

- Decorator

Como um dos requisitos era que os resultados dos testes fossem persistidos, mas não queríamos acoplar este feature ao funcionamento usual do framework, criamos um decorator que realiza automaticamente esta operação, tornando isto transparente para o cliente.

- Template Method

O problema define que devem haver callbacks de inicialização e finalização para os testes. Consideramos o Template Method como melhor solução para este problema. Os métodos setUp() e tearDown() são chamados durante a execução do método run(result: TestResult), estando inicialmente vazios.

- Composite

A flexibilidade e praticidade exigidas do framework fizeram com que criássemos uma classe TestSuite, contendo uma lista de testes a serem executados e possuindo a mesma interface externa deles. Desta forma pudemos reutilizar facilmente outros componentes, como por exemplo os TestResults.

Apêndice – Códigos Fonte

Os códigos seguem hospedados no seguinte endereço: https://github.com/mrodrigues/JayUnit