ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Suporte a Teste de Unidade de Aplicativos J2ME no Ambiente Eclipse

Trabalho de Conclusão de Curso

Engenharia da Computação

Polyana Lima Olegário
Orientador: Prof. Márcio Lopes Cornélio

Recife, 01 de julho de 2005

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Suporte a Teste de Unidade de Aplicativos J2ME no Ambiente Eclipse

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Polyana Lima Olegário
Orientador: Prof. Márcio Lopes Cornélio

Recife, 01 de julho de 2005

UNIVERSIDADE
DE PERNAMBUCO

Polyana Lima Olegário

Suporte a Teste de Unidade de Aplicativos J2ME no Ambiente Eclipse

Resumo

O processo de teste de *software* é uma das atividades do processo de desenvolvimento e tem tido grande importância ultimamente por diminuir os custos de manutenção e permitir que problemas futuros sejam evitados antecipadamente. Aplicativos J2ME são testados de forma diferente das aplicações Java, por terem limites de memória e de processamento. Este trabalho foi desenvolvido com o intuito de facilitar o desenvolvimento de teste de aplicativos J2ME, permitindo que estes pudessem ser escritos e executados no ambiente de programação Eclipse, uma vez que o mesmo facilita o trabalho de codificação para os programadores. Esta vantagem permite que os testes sejam escritos mais rapidamente além de permitir que vários *frameworks* sejam utilizados em um único ambiente de programação. Ao final da construção dos *plug-ins* EclipseJ2MEUnit e EclipseJ2MEUnit_ui foi desenvolvido um estudo de caso com o intuito de facilitar a utilização desses *plug-ins* e também para validar o projeto. Trabalhos que podem ser realizados a partir deste projeto são testes de aplicativos construídos para modelos específicos de celular e a extensão da interface gráfica do *plug-in* para a interface *Swing*.

Abstract

The test software process is an import activitie of the sotware development. It contributes for decreasing the maintenance costs and allowing future problems to be anticipated. Due to its memory and processing speed J2ME applications are tested differently from Java applications. This work was developed to facilitate test development of J2ME applications, allowing to write and execute them in the Eclipse programming environment, as Eclipse facilitates the work of codification for the programmers. This advantage allows to write tests more quickly. Moreover many frameworks can deal with a single environment of programming. After the development of plugins EclipseJ2MEUnit and EclipseJ2MEUnit_ui, case study was done to make its use easier and also to validate the project. Future works are: test applications for specific models of mobile phones, and an extended Swing interface.

Sumário

Índice de Figuras

Tabela de Símbolos e Siglas

J2ME - Java 2 Micro Edition
IDE - Integrated Development Environment
API - Interface para Programação de Aplicativos
XP - Extreme Programming
RUP - Rational Unified Process
UML - Unified Modeling Language
TDD - Test Driver Development
PDE - Plug-in Development Environment
JTD - Java Development Tools
JVM - Máquina Virtual Java
XML - Extensible Markup Language
J2EE - Java 2 Platform Enterprise Edition
J2SE - Java 2 Platform Standard Edition
PDAs - Personal Digital Assistant Profile
KVM - Kilo Virtual Machine
CDC - Connected Device Configuration
CLDC - Connected Limited Device Configuration
MIDP - Mobile Information Device Configuration
AMS - Aplication Manager System
GUI - Grafic User Interface
COTS - Components “Off-The-Shelf”
AWT - Abstract Window Toolkit
WTK - Wireless Toolkit

Agradecimentos

Durante a realização deste projeto muitas foram as dificuldades encontradas, mas com a ajuda e compreensão de algumas pessoas que permaneceram ao meu lado foi possível chegar ao término do projeto.

Gostaria de agradecer ao meu orientador Márcio Cornélio, que além de me orientar nos estudos e por qual caminho seguir, deu me força para continuar o trabalho em momentos difíceis e acabou sendo mais que meu orientador, acabou sendo um psicólogo também.

Aos meus amigos que tiveram paciência e souberam me aconselhar quando precisei e principalmente ao clube da lulu, pelos momentos marcantes durante o curso.

Agradeço em especial aos meus pais, que me deram toda a estrutura para poder ter tempo e conforto, para poder ter forças para terminar este trabalho, além dos meus irmãos que foram muito companheiros e compreensivos quanto estava sem tempo de estar ao lado deles.

A Deus e a todos que estiveram direta ou indiretamente me ajudando e aconselhando durante a realização deste projeto, muito obrigado.

Capítulo 1

Introdução

Testes de *software* devem ser utilizados no projeto de qualquer tipo de aplicação para assegurar a corretude do mesmo. Esta atividade é realizada por meio de verificação cujo objetivo é observar as diferenças entre o comportamento esperado da aplicação e o seu comportamento real [1][2]. Testes de *software* são também capazes de detectar problemas que eventualmente não tenham sido previstos na especificação do projeto.

Existem vários estágios de teste de *software*, como os testes de unidade, de integração, de sistema, e de validação. O teste de unidade divide o sistema em módulos, e testa cada módulo individualmente.

Aplicações *wireless* devem ser testadas, assim como qualquer outro tipo de *software,* para verificar sua funcionalidade e usabilidade [3]. A aplicação de testes é tão importante em aplicações *wireless* quanto em outras aplicações, porque em aplicações *wireless* se trabalha com limites de processamento e de memória menores do que na maioria dos outros *software*s. Isto se dá pelo fato de que tais aplicações possuem restrições de processamento e de memória, portanto seu código deve ser bastante enxuto.

Já existem ferramentas que permitem automatizar o teste de unidade de *software*s durante o desenvolvimento de sistemas, tais como o *framework* JUnit [4], que dá suporte ao teste de unidade, para analisar se os resultados desejados estão de acordo com os aqueles obtidos. Este *framework* incentiva a integração do teste com o desenvolvimento, permitindo que ambos sejam implementados e executados num único ambiente de programação, o Eclipse [5].

O objetivo deste trabalho é dar suporte ao teste de unidade de aplicativos J2ME (*Java 2 Micro Edition*) no Eclipse, permitindo que assim como no JUnit, os testes e a implementação dos aplicativos sejam desenvolvidos numa única IDE (I*ntegrated Development Environment)*, além de fornecer os resultados numa interface mais amigável, facilitando o trabalho do programador. Classes de teste construídas no Eclipse podem ser desenvolvidas mais rapidamente, devido às características desta IDE.

Já existe um *framework* capaz de testar aplicativos J2ME [6], o J2MEUnit [7], que estende do JUnit. Em particular, o JUnit é utilizado para testar classes Java que utilizam a API Java padrão, não sendo possível fazermos o mesmo no caso de se utilizar a tecnologia J2ME, já que a plataforma *micro edition* não está disponível na API (Interface para Programação de Aplicativos) Java padrão. Diferente do JUnit, o J2MEUnit não foi desenvolvido para ser utilizado como um *plug-in* do Eclipse, sendo utilizado em um ambiente emulado, tal como o J2ME *Wireless Toolkit*.

1.1 Estrutura

Os tópicos a serem abordados neste projeto são:

- No Capítulo 2 apresentamos a importância em se aplicar a atividade de teste de *software* durante o processo de *software*; alguns processos de desenvolvimento, como o modelo cascata, modelo espiral e o método XP (*Extreme Programming*) [8] foram explicados, assim como onde e como a atividade de teste ocorre em cada modelo. Conceitos relacionados ao processo de teste de *software*, como os estágios da atividade de teste e os tipos de testes também foram abordados neste trabalho.
- O Capítulo 3 trata dos materiais que foram necessários durante o desenvolvimento deste projeto. O entendimento do funcionamento da IDE Eclipse e também os requisitos necessários para a construção de um *plug-in* para o Eclipse, os *frameworks* JUnit e J2MEUnit, entendendo a estrutura usada por estes *frameworks* para criar casos de teste, e finalmente a plataforma J2ME, foram estes os tópicos abordados neste capítulo.
- No Capítulo 4 apresentamos os *plug-ins* desenvolvidos neste projeto; foram dois *plug-ins:* o EclipseJ2MEUnit e o EclipseJ2MEUnit_ui; neste capítulo também apresentamos os pontos de extensão criados no *plug-in* EclipseJ2MEUnit para permitir que outros *plug-ins* possam se acoplar a ele.
- Exemplos de classes J2ME foram desenvolvidas e suas respectivas classes de teste; estes exemplos foram explicados através de trechos de código que se encontram no Capítulo 5, e auxiliam o estudo do *plug-in* desenvolvido e também ajudaram a validar o *plug-in* de alguma forma.
- A conclusão deste trabalho encontra-se no Capítulo 6, e nele são definidos os problemas encontrados durante o desenvolvimento do projeto, se todos os objetivos do trabalho foram alcançados e também contém uma Seção que trata dos trabalhos que podem ser desenvolvidos a partir deste projeto.

Capítulo 2

Teste de Software e Processos de Software

A fase de teste de *software* é considerada uma das mais importantes dentro do ciclo de vida de construção de *softwares*, por ser uma das maneiras pelas quais podemos assegurar a qualidade do *software* ou de protótipos do *software*. Este processo é aplicado tanto a *softwares* de aplicações críticas, como *software*s de controle de foguetes, como em *softwares* de menor poder computacional, como dispositivos celulares. Obviamente com diferentes enfoques dependendo do tipo de desenvolvimento.

A atividade de teste de *software* é utilizada para verificar as diferenças entre o comportamento esperado do sistema e o comportamento real observado, achando possíveis erros no sistema. É também capaz de detectar problemas que, eventualmente, não tenham sido previstos na especificação. O processo de teste de *software* pode ser visto como "destrutivo" ao invés do desenvolvimento que é "construtivo", isso porque o engenheiro de testes procura percorrer todo o *software* à procura de erros.

Neste capítulo apresentamos tipos de teste de *software* e como a atividade de teste está inserida em alguns modelos de ciclos de vida de *software*.

As abordagens de teste de *software* do tipo caixa branca e caixa preta serão explicados na Seção 2.1, assim como os estágios de testes de *software*, as etapas pelo qual o sistema deve passar para completar toda a fase de testes, e os tipos de teste, que são aplicados durante a realização dos estágios de teste.

A Seção 2.2 apresenta como a fase de testes é incorporada em modelos de processo de desenvolvimento como o ciclo de vida clássico e o modelo espiral, e também em um método ágil de desenvolvimento, como o método XP (*eXtreme Programming).*

2.1 Teste de software

Nesta Seção apresentamos duas abordagens de teste de *software*, as abordagens do tipo caixa branca (estruturais) e caixa preta (funcionais) [2]. Os estágios de teste de *software* também serão explicados, estágios estes: teste de unidade, teste de integração, teste de sistema e teste de aceitação, assim como os tipos de testes também serão apresentados neste capítulo.

Os testes de caixa preta são gerados a partir de uma análise entre os dados de entrada e saída do sistema, de acordo com os requisitos levantados pelo usuário, ou seja, não precisa do

conhecimento de como o sistema funciona internamente, apenas dos dados gerados. Esta abordagem é ideal para teste de interface gráfica, de acesso ao banco de dados, dentre outros.

Nos testes de caixa branca utiliza-se a estrutura de controle, caminhos lógicos possíveis de serem executados, para derivar casos de teste. Através dele o engenheiro de teste pode derivar casos de teste que garantam que todos os caminhos independentes dentro do sistema sejam testados pelos menos uma vez; que todos os laços e suas fronteiras sejam executados; que os valores limites sejam analisados, sabendo-se que a maioria dos erros ocorrem nas fronteiras e não no centro do domínio.

Dentro do processo de teste de *software* existem diversos estágios, pelo qual o código é testado em diferentes ambientes e por pessoas de diferentes papeis. Os estágios pelo qual os testes devem passar até chegar ao produto final são: teste de unidade, teste de integração, teste de sistema e teste de aceitação. Estes testes serão mais bem explicados nas seções a seguir. Além dos estágios de testes há os tipos de testes que podem ser executados em mais de um estágio de teste, são eles:

- Teste Funcional: testa as funcionalidades básicas do programa, testa tanto com condições válidas quanto inválidas para verificar a consistência do programa;
- Teste de regressão: após haver alguma correção ou adição de código os testes são re-executados a fim de garantir que novos erros não tenham sido introduzidos no projeto;
- Teste de estresse: verifica as funcionalidades do programa em situações limite, exige recursos em quantidade, freqüência e volume diferentes do normal;
- Teste de volume: está mais voltado para transações de Banco de Dados, testa o sistema com altos volumes de dados numa transação, verifica quantidade de terminais, se são suficientes, dentre outros testes que são realizados;
- Teste de segurança: verifica se todos os mecanismos de proteção de acesso estão funcionando corretamente. É muito importante hoje em dia diante do grande uso de programas *web*;
- Teste de desempenho: exige configuração de *hardware* e *software* para testar diferentes configurações, número de usuários, tamanho do banco de dados, e pode verificar o tempo de resposta e processamento do sistema.

Existem alguns outros tipos de teste que não foram mencionados aqui, porém estes acima citados são os mais conhecidos.

2.1.1 Teste de Unidade

Dentre os vários tipos de teste que utilizam a técnica de caixa branca, o teste de unidade proporciona que o teste do *software* seja feito em módulos, podendo-se testar cada módulo individualmente, com o objetivo de testar a estrutura interna, a lógica e o fluxo de dados, e o comportamento do sistema.

Durante a fase do teste de unidades são testados:

- Cada componente individual do sistema, ou seja, as classes e métodos, para garantir que eles operam de forma correta individualmente;
- A *interface* com o módulo, para certificar-se de que as informações podem entrar e sair do módulo testado normalmente;
- A inicialização de variáveis com valores *default* e outros valores;
- As condições de limite são usadas para garantir que o módulo opera em seus limites estabelecidos;

- Os dados armazenados temporariamente, através da estrutura de dados verificam se os dados mantêm sua integridade durante os passos de execução de um algoritmo;
- Os caminhos independentes são executados para garantir que todos os caminhos sejam executados ao menos uma vez [1].

A atividade de teste de unidade é aplicada durante várias atividades do desenvolvimento do *software*, como a codificação, revisão e verificação, com o objetivo de descobrir erros em cada uma dessas atividades. Cada caso de teste tem associado a ele um conjunto de resultados esperados, que são comparados com os resultados obtidos para verificar se há erros no código.

No caso do teste de unidade, observando-se que alguns objetos podem depender de algum outro ainda não disponível, pode-se desenvolver *drivers* e *stubs* para cada unidade de teste [9]. Um *driver* pode ser definido como um "programa principal", que aceita os dados dos casos de teste, passa tais dados para o módulo a ser testado e imprime os resultados. Os *stubs* são usados para substituir os módulos que dependem do módulo a ser testado, usam a interface do módulo subordinado e pode fazer manipulações de dados, retornando um valor esperado. A Figura 1 representa graficamente o funcionamento de *drivers* e *stubs*.

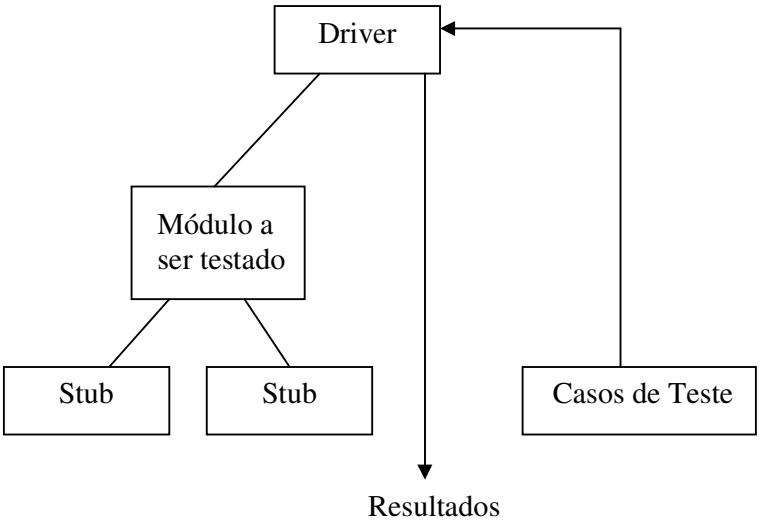

Figura 1. Funcionamento de *Drivers* e *Stubs*.

O teste de unidade geralmente é feito pelo próprio programador e é capaz de detectar erros mais facilmente quando cada módulo realiza apenas uma função, sendo o número de casos de teste diminuídos e os erros mais fáceis de serem previstos e exibidos.

Os testes de unidade são utilizados para melhorar a qualidade do *software*, já que a estrutura interna do projeto é testada, e também reduz os custos dos defeitos encontrados, pois em processos bem planejados as condições de erro são antecipadas e os caminhos alternativos para tratamento de erros são planejados.

Assim como o JUnit [4], que permite que as classes de teste sejam desenvolvidas numa IDE que facilita a construção de um projeto, como o Eclipse, e é bem aplicado para desenvolver teste de unidade durante o processo de desenvolvimento, o EclipseJ2MEUnit, *plug-in* desenvolvido neste projeto, também pode ser utilizado durante o desenvolvimento de teste de unidade por permitir rápida construção das classes de teste dentro do ambiente Eclipse.

2.1.2 Teste de integração

Após os teste de unidades serem construídos e executados é preciso que eles sejam testados em conjunto, de forma integrada. O teste de integração apresentado nesta Seção tem o objetivo de testar os testes de unidade em conjunto assim como a interface entre as unidades integradas.

Esse tipo de teste é feito de forma incremental, as unidades são integradas aos poucos para manter o controle dos erros e facilitar a localização deles. O teste de integração geralmente é executado por um testador de integração ou algum outro programador, diferentemente do teste de unidade que é feito pelo próprio desenvolvedor, pois neste caso, uma outra pessoa que não o desenvolvedor, tem mais chances de encontrar algum erro no programa.

A integração dos módulos pode ser feita de forma *top-down* ou *bottom-up*:

- *Top-down*: Os módulos são integrados a partir dos módulos mais acima na arquitetura de camadas, representando os módulos mais gerais, para os módulos mais abaixo na arquitetura, os módulos mais específicos. Assim como o teste de unidade, o teste de integração também utiliza *drivers* e *stubs*; *driver* representa módulos de controle principal e os *stubs* representam os módulos diretamente subordinados ao módulo de controle principal. Os testes são realizados à medida que cada módulo é integrado, da mesma forma que, à medida que os testes vão sendo executados, os *stubs* são substituídos pelos módulos reais do projeto.
- *Bottom-up*: a integração dos módulos inicia pela parte mais baixa da arquitetura de camadas, os módulos com as funções reais do projeto são logo integrados, dessa forma os *stubs* nem sempre são necessários neste tipo de teste de integração, pois os módulos mais específicos são testados primeiro. A cada integração de módulos um *driver* é criado para coordenar a entrada e saída dos casos de teste, logo após o teste, estes *drivers* são implementados passando a interagir com uma camada mais acima da arquitetura do projeto.

A vantagem em se utilizar a abordagem *top-down* é que as funções principais são implementadas primeiro, porém isso exige que os *stubs* contenham funções complexas para satisfazer as necessidades das classes principais. Ao contrário da abordagem *top-down*, que pode necessitar de *stubs* complexos, a abordagem *bottom-up* quase não requer a construção de *stubs*, porém os modelos principais só podem ser testados ao final da implementação de todos os módulos.

A escolha sobre qual forma de teste de integração utilizar depende dos objetivos do projeto e do cronograma definido. Pode-se usar as duas formas de maneira combinada, utilizando a abordagem *top-down* para os níveis superiores da arquitetura, e a abordagem *bottom-up* para os níveis subordinados a estes.

2.1.3 Teste de Sistema

Os testes de sistema são responsáveis por verificar se a aplicação está funcionando como um todo. Este teste geralmente é do tipo caixa-preta e é executado por um testador de sistemas, um membro do grupo que não esteja relacionado ao processo de teste.

Após os módulos terem sido integrados e testados através do teste de integração, estes módulos são agrupados formando subsistemas, o teste de sistema procura por possíveis erros nas interfaces entre subsistemas, além de validar que o sistema implementou de maneira correta os requisitos funcionais e não-funcionais definidos.

Teste de sistema na verdade se ocupa de várias atividades que procuram por à prova o sistema, por este motivo, um outro passo importante é testar a integração dos componentes de

software com o ambiente no qual será utilizado o *software*. Este ambiente refere-se ao *hardware*, *software* e o usuário final.

2.1.4 Teste de Aceitação

É o estágio final do processo de teste do *software*, este estágio de teste é realizado pelo próprio usuário com a finalidade de demonstrar que os requisitos definidos foram implementados corretamente.

Neste teste, o *software* é testado com os dados reais fornecidos pelo cliente, ao contrário dos dados gerados para simulação, esta troca da base de dados pode revelar erros ou requisitos não implementados, isso porque o *software* é testado de modo diferente ao se trocar a base de dados.

Os Testes de aceitação podem ser de dois tipos:

- Testes *alfa*: Refere-se a testes feitos pelo próprio usuário no ambiente de execução do desenvolvedor. Enquanto o cliente testa, o desenvolvedor observa e registra erros ou problemas que possam acontecer;
- Testes *beta*: testes feitos pelos usuários em suas próprias instalações e sem a supervisão do desenvolvedor. Caso o cliente encontre algum problema, este deve ser relatado ao desenvolvedor.

2.2 Processos de *software*

Vários problemas foram identificados durante a construção de novos *softwares*, dentre eles o custo. Geralmente o *software* custava 50% a mais do que o custo planejado, este aumento se dava principalmente devido ao custo de manutenção, já que não havia muita preocupação em desenvolver *softwares* pensando que os requisitos podem ser modificados, projetos grandes eram desenvolvidos, mas o custo de manutenção era muito alto. Outra dificuldade é o tempo de desenvolvimento, a maioria dos projetos não eram finalizados na data prevista, acarretando no aumento do custo também [2].

Um dos principais problemas encontrados é a qualidade do *software*, pois 75% dos sistemas não funcionavam como deviam e 57% eram entregues sabendo-se que existiam defeitos [10]. Este problema ocorre principalmente porque a atividade de teste de *software* não é inserida durante o desenvolvimento ou não é trabalhada como deveria. Problemas não corrigidos ou não previstos provocam aumento dos custos de manutenção.

Para amenizar estes problemas é necessário identificar as restrições e os riscos do projeto, estabelecer marcos para entrega de produtos (documentos, planilhas, relatórios), definir modelos que sirvam de base para construção de *software*. Os modelos de *softwares* foram idealizados com o intuito de diminuir os problemas encontrados durante o desenvolvimento de *software*.

O processo de *software* é um conjunto estruturado de atividades exigidas para desenvolver um sistema de *software*. Cada processo diferente abrange três elementos fundamentais: métodos, que descrevem o processo de "como" construir o *software*; ferramentas, que proporcionam meios de automatizar algumas tarefas durante o desenvolvimento; e os procedimentos, que constituem o elo entre os métodos e as ferramentas, definindo a seqüência em que os métodos serão aplicados, os marcos de referência e também os documentos, relatórios, formulários que devem ser entregues. Esses elementos juntos fornecem ao desenvolvedor uma base para a construção do *software* e permitem que o *software* atinja alta qualidade.

A utilização de processos de *software* permite o controle do processo de desenvolvimento do *software*, através da definição de datas de entrega dos produtos, das restrições e dos requisitos

do projeto. Outra importância da utilização do processo de *software* é servir de base para construção de *software* de alta qualidade.

Softwares novos podem seguir modelos já existentes ou o *software* requer expansão e modificações no sistema para se adequar às exigências do cliente.

Existem atividades fundamentais no processo de *software*, são elas:

- Especificação de *software*: define as funcionalidades e restrições do *software*;
- Projeto e implementação de *software*: construção do *software* baseado na especificação;
- Validação de *software*: o *software* é validado para certificar que atende aos requisitos do cliente;
- Evolução de *software*: como os requisitos estão em constante mudança, o *software* precisa suportar essas mudanças.

Um modelo de processo de *software* é uma representação abstrata de um processo. O modelo apresenta uma descrição de um processo a partir de uma perspectiva especifica.

Dentre os vários modelos de processo de *software*, destacam-se:

- *Ad-hoc*: Não segue regras de desenvolvimento, em geral, a implementação é iniciada sem uma definição clara dos requisitos. Não é adequado para projeto de médio e grande porte;
- Cascata: Segue atividades específicas de forma seqüencial e será visto na Seção 2.2.1;
- Construção de Protótipos: Baseia na constante construção de protótipos para os clientes, antecipando possíveis mudanças dos requisitos;
- Espiral: Tem como base a construção de protótipos e o processo incremental, a Seção 2.2.2 apresenta este modelo;
- XP (*Extreme Programming*): processo contrário aos modelos burocráticos, será apresentado na Seção 2.2.3.

Durante a construção de um *software* é necessário definir qual processo será utilizado no desenvolvimento, a escolha do processo depende das características do projeto e das necessidades do cliente. Por exemplo, se o objetivo é o desenvolvimento rápido do *software*, a metodologia XP é a mais indicada, mas se a preocupação for na interface do usuário o modelo de construção de protótipos pode ser o mais indicado, já que fornece vários protótipos do projeto ao cliente.

2.2.1 Ciclo de Vida Clássico

A engenharia de *software* permite a utilização de diferentes atividades durante a construção de um *software*. Estas atividades podem ser usadas em diferentes processos de *software*, o processo define os passos pelo qual o processo de desenvolvimento de *software* deve seguir. O ciclo de vida clássico foi um dos primeiros processos a serem criados e ainda é bastante utilizado pelos desenvolvedores, por isso a importância de ser tratado neste trabalho.

O ciclo de vida clássico, também conhecido como modelo cascata, foi sugerido por Royce, evoluindo de outros processos de engenharia [1]. A Figura 2 ilustra este ciclo de vida.

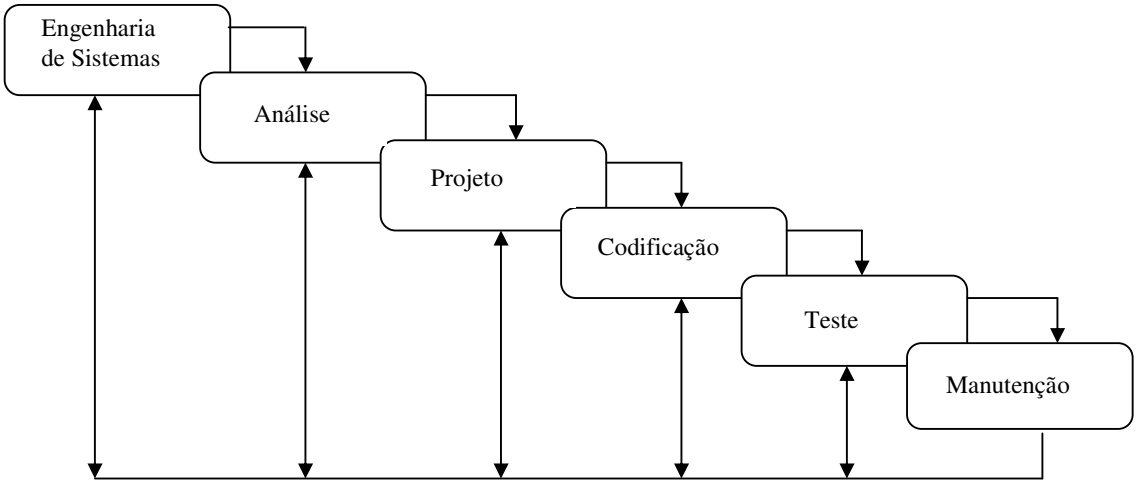

Figura 2. O Ciclo de Vida Clássico

Como ilustra a Figura 2, no ciclo de vida clássico as atividades são executadas de forma seqüencial. A atividade de Engenharia de Sistemas é a fase onde são estabelecidos os requisitos do *software*; na fase de Análise é definido o escopo do projeto, as funções que o *software* irá desempenhar. A fase de Projeto consiste em caracterizar a interface e definir a arquitetura e a estrutura dos dados do projeto. Durante a Codificação o projeto é implementado propriamente. Na fase de Teste, são realizados testes para descobrir possíveis erros. Durante a Manutenção todas as etapas anteriores são aplicadas a um *software* já existente.

Este modelo é fácil de usar, porém pressupõe que os requisitos do sistema ficarão estáveis, fato que raramente ocorre, por isso quando há alguma modificação nos requisitos é necessário voltar para a fase inicial e começar todo o processo novamente, podendo haver atrasos na entrega do projeto.

O fato de que o cliente deve definir todos os requisitos do *software* no começo do projeto para que este modelo não tenha muitos problemas é uma desvantagem, pois é difícil para os clientes definirem todos os requisitos sem ter muita experiência com desenvolvimento de *software*s e sem ter usado algum protótipo anteriormente.

Observa-se que a atividade de teste é executada num período definido neste modelo, ou seja, somente após a fase de codificação é que são executados testes no *software*. Porém, têm surgido novos conceitos sugerindo que a atividade de teste seja aplicada durante toda a fase de construção do sistema, para corrigir erros o quanto antes e evitar futuros problemas [9].

Embora este processo de desenvolvimento de *software* tenha seus pontos fracos, ainda é melhor do que utilizar o modelo *Ad-hoc*, no qual não há processo definido para se desenvolver *software*s.

2.2.2 Modelo Espiral

O modelo espiral procura aprimorar o modelo de ciclo de vida clássico, visto na Seção 2.2.1, só que ao contrário do ciclo de vida clássico, no modelo espiral as atividades não são executadas apenas de forma seqüencial, mas também de forma iterativa, onde novas funções são adicionadas a cada ciclo. As atividades de análise, especificação, projeto, implementação e validação são repetidas a cada ciclo do modelo, gerando uma nova versão do *software*, um protótipo, permitindo maior retorno ao cliente, conseqüentemente possibilita o cliente avaliar melhor os requisitos do sistema a cada iteração.

O modelo foi proposto por Boehm em 1988. A principal característica deste modelo é que foi dada grande importância à análise de riscos, a cada *loop* no espiral é realizada a análise dos riscos e a definição dos planos para minimizar estes riscos.

Cada *loop* no espiral é dividido em quatro etapas, que seguem abaixo:

- Planejamento: nesta etapa é definido o escopo do projeto, as restrições que podem aparecer no processo, o plano de gerenciamento é elaborado e os riscos do projeto são identificados, juntamente com um plano alternativo para cada risco;
- Avaliação e Redução de Riscos: após os riscos terem sido identificados é preciso avaliar o quanto o risco tem impacto no *software.* Uma análise detalhada de cada risco é elaborada, assim como as atividades para reduzir os riscos são realizadas;
- Desenvolvimento e Validação: o modelo de desenvolvimento é escolhido nesta fase, dependendo das restrições e objetivos do projeto;
- Avaliação feita pelo cliente: após ter passado pela fase de implementação o cliente avalia se é necessário passar para o próximo *loop* da espiral e continuar o desenvolvimento do *software*, ou se no ponto em que está o projeto já está suficiente.

A Figura 3 ilustra essas atividades no modelo espiral. O processo inicia na etapa de planejamento e cada vez que passa pela etapa de avaliação do cliente é verificado se haverá continuidade do projeto.

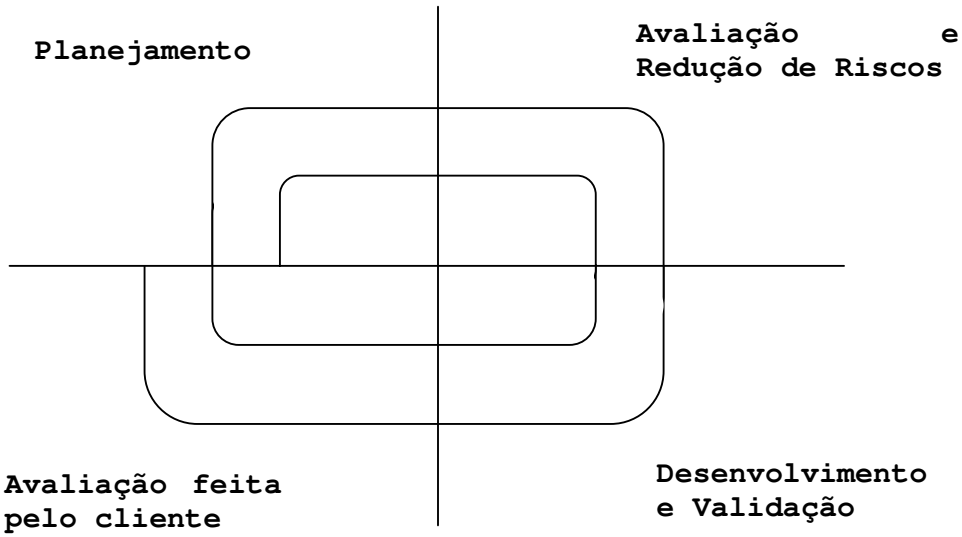

Figura 3. Modelo Espiral

Como pode-se observar, a avaliação de riscos é realmente importante neste modelo, tanto é, que boa parte das atividades estão relacionados com os riscos do projeto. Os riscos representam algo que possa vir a dar errado, como por exemplo, o risco de um sistema bancário na *Web*, que utiliza um servidor de baixo processamento, é alto, pois pode acarretar em muita perda ao banco, de clientes e de dinheiro. Os riscos resultam em problemas ao desenvolvimento, por isso a importância em minimizar os riscos identificados.

Diferentemente do ciclo de vida clássico, no modelo espiral não há necessidade que o cliente tenha todo o conjunto de requisitos no início do desenvolvimento. Este modelo permite que os requisitos sejam acrescentados sem dificuldades conforme sejam encontrados, facilitando

o cumprimento dos prazos, pois os requisitos mudam constantemente. Além disso, a entrega de vários protótipos ao cliente faz com que este estabeleça melhor os requisitos de seu *software*, pois tem algo em que se basear.

A atividade de teste de *software* neste processo acontece mais freqüentemente do que no ciclo de vida clássico, pois a cada *loop* são realizados testes referentes ao que já foi codificado. Apesar realizarem mais testes, este modelo ainda não chega a dar a importância que o método XP, que será explicado na próxima Seção, dá ao processo de teste de *software*, que chega até a fazer a classe de teste antes da classe de código.

A principal preocupação ao se utilizar este modelo é que ele exige que se tenha experiência na avaliação de riscos, pois se um risco de grande impacto passar desapercebido pelos analistas pode acarretar grandes problemas, podendo ser até necessário abortar o projeto.

2.2.3 Método XP

O método usado em *eXtreme Programming* (XP) [8], proposto em 1996 por Kent Beck e Ward Cunningham, enfatiza o desenvolvimento ágil e com uma equipe pequena ou média. O objetivo é desenvolver o projeto rapidamente, cumprindo as estimativas e satisfazendo o cliente. Ao contrário do modelo cascata, esta metodologia prevê a mudança constante dos requisitos e procura amenizar atrasos e elevação de custos, por conta disto.

Algumas características desta metodologia:

- Planejamento: decide o que pode ser feito e o que pode ser adiado no projeto, focando não em requisitos futuros, mas nos requisitos existentes. Os clientes definem o escopo do projeto enquanto os desenvolvedores definem as estimativas de prazo e o processo de desenvolvimento, desta forma as duas partes cooperam com o projeto;
- *Feedback* constante: os testes do código são feitos constantemente, indicando possíveis erros no sistema, desta forma tem-se um *feedback* do *software*. Em relação ao cliente também é preciso mantê-lo informado sobre o desenvolvimento do projeto, este trabalho pode ser feito através da entrega de protótipos que permitam ao cliente conhecer parte do sistema e sugerir mudanças para a próxima versão do protótipo;
- Abordagem incremental: durante o desenvolvimento do sistema segue-se um plano de projeto que procura abranger todas as possíveis melhorias no sistema, isto durante toda a vida do projeto;
- Programação em pares: a implementação do código é feita em dupla, enquanto um programador codifica o outro observa possíveis melhorias ou erros na sintaxe e semântica do código, depois os programadores trocam de pápeis. Esta forma de programar permite que os programadores fiquem sempre aprendendo uns com os outros.
- Testes: a validação do projeto ocorre durante todo o processo de desenvolvimento;
- Encoraja a comunicação entre as pessoas: trocas de informação entre o gerente e os desenvolvedores, e também entre o cliente e os desenvolvedores;
- Simplicidade: desenvolvimento do projeto dentro do escopo definido, utilizando código simples e sem se preocupar com funcionalidades que poderiam vir a ser importantes no futuro, isto porque nem sempre estas funcionalidades seriam usadas, e mesmo porque sempre há mudanças nos requisitos, fazendo com que o projeto seja reavaliado de qualquer forma;
- 40 horas semanais: defende que não se deve fazer hora extra constantemente, se for observado que as pessoas estão fazendo muito hora extra, deve-se avaliar o planejamento do projeto, e não sobrecarregar as pessoas [11].

O processo de teste no método XP é iterativo e deve começar no início do processo de desenvolvimento. O desenvolvedor de *software* tem que ter em mente a idéia de teste de *software* durante a descrição do projeto do *software*. Uma característica importante é esta metodologia procura se seguir a técnica TDD (*Test Driver Development*)[12].

A TDD define que o código da classe de teste deve ser escrito antes da classe da aplicação, ou seja, a classe de teste e os dados usados para testar podem ser vistos como uma especificação da classe a ser usada na aplicação. Este método pode ser aplicado em qualquer processo de desenvolvimento e enfatiza que o teste de *software* seja feito constantemente.

Como para se fazer a classe de teste pode ser preciso instanciar a classe da aplicação ou chamar algum método da classe da aplicação, o esqueleto da classe da aplicação é desenvolvido a médida que o teste é implementado e percebe-se a necessidade de utilizar algum atributo ainda não definido na classe da aplicação. Ou seja, a tecnologia XP é seguida mas não completamente, devido a esta necessidade de se ter o esquelo da classe da aplicação para executar a classe de teste.

O foco do método XP está no desenvolvimento rápido, satisfação do cliente e das pessoas envolvidas no projeto. Não segue atividades seqüenciais como o ciclo de vida clássico e nem chega a ser desprovido de processos como o modelo *Ad-hoc*, mantém uma certa organização no projeto, mas não de forma inflexível, tanto que a atividade de documentação não é tão trabalhada nesta metodologia, ter o *software* pronto é mais importante.

Seguindo o escopo de teste de *software* deste trabalho, percebemos a principal diferença entre o ciclo de vida clássico, o modelo espiral e o método XP: no ciclo de vida clássico os testes são feitos todos durante uma única fase do projeto e quando se encontra algum erro necessita-se voltar à fase anterior ou inicial do projeto; no modelo espiral a cada *loop* no modelo os testes são executados; enquanto que no método XP o processo de teste é feito de forma incremental, não havendo tanta perda de tempo e de custos como no ciclo de vida clássico.

Ferramentas de teste de *software*, *frameworks* ou metodologias podem ser aplicados a qualquer um desses processos de desenvolvimento de *software*. O JUnit é um *framework* voltado para testar *software* e pode também ser usado em qualquer processo, mas é bastante utilizado na metodologia de desenvolvimento XP, porque este *framework* permite a criação rápida das classes de teste e das classes de código do projeto. Além disso, as classes de testes são executadas antes, apenas necessitando do esqueleto da classe de código para retornar algum resultado.

As classes de código são criadas após as classes de testes. Este processo evita que os testes se tornem tendenciosos, já que o próprio desenvolvedor é quem faz os testes unitários do projeto. Quando a classe de código é criada primeiro a tendência é que o programador teste apenas os caminhos que imagina serem os percorridos. Ás vezes esquecendo de testar as fronteiras do sistema, os valores limites ou os laços utilizados.

Capítulo 3

Material

Este Capítulo tem como objetivo apresentar as ferramentas e tecnologias utilizadas neste trabalho. A primeira Seção trata do ambiente de desenvolvimento Eclipse, sua estrutura e funcionalidades. Suas subseções explicam o funcionamento de um *plug-in* no Eclipse e o conhecimento necessário para a construção de novos *plug-ins*; a próxima Seção é sobre o *framework* JUnit, utilizado para realizar teste de unidade, e que pode ser acoplado ao Eclipse sem dificuldades, pois foi desenvolvido para também ser um *plug-in* do Eclipse. A Seção que trata do J2MEUnit define as diferenças deste *framework* em relação ao JUnit e como deve-se utilizá-lo.

A Seção 3.3 explica a plataforma J2ME, quais suas diferenças em relação as outras plataformas da *Sun*, sua arquitetura, além de exemplos de como utilizar os métodos e classes necessárias para executar tais aplicativos.

3.1 Eclipse

Esta Seção explica a arquitetura do ambiente Eclipse e as vantagens em se utilizar esta plataforma. Servirá como base para a próxima Seção que aborda *plug-ins*, e necessita de um prévio conhecimento nesta plataforma de desenvolvimento.

A plataforma Eclipse disponibiliza um conjunto de funcionalidades, sendo a maioria dessas muito genérica, capaz de abranger conceitos básicos mas que permite a construção de novas ferramentas para tratar de casos específicos.

O Eclipse é uma IDE que permite a integração de várias ferramentas que podem ser combinadas formando um ambiente de desenvolvimento personalizado. Estas ferramentas são chamadas de *plug-ins*. Com a utilização do Eclipse pode-se ter uma mesma plataforma que permite sua utilização durante várias fases do desenvolvimento, como programação, testes, integração, *web design*, dentre outros.

O projeto Eclipse começou a ser idealizado em abril de 1999 pela IBM e a OTI, e em 2001 foi lançada a versão 1.0 do Eclipse e aberto o sítio http://www.eclipse.org. Em fevereiro de 2004 o *Eclipse Foundation* se torna uma organização sem fins lucrativos, com o intuito de promover a evolução, promoção e suporte da plataforma, permanecendo concentrado na idéia de *software* livre [5].

Após a instalação desta plataforma percebe-se que a estrutura de pacotes fica dividida da seguinte forma:

- *Configuration*: diretório onde todas as configurações para execução do eclipse se encontram;
- *Features*: ao instalar um novo *plug-in* um arquivo é criado neste diretório contendo informações sobre o *plug-in* instalado;
- *Plugins*: contém todos os plugins presentes na plataforma. Podem não estar sendo executado todo o tempo, mas ficam disponíveis para o usuário habilitá-los;
- *Readme*: disponibiliza um arquivo HTML com informações sobre o eclipse, problemas encontrados, dentre outras informações;
- *Workspace*: todos os projetos criados no Eclipse ficam nesta pasta, podem ser criados dentro do mesmo *workspace* ou não, o conceito de *workspace* será explicado mais adiante.

Esta estrutura permite manter arquivos referentes ao Eclipse organizados.

A arquitetura do Eclipse pode ser dividida em três camadas, `Plug-in Development Environment` (PDE), `Java Development Tools` (JTD) e `Platform`. A Figura 4 mostra essas três camadas.

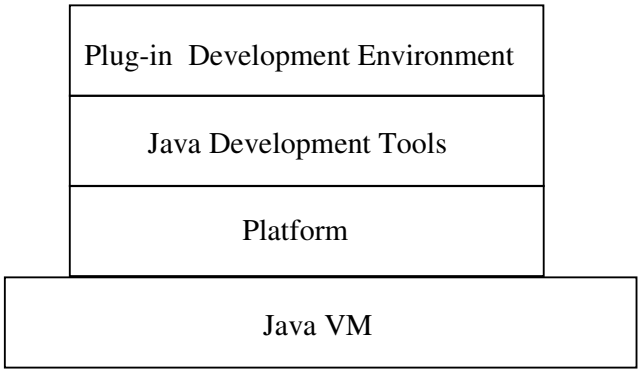

Figura 4. As três camadas do Eclipse [13].

As três camadas do Eclipse mostradas na Figura 4 são executadas na máquina virtual Java (JVM). A camada `Platform` define a estrutura da linguagem de programação neutra. A camada `Java Development Tools` adiciona o ambiente de programação Java ao Eclipse e a camada PDE estende a JDT permitindo o desenvolvimento de *plug-ins* para o Eclipse.

A camada de Plataforma consiste de vários componentes. Seguem alguns deles nos tópicos abaixo:

- `Workspace`: diretório onde ficam armazenados diversos projetos criados no Eclipse, o `workspace` contém também configuração própria, pode-se ter mais de um `workspace` com diferentes configurações, para permitir trabalhar com diferentes tipos de arquivos;
- `Workbench`: consiste na parte visual do Eclipse e define os editores, `views` e perspectivas;
- Editores: podem abrir, fechar e salvar os recursos, entende-se por recursos arquivos e pastas. Vários arquivos podem estar abertos ao mesmo tempo;
- `Views`: exibem informações sobre objetos, podendo ser uma descrição, uma estrutura de diretórios, ou mesmo a estrutura de uma classe;
- Perspectivas: conjunto de *views* e editores formam uma perspectiva, por exemplo, existe uma perspectiva para se utilizar a funcionalidade Debug, outra para permitir o desenvolvimento de aplicativos. Cada perspectiva personaliza o Eclipse de forma a melhorar o ambiente para o desenvolvedor.

- `Runtime`: define a infra-estrutura de *plug-ins*, percebendo a disponibilidade de um *plug-in* na inicialização do Eclipse e gerenciando o carregamento do *plug-in* no ambiente.

A Figura 5 ilustra como esses componentes e camadas do Eclipse são dispostos no projeto do Eclipse.

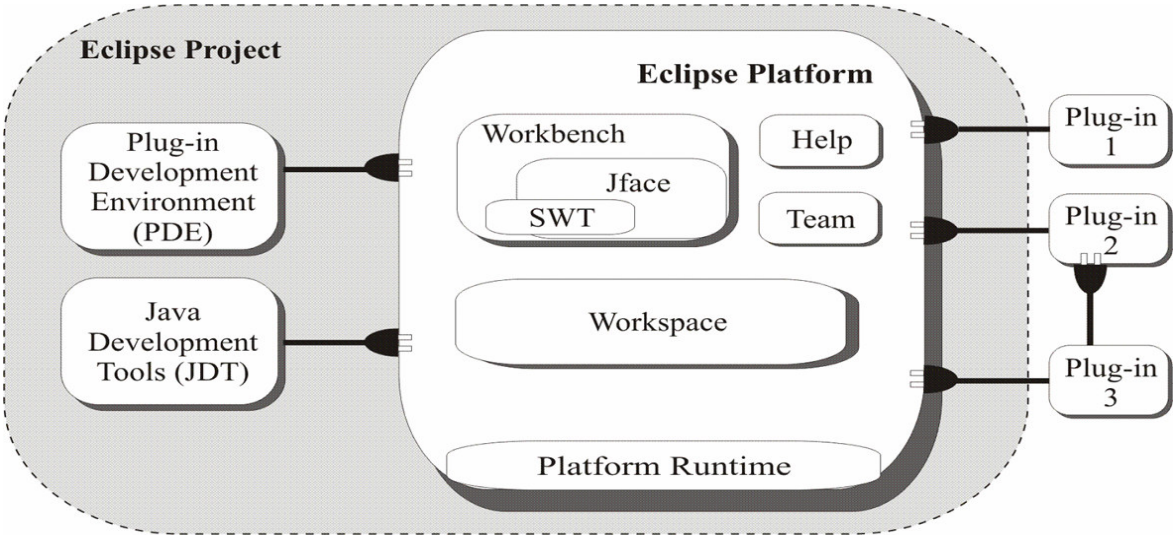

Figura 5. Projeto Eclipse [14].

Através da Figura 5 fica mais fácil identificar como o PDE e o JDT são ferramentas que dão suporte ao Eclipse mas que ficam à parte da plataforma Eclipse, isso porque qualquer alteração nestas ferramentas poderia ser feita sem necessitar alterar o Eclipse também. Percebe-se que dentro da plataforma Eclipse encontramos todos os componentes já citados anteriormente e alguns que não foram mencionados como o *JFace* e o SWT, que são componentes de interface gráfica em Java. A Figura 5 ilustra também como os *plug-ins* são acoplados à plataforma Eclipse, e que pode haver dependências entre *plug-ins* como o *plug-in* 3 depende do *plug-in* 2.

Depois de conhecido o funcionamento do ambiente Eclipse fica mais fácil explicar como *plug-ins* são acoplados no Eclipse e como se dá a dependência entre *plug-ins*, a próxima Seção explica estas tarefas.

3.1.1 *Plug-ins* no Eclipse

Esta Seção descreve o funcionamento de plugins no Eclipse, utilizando um exemplo, e também permite conhecimento para construção de um novo *plug-in* e acoplamento deste no ambiente Eclipse.

A plataforma Eclipse foi construída com base num mecanismo de descoberta, integração e execução de módulos chamados *plug-ins*. Quando a plataforma é carregada, é apresentado o ambiente integrado de desenvolvimento (IDE) composto de um conjunto de *plug-ins*. Nesta Seção, explicamos como um *plug-in* é reconhecido pela plataforma e como estes são executados.

A primeira regra do Eclipse é "Tudo é uma contribuição". Como conseqüência desta regra, existem diversas contribuições [13]. O ambiente de programação Java e a base do Eclipse, juntos tem mais que 60 grandes *plug-ins*. A IBM contribuiu com mais 500 *plug-ins*.

O mecanismo básico pelo qual pode-se estender a utilização do Eclipse é através de novos *plug-ins* que podem adicionar novas tarefas a *plug-ins* já existentes, estendendo funções que já existiam ou modificando-as.

Um *plug-in* no Eclipse é um componente que fornece um determinado serviço utilizando o ambiente de programação Eclipse. O Eclipse provê uma infra-estrutura que suporta a operação de vários *plug-ins* trabalhando em conjunto no mesmo ambiente para facilitar o desenvolvimento de determinadas tarefas.
Cada contribuição ou *plug-in* deve ser sempre adicional, nunca se deve substituir ou realocar outra contribuição, é dever do desenvolvedor pensar nisso antes de fazer a contribuição. O Eclipse se preocupa apenas em manter o ambiente em harmonia, combinando as contribuições.

Para que um *plug-in* possa ser incluído no Eclipse, o *plug-in* desejado é adicionado a instancia da classe *Plugin*. A classe *Plugin* é responsável por configurar e gerenciar as tarefas que podem ser suportadas por um *plug-in* que deve ser construído estendendo a classe `org.eclipse.core.runtime.Plugin`, que é uma classe abstrata usada para facilitar o gerenciamento dos *plug-ins*.

Após a instalação do Eclipse cada *plug-in* encontra-se em diretórios diferentes. Cada *plug-in* é descrito em um arquivo *manifest* do tipo XML (*Extensible Markup Language*) chamado `plugin.xml`, localizado no mesmo diretório do *plug-in*. Este arquivo XML é responsável por transmitir ao Eclipse as configurações necessárias para a ativação do *plug-in* que significa carregar suas classes e instanciar e inicializar estas classes, tornando o *plug-in* pronto para ser usado dentro do Eclipse.

Um *plug-in* é estruturado da seguinte forma:

- plugin.xml – o arquivo *manifest* contendo a descrição da contribuição do *plug-in*;
- /icons/*.gif ou *png - recursos utilizados, como imagens;
- about.html – licença do *plug-in*;
- *.jar – código Java;

Logo abaixo, na Figura 6, segue um exemplo de um arquivo XML usado por um *plug-in*.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="JUnit Testing Framework"
  id="org.junit"
  version="3.7"
  provider-name="Eclipse.org">
  <runtime>
    <library name="junit.jar">
      <export name="*"/>
    </library>
  </runtime>
</plugin>
```

Figura 6. Exemplo de um arquivo *manifest*.

Neste arquivo XML encontra-se as seguintes *tags*:
`name`: descreve o nome do *plug-in*, deve ser de fácil entendimento pelo usuário;

- `id`: identificador do *plug-in*, neste exemplo, `org.junit`. O `id` deve ser único;
- `version`: versão do *plug-in* criado, é importante observá-la porque pode-se usar versões diferentes de um mesmo *plug-in*.
- `library`: declara a biblioteca onde os arquivo do *plug-in* estão disponíveis, `junit.jar`;

A ativação ou desativação de um *plug-in* está relacionada com a alocação e liberação de recursos. Quando um *plug-in* necessita ser ativado ou desativado a classe que estende

`org.eclipse.core.runtime.Plugin` sobrescreve os métodos *startup* e *shutdown*, para realizar a ativação ou desativação do *plug-in*, respectivamente.

O Eclipse contém uma plataforma *runtime*, a qual é a única parte do Eclipse que não é um *plug-in*. Esta plataforma é responsável por gerenciar os *plug-ins*, quando percebe a existência de um novo *plug-in*, durante a inicialização do Eclipse, mantendo as informações do *plug-in* num registro. Os *plug-ins* que são codificados diretamente na plataforma Eclipse são ativados pela plataforma. Os que não estão codificados na plataforma necessitam de outros *plug-ins* para serem ativados. Os *plug-ins* não são carregados durante a inicialização do Eclipse, permitindo que o tempo de abertura do Eclipse seja independente da quantidade de *plug-ins* instalados.

Existem duas formas de um *plug-in* necessitar de outro: através da dependência, onde o *plug-in* pré-requisitado contém as funções do *plug-in* dependente; ou através de extensão, onde o relacionamento é entre um *plug-in* principal e um *plug-in* que o estende, este último acrescenta funções ao *plug-in* principal [14]. Estes dois tipos de relacionamentos precisam ser declarados no arquivo XML, caso sejam utilizados. Em seguida será melhor detalhado estes dois relacionamentos.

A dependência de um *plug-in* em relação a outro também é tratada no arquivo *manifest*. Esta configuração é feita através da *tag* `requires`, a qual define de qual *plug-in* será a dependência.

A relação de dependência entre *plug-ins* é encontrada tanto em tempo de execução como em tempo de compilação. Durante a execução, o Eclipse tem que ter certeza que o *plug-in* pré-requisitado estará disponível quando o *plug-in* dependente for ativado. Durante a compilação, o Eclipse modifica o *classpath* para compilar o *plug-in* dependente pelos arquivos *jars* dos *plug-ins* pré-requisitados.

O processo de adicionar funções a um *plug-in* já existente é conhecido como extensão, o *plug-in* sendo estendido é conhecido como *plug-in* principal neste tipo de relacionamento. Em casos de extensão simples, um objeto *callback* é adicionado para permitir a comunicação entre o *plug-in* principal e o *plug-in* que o estende. Uma extensão pode criar vários objetos *callback* utilizados no ambiente de desenvolvimento.

Os *plug-ins* que podem ser estendidos possuem diferentes tipos de *slots* nos quais os *plug-ins* estendidos podem ser acoplados. Esses *slots* são chamados de pontos de extensão (*extension point*). A plataforma `runtime` é responsável por ligar as extensões criadas nos pontos de extensão corretos.

A Figura 7 ilustra o relacionamento de extensão entre a área de trabalho (*Workspace*) do Eclipse e o *plug-in* de ajuda do Eclipse. Neste relacionamento o *plug-in* principal é a interface da área de trabalho do Eclipse, `org.eclipse.ui`, cujos menus podem ser estendidos através do ponto de extensão conhecido como *actionSets*. O *plug-in* que estende o *plug-in* principal é a interface de ajuda do Eclipse, *org.eclipse.help.ui*. O *plug-in* "extensor" estende o *plug-in* principal usando itens de menu específicos, neste caso, temos: *Help -> Help Contents* e *Help -> Search* [15]. Esta simples extensão aumenta a área de trabalho do Eclipse com mais itens de menu.

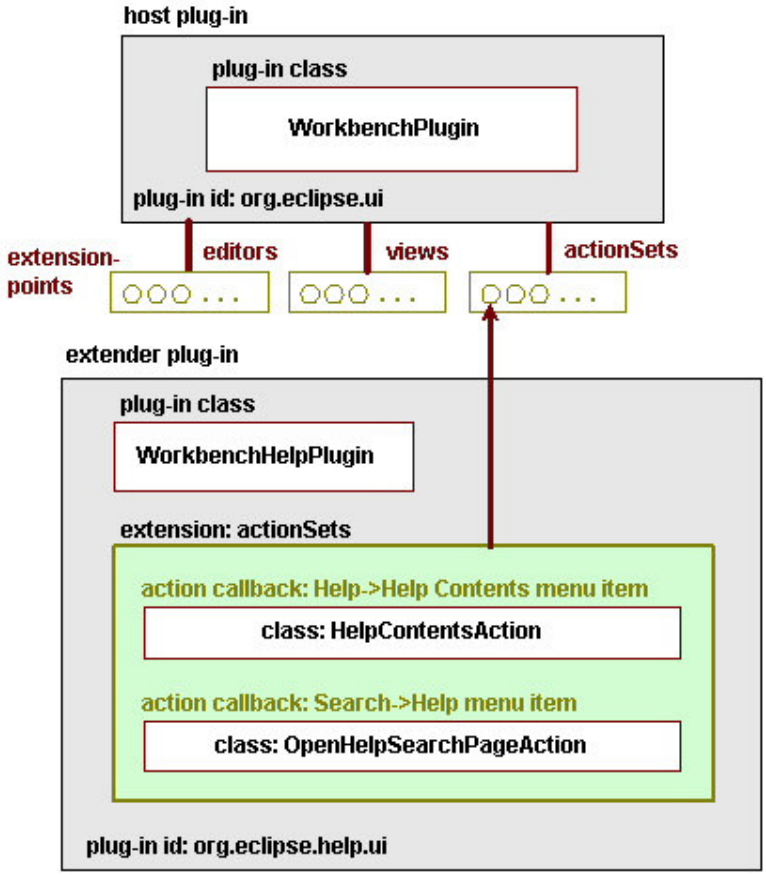

Figura 7. Elementos de uma extensão de *plug-in*. O `plug-in Workbench UI` é estendido pelo `plug-in workbench help` através do ponto de extensão `actionSets` [15].

As classes de extensão dos objetos *callback* também são mostradas na figura acima. A classe *OpenHelpSearchPageAction* é a classe que irá gerar o objeto *callbak* para *Search -> Help* e a classe *HelpContentsAction* para o item de menu *Help -> Help Contens*.

Assim como o teste de aplicações Java tem grande importância durante o desenvolvimento, o teste de plugins também tem grande importância, principalmente porque um *plug-in* é desenvolvimento não para uma empresa ou trabalho em particular, mas para poder ser utilizado por outras pessoas interessadas.

Um passo bastante importante durante a construção de um *plug-in* é testá-lo constantemente. Diferentemente das aplicações Java de propósito geral cujos testes são realizados através do JUnit, *plug-ins* são testados através do PDE JUnit. O JUnit permite rodar testes para aplicações Java, porém o mesmo falha ao tratar de testes de *plug-ins*, pois ao rodar o teste para *plug-in* não se está trabalhando dentro da área de trabalho do Eclipse.

Um PDE (*Plug-in Development Enviroment*) é um ambiente de desenvolvimento que permite a criação, manipulação e implantação de novos *plug-ins* no Eclipse. É através dele que os *plug-ins* são executados durante sua criação.

Para realizar testes no Eclipse de novos *plug-ins* existe uma versão estendida do JUnit, o PDE JUnit, que cuida da inicialização do Eclipse e roda os testes dentro de uma área de trabalho do Eclipse. A partir do Eclipse 3.0 o PDE JUnit vem junto com o Eclipse.

PDE JUnit é usado para escrever testes unitários para novos *plug-ins*. Da forma que em aplicações Java, os testes são organizados em subclasses da classe `TestCase`, existindo um teste

para cada método, e sobrescrevendo os métodos `setUp` e `tearDown`, métodos para criar e desfazer um ambiente requerido pelo teste.

A diferença do PDE JUnit está na forma em que ele é executado. Quando se executa aplicações Java, a aplicação é executada na mesma área de trabalho (`workspace`) do projeto, só que em outra máquina virtual. Quando se executa um projeto através do *Runtime Workbench*, disponível na opção *Run As* do menu Eclipse, é inicializada outra área de trabalho do Eclipse contendo o projeto da primeira área de trabalho. O PDE JUnit age de forma semelhante ao *Runtime Workbench*, Figura 8, inicializando outro `workspace`. O PDE JUnit é executado selecionando o item do menu Eclipse: *Run -> Run As -> JUnit Plug-in Test*.

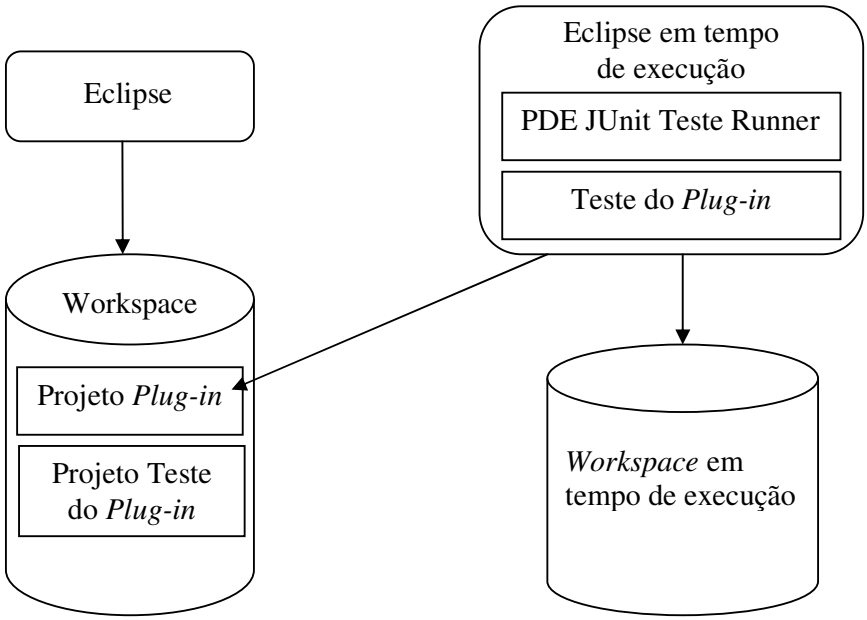

Figura 8. PDE JUnit roda os testes em tempo de execução do Eclipse [13].

Como pode-se ver na Figura 8, o PDE JUnit inicializa outro `workspace` do Eclipse limpa, sem classes ou arquivos de configuração. Porém, será necessário que haja dados dentro do `workspace`, para rodar os casos de teste.

É possível criar o `workspace` com um ponto fixo de inicialização colocando-se dentro do método de inicialização `setUp` uma chamada a uma classe responsável por fazer toda a configuração do ambiente, e no método `tearDown` deletar o projeto do *workspace*, permitindo que o próximo caso de teste seja executado sem problemas.

Para se construir um novo *plug-in* é preciso utilizar o PDE e conhecer antecipadamente o comportamento desejado do *plug-in* para evitar incompatibilidade no ambiente e substituição de funções de outros *plug-ins*.

3.2 Framework JUnit

O *framework* JUnit [4] foi desenvolvido para permitir que aplicações Java pudessem ser testadas, este *framework* pode ser utilizado como um *plug-in* do Eclipse e a partir dele outros *plug-ins* já foram desenvolvidos para testar outras aplicações, como aplicações J2ME e J2EE, por exemplo.

Após a leitura deste Capítulo será possível entender como o JUnit trabalha para executar os casos de teste e como proceder para utilizar o JUnit num projeto, como as classes devem ser construídas e quais métodos devem ser implementados. Há exemplo de um caso de teste,o qual permite melhor entendimento do assunto, assim como um exemplo de uma suíte de teste também.

Um *framework* é visto como um conjunto de classes e interfaces que trabalham juntos para resolver algum tipo de problema de *software*. Trabalham fortemente com o conceito de reuso de código, pois as principais vantagens de se trabalhar com *frameworks* consiste na diminuição do tempo para desenvolver uma aplicação. Assim como as API's, os *frameworks* permitem aos desenvolvedores, partir de um patamar superior de implementação da arquitetura, abstraindo funções específicas, facilitando o desenvolvimento de aplicações.

O *framework* de teste JUnit foi desenvolvido por Kent Beck e Erich Gamma e possui o código aberto, possibilitando o estudo do código fonte do *framework*, bem como extensão de sua funcionalidade [17].

O JUnit é fácil de ser utilizado, e já possui tamanha popularidade que em alguns ambientes de programação este *plug-in* já vem instalado, como é o caso do Eclipse 3.0. O JUnit tem um código não tão simples, por ser projetado como um *framework*, permitindo flexibilidade para que o desenvolvedor sobreponha algumas partes enquanto preserva outras de seu interesse.

O método de desenvolvimento XP, já descrito no Capítulo 2.2.3, defende a idéia de testar e depois codificar, o *framework* JUnit é bem utilizado neste processo. Escreve-se a classe de teste e depois a classe de código, desta forma é mais fácil manter a estabilidade do programa e a produtividade do desenvolvedor, diminuindo o tempo necessário para re-trabalho (*debugging*).

O objetivo do JUnit é facilitar a criação de casos de teste, já que para cada classe a ser criada deve existir um caso de teste; permitir que testes possam ser reutilizados, evitando que os programadores tenham que escrever testes duplicados.

Para se testar códigos de um projeto, um trecho de código não pode ser testado isoladamente, ele precisa fazer parte do ambiente do tempo de execução do projeto. Outra necessidade é automatizar a execução dos testes de unidade, executando periodicamente todo o teste no sistema para ter certeza que nada está errado. Para isso, o teste de unidade não deve ser checado manualmente, uma falha no teste deve ser documentada para se fazer à análise do mesmo. A execução dos testes, gravando os resultados e documentando os erros encontrados, também pode ser feita através do JUnit.

A arquitetura do JUnit será mostrada a seguir, explicando e exemplificando alguns padrões de projeto que são aplicados no JUnit [16].

O primeiro passo, quando se deseja realizar testes, é desenvolver o objeto para representar os conceitos básicos do teste, o `TestCase`. Geralmente o desenvolvedor, antes de criar a primeira classe, já tem em mente os casos de teste que deverão ser realizados. Estes objetos auxiliam a manipulação de testes facilmente.

Toda classe de teste que o programador irá criar deve ser uma subclasse da classe `TestCase`, que é declarada no pacote do `junit.framework` localizado no arquivo *junit.jar*. O nome usado na classe de teste pode ser qualquer um, já os nomes dos métodos devem começar com *test*, por exemplo: `testNomeDoMétodo`.

O JUnit tem a vantagem de poder usar a técnica de reflexão disponível na API JDK. Esta técnica permite que métodos declarados com a string inicial *test* sejam reconhecidos pelo objeto `TestRunner` e executados automaticamente dentro da classe `TestSuite`, que reconhece todos os testes a serem executados dentro de uma determinada classe e os executam automaticamente. Estes métodos só são reconhecidos se declarados como públicos, de outra forma, haverá erro durante a execução da `TestSuite`.

A classe `TestCase` implementa a interface `Test`, que contém métodos como *do*, *run* e *perform*. Os testes podem ser executados individualmente ou podem ser executados em conjunto, todos ao mesmo tempo, através da utilização do objeto composto `TestSuite`.

A classe `TestSuite`, assim como `TestCase`, implementa a interface `Test`, porém seu método `run` é diferente do `run` do objeto `TestCase`. O método `run` de `TestSuite`, chama o método `run` de cada um dos testes que a *suite* de testes contém.

A classe `TestRunner` é uma outra classe que faz a interface com o usuário exibindo os resultados do teste através de uma versão gráfica (*junit.Swing*) e uma versão com textos (*junit.textui*).

Nas figuras seguintes serão apresentados trechos de código que facilitarão o entendimento deste *framework*. A Figura 9 ilustra a construção da classe `TestCase`.

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

Figura 9. Assinatura da classe `TestCase`.

O método `run` da classe `Test` é responsável pela execução do teste. Sua implementação é mostrada na Figura 10.

```
public void run () {
    setup();
    runTest();
    teardown();
}
```

Figura 10. Assinatura do método `run`.

O método `setUp` é invocado antes de cada método de teste, é responsável por configurar o ambiente, mantendo um contexto padrão para um teste, para manter os testes independentes. Um bom exemplo deste método seria introduzir diversos tipos de sons, como MP3 ou MIDI, para estes poderem ser testados. Ao contrário do método `setUp`, o método `teardown` é chamado depois de cada método do teste especificado, com a função de desfazer o contexto criado pelo `setUp`. Seguindo o mesmo exemplo acima, remover os arquivos de sons para manter a configuração inicial. O método `runTest`, é utilizado para controlar a execução de um teste particular.

Um objeto `TestCase` tem a capacidade de realizar a execução do teste, porém, o desenvolvedor pode desejar que seja gerado um documento informando as falhas e os casos de sucesso do teste durante sua execução. Para esta finalidade é criado um novo objeto, chamado `TestResult`, que coleta os resultados da execução do teste, parte de sua estrutura está mostrada na Figura 11. A criação deste objeto baseou-se no padrão *Collecting Parameter* [18].

```
public class TestResult extends Object {
 public TestResult() {
 … }
 public synchronized void addError(Test test, Throwable t) {
 … }
 public synchronized void addFailure(Test test,
      AssertionFailedError t) {
     … }
 public synchronized Enumeration errors() {
 … }
 public synchronized Enumeration failures() {
 … }
}
```

Figura 11. Exemplo da classe `TestResult`.

Ao utilizarmos o JUnit os valores esperados são testados através dos métodos `assertTrue`, `assertFalse` ou `assertEquals`, existem ainda outras condições além de *True*, *False* e *Equals*, porém estes são os mais conhecidos. O JUnit registra todo o caminho percorrido durante as falhas obtidas nos métodos `assertXxx` gerando um relatório após a execução de todos os testes.

A seguir são mostrados alguns dos métodos de assertivas encontrados na classe `TestCase` do JUnit, assim como sua descrição e assinatura:

- `assertFalse`: Verifica se a expressão é falsa, caso não seja, a mensagem de erro é adicionada a lista de mensagens a ser mostrada no termino da execução dos testes.
- `assertEquals`: Verifica se o valor do primeiro objeto é igual ao valor do segundo objeto, caso não seja, a mensagem de erro é adicionada a lista. Estes objetos podem ser do tipo *String*, *float*, *double*, *long*, *bollean*, *byte*, *char*, *shor*, *int* e *Object* desde que os objetos sejam do mesmo tipo. No caso do tipo *Object,* verifica se apontam para a mesma referencia.
- `assertNull`: Verifica se o objeto é nulo. Caso não seja, adiciona a mensagem de erro à lista.

Existem outros tipos de assertivas que podem ser encontradas em [4], e todos esses métodos `assertXxx`, podem ser usados sem a utilização mensagem de erro, esperada com um dos parâmetros.

Na Figura 12 podemos observar um exemplo bastante simples de código utilizando o *framework* JUnit. Este classe de teste, `TestSoma`, irá testar método `somar` da classe `Soma`.

```
import junit.framework.*;
public class TestSoma extends TestCase{
      private Soma som;
      public TestSoma(String name){
            super(name);
      }
      public void setUp(){
            som = new Soma();
      }
      public void testSomar(){
            int valorEsperado = 5;
            assertEquals(valorEsperado,som.somar(2,3));
      }
}
```

Figura 12. Exemplo de Caso de Teste do JUnit.

Como foi dito anteriormente, para se utilizar o JUnit para escrever classes de teste primeiramente estende-se a classe `junit.framework.TestCase`. As subclasses de `TestCase` irão fazer uma chamada aos testes na ordem desejada. Esta classe contém dois métodos responsáveis por iniciar e finalizar a tarefa de testar todos os testes que o desenvolvedor deseja realizar, são os métodos `setUp` e `tearDown`, que são responsáveis por realizar estas tarefas, respectivamente. No exemplo acima o método `setUp` inicializa a classe `Soma`, a ser testada.

O construtor recebe como parâmetro uma String `name` que é passada para a super classe, `TestCase`. Este construtor funciona desta forma pois quando se está rodando vários testes dentro de um `TestSuite` precisamos saber o nome do `TestCase` que falhou.

O método `testSomar` testará o método `somar` da classe `Soma` através da assertiva `assertEquals`. O método `somar` recebe dois inteiros e faz a operação de soma, retornando um inteiro como resultado. Neste caso a assertiva precisa receber tipos iguais para fazer a comparação, a variável `valorEsperado` contém o valor que espera como resultado da operação de soma, e através dela pode-se observar se o método exerce o comportamento correto.

Depois de feita a classe de teste pode-se executá-la a partir de três interfaces nas quais os testes podem ser rodados para o JUnit:

- *TextUI*: Mostra um texto como saída da execução.
- *AwtUI*: Exibe uma GUI (*Grafic User Interface*) como saída usando o AWT (*Abstract Window Toolkit*) de Java.
- *Swing*: Exibe uma GUI na saída usando a interface de usuário Swing de Java.

Após a execução do teste, é exibido uma tela ou texto com o resultado do teste, relatando se o teste obteve êxito ou não. Em caso de erro ou falha, aparece a linha do erro ou da falha também. Um erro está relacionado a problemas de execução como *ArrayIndexOutOfBoundsException*, e a falha está relacionada ao resultado da assertiva.

Caso o valor esperado seja igual ao valor resultante do método `somar`, será exibido na interface do JUnit uma barra com a cor de preenchimento verde, indicando que o teste passou sem falhas, caso haja falhas a cor exibida é o vermelho.

Como para cada classe de código é feita uma classe de teste, geralmente há vários testes para serem executados. O JUnit fornece um objeto `TestSuite`, o qual permite que todos os testes sejam executados de uma só vez.

Na Figura 13 há um exemplo de classe `TestSuite`, executando o *TestCase* `TestSoma` e `TestSub`, o caso de teste `TestSub` não foi implementado, mas fica implícito sua função de fazer a operação de subtração entre dois operandos.

```
import junit.framework.Test;
public class AllTests {
      public static Test suite() {
            TestSuite suite = new TestSuite("Test Operadores");
            suite.addTestSuite(TestSoma.class);
            suite.addTestSuite(TestSub.class);
            return suite;
      }
}
```

Figura 13. Exemplo de um `TestSuite`.

Os casos de teste a serem executados são adicionados à classe `TestSuite`, através do método `addTestSuite`. Se a classe `TestCase` são contiver o método `suite`, o `TestRunner` irá procurar por todos os métodos da classe que começam com *test* e colocá-los dentro de uma *suite*, para poder executá-los.

O entendimento aprofundado do *framework* JUnit é de grande importância para realização deste trabalho, visto que o JUnit tem forte apoio de desenvolvedores e empresas, devido a sua robustez, e serve como base para construção de outros *frameworks* de teste de aplicativos específicos, como celulares, aplicativos Web, dentre outros.

3.2.1 *Framework* J2MEUnit

Esta Seção é grande importância para este trabalho por se tratar do *framework* J2MEUnit, base para o desenvolvimento deste trabalho e fonte de pesquisa. A seguir serão detalhadas as diferenças entre este *framework* e o *framework* JUnit e também alguns exemplos de como utilizar o J2MEUnit.

Este *framework* baseado na plataforma J2ME, contém um *framework* de teste de aplicativos J2ME, permite que tais aplicativos sejam testados segundo suas limitações de hardware, como tempo de processamento, memória restrita, dentre outros [19].

Foi desenvolvida baseando-se no *framework* de teste JUnit, porém, o JUnit pode utilizar a técnica de reflexão pois a API da plataforma J2SE permite que esta técnica seja utilizada, mas o J2MEUnit utiliza a plataforma J2ME, onde não há a API de reflexão disponível.

No JUnit, a técnica de reflexão transforma os métodos de uma classe em instâncias individuais do tipo `Test`, facilitando a execução dos casos de teste do aplicativo. Esta técnica de reflexão evita que na suite de teste seja criada uma subclasse da classe `TestCase` para cada caso de teste, e que todas elas sejam instanciadas. No JUnit, o desenvolvedor cria uma suite de testes, estendendo a classe `TestCase`, nesta suite cada método representa um caso de teste individual e devem começar com a string *test*, para poderem ser reconhecidas pelo objeto `TestRunner`.

O J2MEUnit tenta aliviar o fato de não usufruir a técnica de reflexão, utilizando alguns mecanismos novos ou implementando algumas classes ou métodos de forma diferente do JUnit [7]. Assim como no JUnit, no J2MEUnit para criarmos uma classe de teste estendemos a classe `TesCase`, e usamos os métodos `setUp` e `tearDown` para iniciar e finalizar os métodos de teste. Porém, no J2MEUnit, como não há API de reflexão, os métodos de teste não são reconhecidos automaticamente, desta forma, o desenvolvedor deve construir a suite de testes que deve ser executada. O método *suite* permite a construção desta *suite* de testes e retorna uma instância da classe `TestSuite`, que contém todos os métodos de teste que foram incluídos na *suite* de teste. Na Figura 14 há um esqueleto para melhor ilustrar este método.

```
public Test suite() {
      return new TestSuite(Class classe, String[] nomeTestes);
}
```

Figura 14. Exemplo de utilização do método *suite* da classe `j2meunit.framework.TestCase`.

A assinatura do método acima é usada dentro de cada classe de teste com o intuito de ter um único método, `suite`, capaz de chamar todos os métodos que realizam testes do aplicativo naquela determinada classe. O argumento `classe`, representa a própria classe de teste e o argumento `nomeTestes`, os nomes de todos os métodos que realizam testes.

Depois de codificadas todas as classes de teste, é criada uma classe responsável por invocar todos os métodos *suite* das classes de teste. A Figura 15 ilustra este método da classe que será responsável por testar todos os outros casos de teste.

Cada método *suite* que se quer testar é adicionado à *suite* de teste, através de `addTest`. A instância da classe `TestSuite` contém todos os métodos das classes de teste que se quer executar.

```
public Test suite() {
      TestSuite suite = new TestSuite();

      suite.addTest(new TestConta().suite());
      suite.addTest(new TestPoupanca().suite());
      return suite;
}
```

Figura 15. Exemplo de utilização do método *suite* da classe que irá invocar todos os métodos de teste.

Para executar os casos de teste é criada uma subclasse de `j2meunit.midlet.TestRunner`, nesta classe é necessário implementar o método `startApp` e chamar o método `start`, passando como parâmetro a *suite* de teste que será executada. Na Figura 16, segue um exemplo de como executar os casos de teste.

```
protected void startApp()
{
      start(new String[] { "j2meunit.pacote.classe" });
}
```

Figura 16. Exemplo de utilização do método *startApp* da classe `j2meunit.midlet.TestRunner`.

O J2MEUnit ao contrário do JUnit não foi desenvolvido para ser utilizado também como um *plug-in* do Eclipse, é possível utilizar este *framework* no Eclipse fazendo referencia ao o arquivo *.jar* do J2MEUnit , configurando o ambiente, e re-configurando cada vez que a classe a ser executada é trocada. Por isso a necessidade de se ter um *framework* capaz de realizar testes de aplicativos J2ME dentro do Eclipse, por ser uma ferramenta que facilita o trabalho do desenvolvedor.

3.3 Plataforma J2ME

Esta Seção aborda definições e arquitetura da plataforma de desenvolvimento J2ME, bastante utilizada hoje em dia e que possui grande crescimento de usuários e de novos aplicativos. São exibidos exemplos com explicações de codificações de aplicativos J2ME para melhor entendimento de sua utilização.

É cada vez mais comum encontrarmos sistemas embutidos no dia a dia das pessoas, como nos carros, eletrodomésticos, celulares, caixas eletrônicos de bancos, dentre outros. Estes sistemas apresentam restrições de velocidade de processamento de dados, tamanho físico e requisitos de memória [20].

Os sistemas embarcados são projetos bastante específicos, tendo em vista que, só alguns deles possuem sistema operacional, pois geralmente eles são tão especializados que suas tarefas podem ser implementadas em um único programa.

A linguagem de programação Java possui características que a tornam ideal para o desenvolvimento de sistemas embarcados, como portabilidade, reuso de código, segurança, conectividade com a WEB, dentre as facilidades que se tem em usar Java para desenvolver projetos, como a quantidade e qualidade de ferramentas e os conceitos de orientação a objeto.

A *Sun Microsystems*, empresa responsável pela criação da linguagem JAVA, reagrupou suas tecnologias em três plataformas diferentes: *Java 2 Platform Enterprise Edition* (J2EE), *Java 2 Platform Standard Edition* (J2SE), e *Java 2 Platform Micro Edition* (J2ME) [6]. Cada uma dessas edições trabalha com um tipo de mercado especifico.

Algumas APIs (Interface para Programação de Aplicativos) da plataforma J2SE são usadas nas outras plataformas existentes como subconjuntos básicos. Através da utilização desta plataforma é possível desenvolver aplicações convencionais. O alvo da plataforma J2EE são aplicações Web, distribuídas e transacionais, ou seja, aplicações utilizadas para manter informações numa rede de comunicação, geralmente a Internet.

A plataforma J2ME é utilizada para dispositivos com menor poder computacional, geralmente para dispositivos móveis. Seu alvo de trabalho pode ser dividido em duas classes: para dispositivos pessoais, móveis, como telefones celulares, *pagers* e PDAs (um dos primeiros computadores que podiam ser "locomovidos"); ou para dispositivos compartilhados, fixos, como os sistemas de navegação automotiva e comunicadores *high-end*. Este trabalho tem como foco testar aplicações baseadas nesta plataforma.

Apesar da grande vantagem de Java de se poder executar um programa em qualquer lugar compilando uma só vez, em sistemas embutidos não é possível se aproveitar desta característica. Sistemas embarcados possuem restrições de memória, o que impossibilita carregar num dispositivo todas as classes necessárias para a execução de um programa. Por este motivo dividiram-se os sistemas embutidos em subclasses, as quais podem utilizar as seguintes plataformas:

- *PersonalJava*: utilizado por dispositivos que tem acesso a Internet, interconectados em rede;
- *EmbeddedJava*: para dispositivos embarcados de função dedicada, dispositivos embutidos;
- *Java Card*: usado em cartões inteligentes;
- *Java 2 Micro Edition* (J2ME): abrange uma ampla área de dispositivos, como celulares, PDAs (*Personal Digital Assistant Profile*), *pagers*, dentre outros [21].

Cada subclasse destes dispositivos não tem acesso a API de sua plataforma, apenas executam as funções disponibilizadas pela máquina virtual. É por este motivo que não há transmissão de vírus entre celulares, o Java não tem acesso a API do celular, então não pode apagar ou modificar seus dados sem intervenção humana, a não ser que o fabricante disponibilize a API do celular ou que o proprietário permita a utilização do meio de transmissão *Bluetooth* em seu aparelho. O caso de disponibilizar a API, ocorre durante a instalação de jogos nos celulares. E como para cada fabricante há uma API diferente, a característica de Java "*Write Once, Run Anywhere*" é perdida neste caso, isto porque cada aparelho de fabricante diferente teria que ter sua própria API, ao invés de uma única API que pudesse ser executada em qualquer aparelho.

Embora os dispositivos móveis tenham muito em comum, eles diferem na forma, função e características, por isso existe o conceito de configuração e de perfis, para diferenciar as necessidades de cada aplicação. A Figura 17 ilustra esta divisão de camadas que há na plataforma J2ME. Dependendo da configuração do sistema, uma aplicação escrita na linguagem Java será interpretada por uma máquina virtual KVM (*Kilo Virtual Machine*) [22] ou JVM (*Java Virtual Machine*) [23]. Definida a configuração da aplicação, escolhe-se o perfil de acordo com cada configuração.

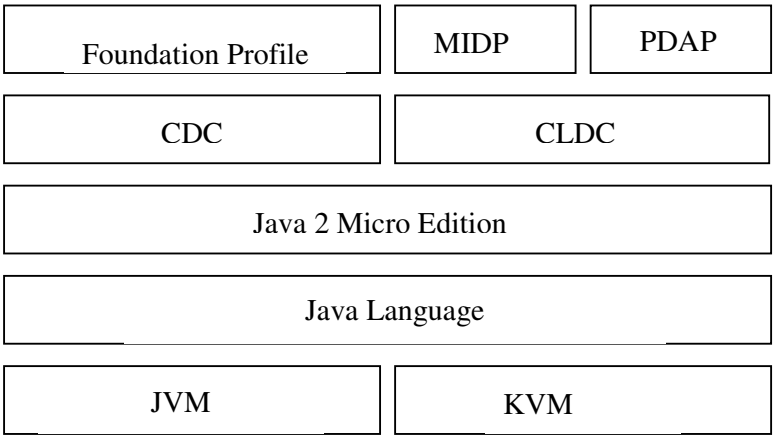

Figura 17. Camadas de *software* do J2ME

A máquina virtual KVM interpreta códigos gerados com a tecnologia J2ME, respeitando as limitações de hardware dos dispositivos moveis, ou seja, é uma otimização da máquina virtual (VM) para aplicativos J2ME. Já a máquina JVM é mais completa, capaz de interpretar códigos gerados com a tecnologia J2SE, é usada para permitir que aplicações Java possam rodar em qualquer plataforma, desde que elas tenham a JVM instalada.

Através da configuração define-se um denominador comum que suporta uma determinada categoria de dispositivos. As configurações podem ser do tipo CDC (*Connected Device Configuration*) ou CLDC (*Connected Limited Device Configuration*).

A configuração do tipo CDC é suportada por dispositivos com ampla variação de capacidade computacional, como PDAs poderosos, *settop boxes* digitais, *screen-phones*, dentre outros. A configuração do tipo CLDC contém uma API mínima que permite rodar aplicativos em dispositivos móveis com limites de processamento e memória, tais como, telefones celulares, *pagers* e *smartphones*. Estas APIs possibilitam o desenvolvimento de aplicativos em qualquer dispositivo que possua suporte às mesmas.

Os perfis, ou *profile*, são um conjunto de bibliotecas (APIs) que são mais específicas a uma categoria de dispositivos do que às bibliotecas disponíveis pela configuração.

Os perfis exibidos na Figura 17 são: MIDP (*Mobile Information Device Configuration*), que contém APIs específicas para o desenvolvimento de aplicações para celular; PDAP (*Personal Digital Assistant Profile*) contém APIs para PDAs; o perfil *Foundation Profile*, que estende a configuração CDC e é a base para todo perfil CDC. Outros perfis são a TV *Profile*, usada para desenvolvimento de aplicações de televisões digitais e também o *Bluetooth*.

Enquanto a Configuração descreve de forma geral uma família de dispositivos, os perfis são mais específicos para um tipo particular de aparelho de uma determinada família.

A configuração do tipo CLDC juntamente com o perfil MIDP, que contém os pacotes básicos *javax.microedition.Icdui* (interface com o usuário), *javax.microedition.rms* (sistema de gerência de registros para persistência de informações) e o *javax.microedition.midlet* (suporte para aplicações MIDP), são mais comumente utilizados para desenvolvimento de dispositivos móveis de baixo custo, como PDAs e telefones celulares. Esta configuração juntamente com o perfil MIDP, faz parte da configuração da plataforma J2ME usada neste trabalho.

As MIDlets são aplicações desenvolvidas sobre a plataforma J2ME. Um dos pontos importantes das aplicações J2ME refere-se ao ciclo de vida das MIDlets. Cada dispositivo contém um AMS (*Aplication Manager System*), o qual é responsável por gerenciar os aplicativos a serem instalados, onde e como serão armazenados. Quando a MIDlet é chamada o AMS é responsável por invocar o método `startApp`, inicializando a execução da MIDlet. O método

`pauseApp` é chamado para liberar recursos temporariamente, ou seja, ao receber uma mensagem de texto, esta é pausada para atender uma chamada. O método `destroyApp` é invocado quando se quer liberar todos recursos utilizados pela MIDlet. A Figura 18, ilustra o ciclo de vida de uma MIDlet.

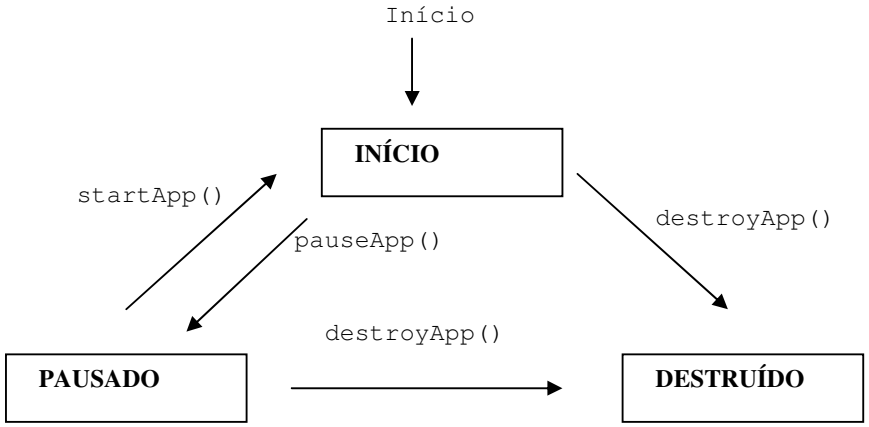

Figura 18. Ciclo de Vida da MIDlet [6].

O *software* AMS já vem instalado na MIDP, e é o responsável por fazer as chamadas aos métodos mostrados no ciclo de vida da MIDlet. No exemplo a seguir será mostrado como se da a implementação destes métodos.

O exemplo mostrado a seguir faz uso do perfil MIDP, e é exibido para melhor explicar o funcionamento de dispositivos móveis. É um exemplo básico que faz uso da classe `Canvas` e `Display`, utilizadas para desenhar no display e gerenciar a exibição de objetos no dispositivo, respectivamente.

O *namespace javax.microedition.midlet* inclui a classe MIDlet para o aplicativo, que utiliza os três métodos (`startApp`, `pauseApp` e `destroyApp`) para permitir a comunicação com os aplicativos que estão rodando. O nome da classe principal deste exemplo é `HelloWorld`, Figura 19 que estende a classe MIDlet, e dentro da qual, criamos atributos do tipo `Display` e `HelloCanvas`.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class HelloWorld extends MIDlet {
 private boolean paused;
 private Display myDisplay;
 private HelloCanvas myCanvas;
 public HelloWorld() {
  paused = false;
 }
```

Figura 19. Trecho de código do exemplo de aplicativos J2ME.

O método `startApp` é chamado cada vez que um aplicativo é ativado, não somente quando o aplicativo é iniciado. Sendo bastante útil para iniciar os aplicativos durante as fases de transição destes, do estado de ativa para inativa, ou vice-versa, transição esta que acontece várias vezes enquanto a aplicação está sendo rodada. Este método está exemplificado na Figura 20.

```
public void startApp()  {
  //Verifica se é a primeira vez que a aplicação está rodando ou se a VM
retornou do estado de pausa
  if( paused ) {
   //volta da pausa
   myCanvas.repaint();
  }
  else {
   //cria o form
   myDisplay = Display.getDisplay( this );
   myCanvas = new HelloCanvas();
   myDisplay.setCurrent( myCanvas );
  }
 }
```

Figura 20. Trecho de código do exemplo de aplicativos J2ME com o método `startApp`.

O método `pauseApp`, Figura 21, é utilizado várias vezes durante a execução do dispositivo, pois permite que um aplicativo que esteja rodando seja pausado para permitir que outro aplicativo seja ativado. Este método é bastante útil pelo fato de que a maioria dos dispositivos móveis não possui suporte a multiprocessamento de tarefas.

```
public void pauseApp() {
  //pausa a aplicação
  paused = true;
  notifyPaused();
 }
```

Figura 21. Trecho de código do exemplo de aplicativos J2ME com o método `pauseApp`.

O método `destroyApp`, Figura 22, é chamado para indicar que uma aplicação está prestes a ser terminada. Ao contrário do método `startApp`, este método só é chamado uma única vez durante a execução do aplicativo.

```
public void destroyApp( boolean unconditional ) {
  // diz ao sistema que o midlet está pronto para ser destruído
notifyDestroyed();
 }
```

Figura 22. Trecho de código do exemplo de aplicativos J2ME com o método `destroyApp`.

A classe `HelloCanvas`, Figura 23, foi criada para ilustrar um exemplo de uma classe que herda a classe `Canvas`, responsável por proporcionar uma área para gráficos. É usada para exibir mensagens diferentes quando os aplicativos estão pausados ou não estão.

```
class HelloCanvas extends Canvas {

  public void paint( Graphics g ) {
   g.setColor( 0x0FFFFFF );
   //preenche a tela
   g.fillRect( 0, 0, getWidth(), getHeight() );
   g.setColor( 0 );
   if( paused ) {
    paused = false;
    g.drawString("Voltei da pausa.", 0, 0, Graphics.TOP | Graphics.LEFT );
   }
   else {
    g.drawString("Olá, pessoal!", 0, 0, Graphics.TOP | Graphics.LEFT );
   }
}
}
}
```

Figura 23. Trecho de código do exemplo de aplicativos J2ME.

No exemplo acima quando o aplicativo está pausado o próximo passo da execução é configurar a variável `paused` para `false`, para que ela possa continuar rodando, e é exibido a mensagem "Voltei da pausa", indicando que a pausa acabou.

Quando o aplicativo não está pausado a mensagem apresentada é: "Olá, pessoal!", e o aplicativo continua rodando normalmente.

Através dos exemplos e definições citadas nesta Seção é possível entender que aplicativos móveis trabalham de forma diferente dos aplicativos Java comuns, por isso necessitam de sua API própria e de técnicas de teste de *softwares* diferenciadas. Com a grande disseminação do uso de aparelhos celulares são criadas novos aplicativos que podem ser acoplados aos aparelhos e que precisam ser testados, desta forma, estas técnicas de teste, assim como ferramentas, tornam-se cada vez mais específicas para cada fabricante.

Capítulo 4

Teste de Unidade de Aplicativos J2ME no Eclipse

Para prover suporte ao teste de unidade de aplicações escritas em J2ME dentro do Eclipse, realizamos a integração do J2MEUnit dentro do mesmo. Por outro lado, o desenvolvimento de aplicações J2ME pode ser realizado no Eclipse com o uso da API J2ME. Contudo, esta API não apresenta suporte adequado ao teste de aplicações J2ME.

Neste Capítulo apresentamos, inicialmente, como realizamos a integração do J2MEUnit ao Eclipse, assim como a interface que exibe os resultados, baseada na interface do JUnit. Estas integrações resultam no *plug-in* EclipseJ2MEUnit.

Inicialmente o objetivo do projeto seria a extensão do *plug-in* JUnit, fazendo com que este permitisse o teste de unidade para aplicativos J2ME além de aplicações Java, baseados na plataforma J2SE, já suportados pelo JUnit. Porém, esta atividade de teste de unidade de aplicativos J2ME já é suportada pelo *framework* J2MEUnit, que não foi construído para ser utilizado como um *plug-in* no Eclipse. Procuramos evitar duplicação desnecessária de código, além de economizar tempo do projeto, visto que o J2MEUnit já é bastante utilizado pelos desenvolvedores e possui base sólida para permitir o desenvolvimento de teste, sabendo que já foram lançadas várias versões deste *framework* com correções de problemas e acréscimo de melhorias.

O desenvolvimento do *plug-in* EclipseJ2MEUnit foi baseado em reuso, reutilizamos o J2MEUnit para fazer com que este *framework* pudesse ser utilizado como um *plug-in* do Eclipse. O reuso não aconteceu como o reuso de sistemas COTS (*Components "Off-The-Shelf"*), sistemas propriamente ditos, que realizam funções específicas e podem ser acoplados a outros sistemas sem necessitar de mudanças. Reutilizamos todo o projeto J2MEUnit, porém, mudanças foram feitas para permitir que a interface AWT com os resultados fosse exibida e que outros *plug-ins* pudessem sem acoplados ao EclipseJ2MEUnit.

Observando-se também que o EclipseME [24], *framework* que ajuda na construção de aplicativos J2ME, não suporta teste de unidade para aplicativos J2ME, sentimos a necessidade de permitir que desenvolvedores J2ME pudessem testar seus aplicativos dentro da mesma IDE na qual desenvolveram os aplicativos. Esta atenção dada ao teste de aplicativos J2ME deve-se à importância do teste de *software* durante o processo de desenvolvimento de projetos.

Na Seção 4.1 é apresentado o *plug-in* EclipseJ2MEUnit e as classes principais deste *plug-in*. A Seção 4.2 apresenta o *plug-in* EclipseJ2MEUnit_ui construído para auxiliar o processo de

execução de aplicativos baseados no *plug-in* EclipseJ2MEUnit, uma nova configuração foi adicionada ao menu "*Run*" do Eclipse.

Na Seção 4.3 os pontos de extensão criados neste projeto são apresentados, descrevemos detalhadamente a implementação destes pontos de extensão assim como exemplos de trechos de código do arquivo manifesto são apresentados para auxiliar a extensão deste projeto.

4.1 EclipseJ2MEUnit

A possibilidade de se desenvolver testes em um ambiente de desenvolvimento como o Eclipse, descrito na Seção 3.1, o qual possibilita a integração com outros *plug-ins*, possui interface amigável, dentre outras vantagens, simplifica o processo de desenvolvimento de aplicativos. O *plug-in* EclipseJ2MEUnit permite que os testes de aplicativos J2ME sejam desenvolvidos na IDE Eclipse por introduzir o conceito de *plug-in* durante o desenvolvimento do projeto, acrescentando um *framework* que permite o teste para os aplicativos desenvolvidos, além de possuir uma interface gráfica com os resultados dos testes realizados.

O *framework* de teste foi construído a partir do *framework* de testes J2MEUnit, que permite o teste de aplicativos baseados na plataforma J2ME. Em algumas classes não houve necessidade de modificação, mas em outras, algumas modificações foram feitas para adaptar a execução dos testes e exibição dos resultados tanto no console do Eclipse quanto numa interface do tipo AWT (*Abstract Window Toolkit*), interface esta que o J2MEUnit não disponibiliza.

A classe `TestRunner` é responsável por executar as classes de teste, esta classe foi definida no projeto J2MEUnit e reimplementada neste projeto, pois houve necessidade de agrupar funcionalidades básicas numa classe `BaseTestRunner` para que as classes `TestRunner` dos pacotes `eclipseJ2MEUnit.textui` e `eclipseJ2MEUnit.awtui` possam utilizá-la.

Existem dois pacotes diferentes implementando a classe `TestRunner`, pois no pacote `eclipseJ2MEUnit.textui` os testes são executados e os resultados exibidos textualmente no console do Eclipse. A Figura 24 exibe um esqueleto desta classe e de alguns de seus métodos.

```
package eclipseJ2MEUnit.textui;

public class TestRunner extends BaseTestRunner{
      protected void start(String[] args)
      {
            String testCaseName = processArguments(args);
            ...
            Test     suite    = testCase.suite();
            doRun(suite);
      }
      protected void doRun(Test suite)
      {
            TestResult result = createTestResult();
            ...
            suite.run(result);
            ...
            print(result);
            fWriter.println();
            ...
      }
}
```

Figura 24. Esboço da classe `TestRunner` do pacote eclipseJ2MEUnit.textui.

Como pode-se observar a classe `TestRunner` da Figura 24 está dentro do pacote `eclipseJ2MEUnit.textui` e estende a classe `BaseTestRunner`. O método `start` é invocado durante a execução das classes de teste e é responsável por capturar a *suite* de teste e executá-la através do método `doRun`, que executa a *suite* de teste e exibi os resultados no console.

O *framework* J2MEUnit não disponibiliza a visualização em uma tela com os resultados do teste, nem numa interface do tipo AWT, nem numa interface *Swig*, os resultados são exibidos em forma de texto. Para permitir que a classe `TestRunner` executasse os testes e exibisse eles numa interface AWT foi necessário modificar a classe `TestRunner` do J2MEUnit e acrescentar a classe `BaseTestRunner`, para que as classes `TesRunner` dos pacotes `eclipseJ2MEUnit.awtui` e `eclipseJ2MEUnit.textui` pudessem implementar esta classe.

Como houve esta mudança na classe `TestRunner` do J2MEUnit e conseqüentemente em outras classes também, não foi possível utilizar este *framework* como uma camada da arquitetura de camadas em que a interface apenas utilizasse os serviços providos pelo J2MEUnit.

Assim como no pacote `textui`, a classe `TestRunner` do pacote `eclipseJ2MEUnit.awtui` executa o método `start`, que invoca o método `createUI`, responsável por criar a interface gráfica. O método `runSuite` inicia a *thread* responsável pela execução dos testes da *suite* de teste. Na Figura 25 encontra-se um esboço destes métodos da classe `TestRunner`.

Diferente da classe `TestRunner` da Figura 24, a classe da Figura 25, executa a *suite* de teste através de um *thread*.

```
package eclipseJ2MEUnit.awtui;
public class TestRunner extends BaseTestRunner {
      public void start(String[] args) {
            String suiteName= processArguments(args);
            fFrame= createUI(suiteName);
            ...
            runSuite();
      }

      protected Frame createUI(String suiteName){
            Frame frame= new Frame("EclipseJ2MEUnit");
            ...
      }

      synchronized public void runSuite() {
                  ...
                  fRunner= new Thread()
                  ...
                  fRunner.start();
      }
}
```

Figura 25. Esboço da classe `TestRunner` do pacote `eclipseJ2MEUnit.awtui`.

Outra importante classe que foi reusada, a classe `TestCase` é estendida pelas classes de teste para permitir que seus métodos fossem utilizados nas classes de teste. A Figura 26 ilustra um esboço desta classe e alguns de seus métodos.

```
package eclipseJ2MEUnit.framework;
public abstract class TestCase extends Assert implements Test{
...
      public String getTestMethodName() ...
      public TestResult run() ...
      public Test suite() ...
      protected void setUp() ...
      protected void runTest() ...
      protected void tearDown() ...
}
```

Figura 26. Esboço da classe `TestCase`.

O método `getTestMethodName` retorna uma *String* com o nome do método que irá executar o teste. Os métodos `setUp` e `tearDown` são utilizados para configurar os testes, o método `suite` é invocado para permitir a criação da *suite* de teste da classe, onde todos os testes a serem executados são chamados. Os métodos `run` e `runTest` são chamados para executar os testes da classe de teste.

A arquitetura do *plug-in* através da exibição de algumas de suas classes é apresentada na Figura 27.

A interface `Test` coleta os resultados para a execução dos testes e permite que a classe abstrata `TestCase` e a classe `TestSuite` implementem seus métodos. A classe `TestCase`, como já foi mencionada anteriormente, define a estrutura dos casos de teste. A classe `Assert`, estende a classe `TestCase`, nela encontramos as assertivas responsáveis por analisar o resultados dos testes através dos métodos `assertEquals`, `assertTrue`, dentre outros. A classe `TestSuite` representa uma coleção de instâncias de `Test` que podem ser executadas todas de uma só vez.

A interface `IBaseTestRunner` que foi criada para permitir que um ponto de extensão fosse feito usando a mesma como entidade a ser implementada. A interface `TestListener` contém operações que permitem o acompanhamento dos testes, como os métodos `addError` e `startTest`, dentre outros que existem nesta interface. A classe abstrata `BaseTestRunner` é estendida pelas classes `TestRunner` e implementa as interfaces `IBaseTestRunner` e `TestListener`, contém as operações básicas para executar os testes, independente da forma como eles serão exibidos.

As responsabilidades das classes `TestRunner` já foram citadas, ambas estendem a classe abstrata `BaseTestRunner` e implementam o método abstrato `testStarted`, cada uma de forma particular. Como se pode observar na Figura 27, uma classe `TestRunner` tem associada a ela a classe `ResultPrinter`, esta `TestRunner` refere-se a classe do pacote `eclipseJ2MEUnit.textui` e exibe os resultados no console do Eclipse, utilizando a classe `ResultPrinter` para definir operações específicas do processo de exibição no console.

A classe `TestRunner` do pacote `eclipseJ2MEUnit.awtui` utiliza a classe `ProgressBar` e outras, para exibir uma interface gráfica com os resultados do teste. Existe ainda outra classe `TesRunner` do pacote *midletui* que é implementado como uma MIDlet que pode ser executada pelo WTK (*Wireless Toolkit*).

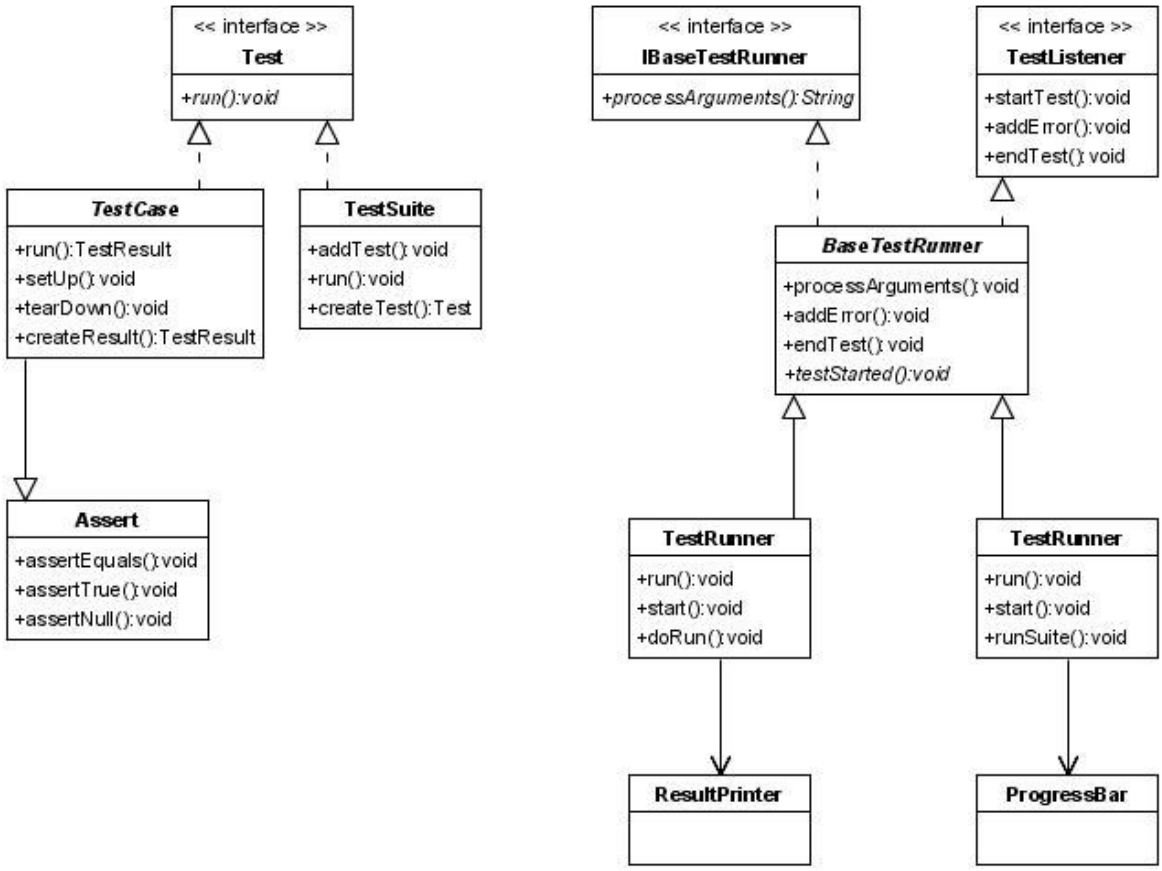

Figura 27. Diagrama de classes do *plug-in* EclipseJ2MEUnit.

A tela com os resultados do teste é exibida quando a classe `TestRunner` do pacote `eclipseJ2MEUnit.awtui` é executada. Dentro deste pacote há outras classes que auxiliam a exibição desta tela. A Figura 28 ilustra um esboço da classe `TestRunner`, e os principais métodos utilizados para gerar a tela da Figura 29.

```
package eclipseJ2MEUnit.awtui;
public class TestRunner extends BaseTestRunner {
      protected Frame createUI(String suiteName) ...
      protected Panel createCounterPanel() ...
}
```

Figura 28. Esboço da classe eclipseJ2MEUnit.awtui.TestRunner.

O método `createUI`, como já foi mencionado anteriormente, na Figura 25, é invocado quando inicia a execução dos testes e é responsável por inicializar as configurações para exibição da tela de resultados e por invocar as classes que auxiliam a exibição desta tela, como a classe `ProgressBar`, responsável pela barra de progresso com o resultado do teste. O método `createCounterPanel` indica a quantidade de testes executados e a quantidade de testes que falharam ou erraram.

Assim como o JUnit, que possibilita a exibição desta tela de resultados com a interface AWT ou *Swing* este projeto se guiou pelo *framework* JUnit para gerar esta tela. Apesar do fácil reuso das classes disponíveis pelo JUnit, a classe em que houve mais modificações foi a classe

`TestRunner`, para adaptar-se aos testes do *plug-in* EclipseJ2MEUnit. Percebe-se que a tela gerada neste projeto tem a mesma estrutura das telas geradas pelo JUnit, facilitando a adaptação dos usuários do *plug-in* EclipseJ2MEUnit.

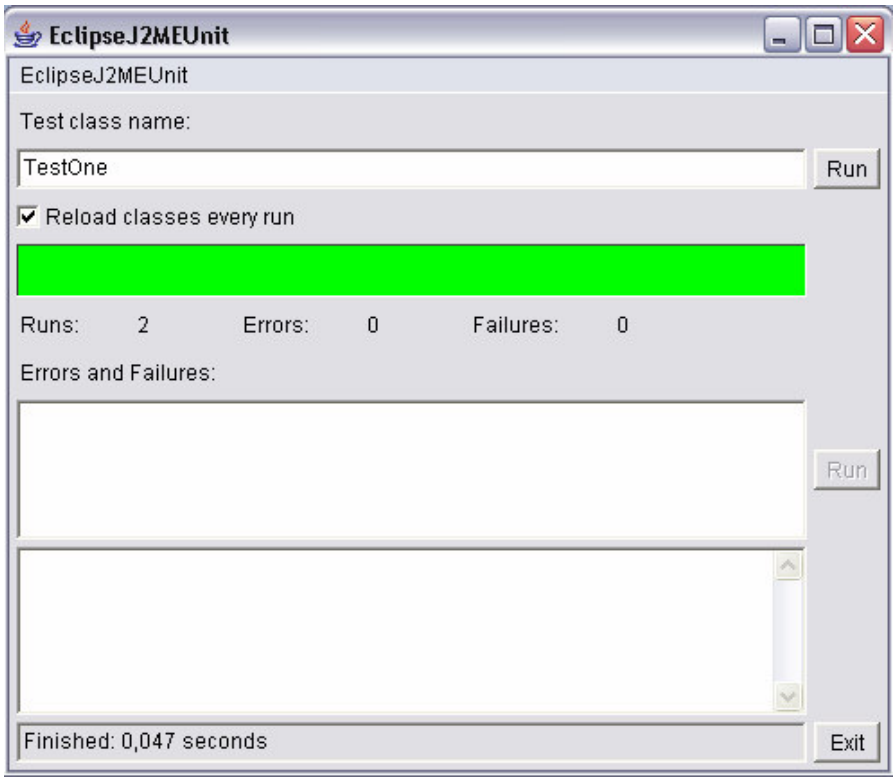

Figura 29. Tela de Exibição dos resultados.

A tela exibe a quantidade de testes executados, no campo *Runs*, a quantidade de teste com erros em *Erros*, e com falhas em *Failures*. Erros se referem a problemas ocorridos durante a execução, e a falha está relacionada ao resultado esperado da assertiva.

A barra de progresso indica a situação do teste, barra verde indica que não há falhas nem erros nos testes, barra vermelha indica que há alguma falha ou erro nos teste.

A interface do tipo AWT foi escolhida pra ser reusada por ser mais simples de reusar e satisfazer o objetivo deste projeto, ou seja, tornar a interface mais amigável ao usuário. Para que seja possível ter uma interface do tipo *Swing* seria necessário implementar algumas classes a mais para permitir que esta interface pudesse ter o mesmo padrão que o adotado pelo JUnit. Como a interface AWT satisfaz os objetivos do projeto, o desenvolvimento da interface *Swing* poderá ser feita como um trabalho futuro.

A utilização deste *plug-in* por desenvolvedores de aplicativos J2ME facilita a atividade de teste do processo de desenvolvimento, permitindo que as classes de teste sejam desenvolvidas na IDE Eclipse e que os resultados sejam mais rapidamente analisados com a utilização desta tela exibindo os resultados, no Apêndice A.1 está descrito o arquivo manifesto do *plug-in* EclipseJ2MEUnit. O *plug-in* EclipseJ2MEUnit_ui procura facilitar ainda mais a tarefa do desenvolvedor através da configuração da execução dos testes no *plug-in* EclipseJ2MEUnit.

4.2 EclipseJ2MEUnit_ui

A API JDT (*Java Development Toolkit*) permite que programadores Java possam editar, testar, programar, usar modo *debug* e rodar aplicações Java, o *plug-in* EclipseJ2MEUnit_ui baseia-se nesta API para desenvolver uma nova forma de se executar aplicativos do *plug-in* EclipseJ2MEUnit. Esta Seção descreve como usar o módulo EclipseJ2MEUnit_ui e uma descrição detalhada da implementação do EclipseJ2MEUnit no *plug-in* EclipseJ2MEUnit_ui.

A Figura 30 mostra o menu *Run* da barra de ferramentas do Eclipse, este menu é utilizado para executar aplicativos e dentro dele existem entradas de menu *Run As*. Estes menus permitem aos usuários configurar uma configuração de execução de um aplicativo.

Uma configuração de execução descreve atributos necessários para que o aplicativo seja executado. Por exemplo, deve-se definir qual o projeto no qual a classe a ser executada encontra-se, qual a classe que será executada, se existe alguma dependência de outras APIs, dentre outros atributos.

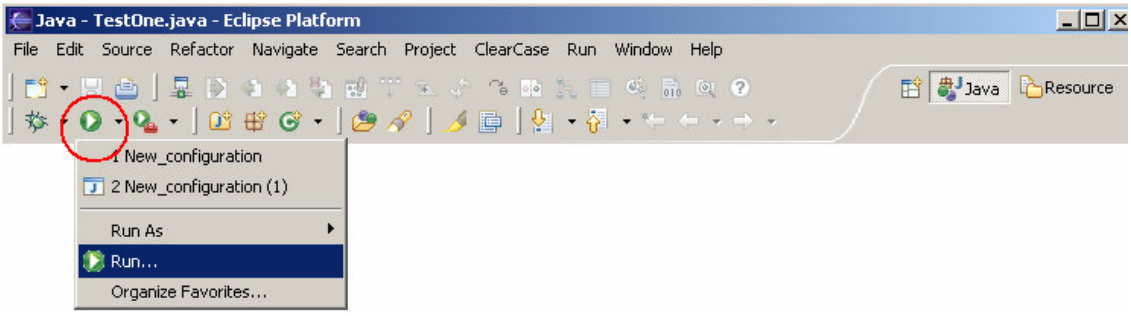

Figura 30. Barra de ferramentas do Eclipse, focando no menu *Run*.

Após clicar na opção *Run...* uma janela com diversas configurações de execução é apresentada. A Figura 31 mostra esta janela, do lado esquerdo encontram-se todos os tipos de aplicativos cuja execução podemos configurar, e do lado esquerdo está a aberta a visão para configurar os aplicativos EclipseJ2MEUnit. Nesta visão existem três abas, são elas: `EclipseJ2MEUnit Main`, `Program Arguments` e `Common`.

Na aba `EclipseJ2MEUnit Main` definimos que projeto contém a classe a ser executada no campo `Project`, e no campo `Main class` a classe que será chamada para executar os testes desejados. Este campo está configurado para executar a classe `eclipseJ2MEUnit.awtui.TestRunner` do EclipseJ2MEUnit, mas este pode ser alterado para executar também a classe `eclipseJ2MEUnit.textui.TestRunner`.

A aba `Arguments` contém dois campos, no primeiro define-se os argumentos do programa, `Program Arguments`, que será o nome da classe a ser testada, no segundo campo define-se os argumentos da máquina virtual, `VM Arguments`, neste campo não precisa colocar nenhum valor.

A aba `Common` é comum a todas as visões de configuração de execução e nela podemos definir se a execução será local ou compartilhada. Esta aba geralmente não precisa ser modificada.

Para se executar aplicativos EclipseJ2MEUnit basta completar os campos necessários nesta janela e apertar no botão *Run*.

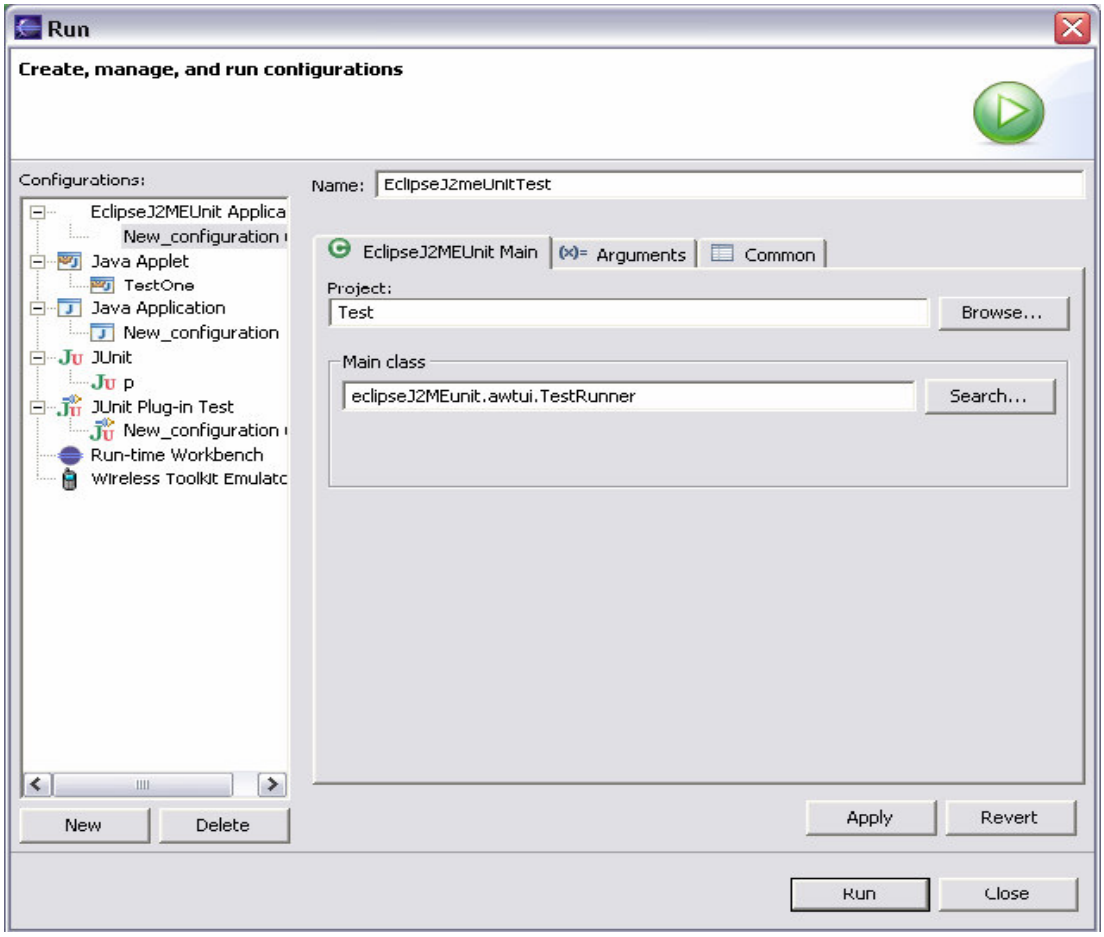

Figura 31. Janela com configuração de execução dos aplicativos.

Para que esta configuração de execução seja feita é preciso estender alguns pontos de extensão do Eclipse e implementar algumas interfaces. O tipo de configuração de execução é usado quando se deseja executar aplicações Java locais, e é o primeiro ponto de extensão que deve-se definir ao criar uma nova configuração de execução [25].

A Figura 32 contém a declaração da extensão ao ponto de extensão `org.eclipse.debug.core.launchConfigurationTypes` que esta definido no atributo `point`. No Apêndice A.2 podemos observar todo o arquivo manifesto deste *plug-in*.

```
<extension
    point="org.eclipse.debug.core.launchConfigurationTypes">
    <launchConfigurationType
            name="EclipseJ2MEUnit Application"
          delegate= "eclipseJ2MEUnit.launch.JavaLocalLaunchConfiguration"
            modes="run"
            id="EclipseJ2MEUnitconfigurationType">
    </launchConfigurationType>
</extension>
```

Figura 32. Trecho do arquivo manifesto com a extensão ao ponto `org.eclipse.debug.core.launchConfigurationTypes`.

A utilização deste ponto de extensão implica na implementação da interface `org.eclipse.debug.core.model.ILaunchConfigurationDelegate`, que define a execução da aplicação para o tipo de configuração especificado [26]. O atributo `delegate` indica que a classe `eclipseJ2MEUnit.launch.JavaLocalLaunchConfiguration` do projeto irá implementar a interface `ILaunchConfigurationDelegate`. Os atributos `name` e `id` também são especificados, o atributo `id` tem grande importância nesta extensão pois será necessário invocá-lo ao se criar as abas da janela de configuração. O atributo `mode` indica a forma de execução do aplicativo, pode ser no modo *run*, *debug*, ou em ambos os casos. Nesta extensão só será necessário utilizar o modo `run`, para executarmos os aplicativos no *plug-in* EclipseJ2MEUnit.

A classe `JavaLocalLaunchConfiguration`, que implementa a interface `ILaunchConfigurationDelegate` deve implementar o método `launch`, cujo objetivo é executar os aplicativos, recuperando as informações definidas nos argumentos da janela de configuração de execução (Figura 31). A Figura 33 contém trechos da implementação desta classe.

```
package eclipseJ2MEUnit.launch;

      public class JavaLocalLaunchConfiguration extends
      AbstractJavaLaunchConfigurationDelegate implements
      ILaunchConfigurationDelegate {

      public void launch(ILaunchConfiguration configuration, String mode,
      ILaunch launch, IProgressMonitor monitor){
      . . .
```

Figura 33. Esboço da classe `JavaLocalLaunchConfiguration`.

O parâmetro `configuration` armazena todas as informações necessárias para executar o aplicativo. O parâmetro `mode` descreve a forma de execução do aplicativo, `launch` é um objeto que representa a execução e `monitor` é responsável por exibir o progresso dos testes na barra de progresso.

Além de configurar a execução dos aplicativos através desta classe, que também executa o aplicativo, é preciso criar a janela que irá exibir os campos para o usuário preenchê-los. Esta nova visão que aparece na janela de configuração é criada através da extensão a uma classe que cria grupos de abas para configuração da execução. A Figura 34 mostra o trecho do arquivo manifesto no qual esta extensão foi feita.

```
<extension
 point="org.eclipse.debug.ui.launchConfigurationTabGroups">
  <launchConfigurationTabGroup
      type="EclipseJ2MEUnitconfigurationType"
      class="eclipseJ2MEUnit.launch.EclipseJ2MEUnitTabGroup"
      id="EclipseJ2MEUnitTabGroup">
  </launchConfigurationTabGroup>
</extension>
```

Figura 34. Trecho do arquivo manifesto com extensão ao ponto `org.eclipse.debug.ui.launchConfigurationTabGroups`.

Como se observa na Figura 34 o ponto de extensão é o `eclipse.debug.ui.launchConfigurationTabGroups`, implementado pela classe `eclipseJ2MEUnit.launch.EclipseJ2MEUnitTabGroup`, como está descrito no atributo `class`. Ao estender este ponto de extensão é preciso declarar o atributo `type`, o qual recebe como valor a

mesma *string* que foi dada ao atributo `id` na extensão ao ponto `org.eclipse.debug.core.launchConfigurationTypes` explicado anteriormente.

Ao se utilizar este ponto de extensão é preciso que a classe definida no atributo `class` implemente a interface `org.eclipse.debug.ui.ILaunchConfigurationTabGroup`, interface que chama as abas que irão aparecer na janela de configuração de execução do aplicativo, como está ilustrado na Figura 35.

```
package eclipseJ2MEUnit.launch;

public class EclipseJ2MEUnitTabGroup extends
AbstractLaunchConfigurationTabGroup implements ILaunchConfigurationTabGroup{

      public void createTabs(ILaunchConfigurationDialog arg0, String arg1) {
            ILaunchConfigurationTab[] tabs = new ILaunchConfigurationTab[] {
                  new EclipseJ2MEUnitTab(),
                  new JavaArgumentsTab(),
                  new CommonTab()
            };
            setTabs(tabs);
      }
}

}
```

Figura 35. Classe `EclipseJ2MEUnitTabGroup`.

Como se pode observar o método `createTabs` inicia as abas que irão ser exibidas na janela de configuração. A classe `EclipseJ2MEUnitTab` implementa a aba na qual será descrita a classe que irá executar os testes.

Esta classe `EclipseJ2MEUnitTab`, Figura 36, estende a classe `JavaMainTab`, que é responsável por exibir e editar o projeto e o nome da classe principal. O método `updateMainTypeFromConfig` inicia o nome da classe principal para `eclipseJ2MEUnit.awtui.TestRunner`, classe responsável por executar os testes do projeto.

```
package eclipseJ2MEUnit.launch;
public class EclipseJ2MEUnitTab extends JavaMainTab {

protected void updateMainTypeFromConfig
(ILaunchConfiguration config) {
      String mainTypeName = "EclipseJ2MEUnit.awtui.TestRunner";
      fMainText.setText(mainTypeName);
      }
      . . .
}
```

Figura 36. Esboço da classe `EclipseJ2MEUnitTabGroup`.

A Figura 37 apresenta parte das classes do *plug-in* EclipseJ2MEUnit_ui e sua arquitetura. As classes `JavaMainTab`, `IJavaLaunchConfigurationDelegate` e `ILaunchConfigurationTabGroup` não foram desenvolvidas neste projeto e só estão incluídas na figura para melhor entendimento de como ocorreu o processo de desenvolvimento deste *plug-in*. Estas classes são implementadas na API do Eclipse.

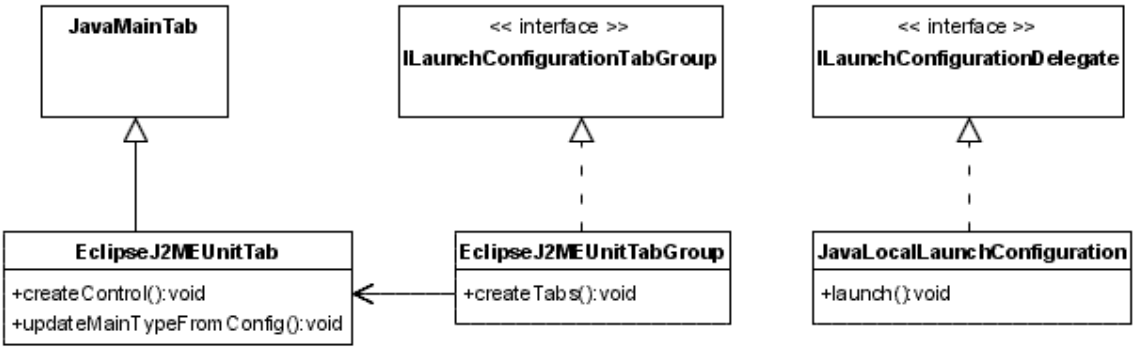

Figura 37. Diagrama de Classes do *plug-in* EclipseJ2MEUnit_ui.

A classe `JavaLocalLaunchConfiguration` tem a responsabilidade de capturar os atributos de configuração e executar o aplicativo, o método `launch` deve ser implementado para atingir o objetivo desta classe. Como já foi explicado antes, a classe `EclipseJ2MEUnitTabGroup` invoca a classe `EclipseJ2MEUnitTab` para criar a aba de configuração do *plug-in* EclipseJ2MEUnit.

Este *plug-in* foi criado para evitar que o nome da classe que irá executar os testes, neste caso a classe `eclipseJ2MEUnit.awtui.TestRunner`, tivesse que ser escrita pelo usuário. Além disso, este *plug-in* cria a própria aba de configuração de execução para que os testes baseados no *plug-in* EclipseJ2MEUnit possam ser executados mais facilmente.

O EclipseJ2MEUnit_ui não foi implementado diretamente no *plug-in* EclipseJ2MEUnit pois seguindo o padrão de outros *plug-ins*, como o JUnit, a configuração da execução fica separada das classes principais do *plug-in*.

4.3 Pontos de extensão

Um *plug-in* pode definir diversos pontos de extensão (*extension point*) em seu arquivo manifesto, permitindo que outros *plug-ins* estendam suas funções aprimorando-o ou adaptando-o para outras tarefas, seguindo o mesmo padrão do *plug-in* estendido. Esta Seção aborda os pontos de extensão construídos no projeto (`wirelessToolkitTest` e `newTestRunner`), permitindo que outros *plug-ins* sejam acoplados ao EclipseJ2MEUnit, estendendo suas funcionalidades e aprimorando o projeto.

Neste trabalho foram definidos dois pontos de extensão para o *plug-in* EclipseJ2MEUnit. O primeiro permite a extensão de uma interface que define a estrutura dos casos de teste e o comportamento deles, permitindo que testes para *wireless toolkits* sejam acoplados ao *plug-in*. Este ponto de extensão tem o `id wirelessToolkitTest`, como mostra a Figura 38.

O outro ponto de extensão, `newTestRunner`, foi criado para permitir que outras formas de se executar os testes do *plug-in* sejam analisadas. Neste trabalho os testes são executados e exibidos em forma de texto no console do Eclipse e também na forma de uma interface AWT, mas através deste ponto de extensão permitimos que sejam feitas adaptações para os testes serem exibidos na forma de uma interface do tipo *Swing*, por exemplo.

```
<extension-point   id="wirelessToolkitTest"   name="Test   WirelessToolkit"
schema="schema/wirelessToolkitTest.exsd"/>

<extension-point       id="newTestRunner"       name="New       TestRunner"
schema="schema/newTestRunner.exsd"/>
```

Figura 38. Trecho do arquivo manifesto com a declaração dos pontos de extensão.

A Figura 38 ilustra a declaração desses pontos de extensão. O atributo `id` define o identificador do *plug-in*, `name` recebe o nome dado ao ponto de extensão e `schema` especifica o arquivo esquema do ponto de extensão, neste exemplo este arquivo encontra-se dentro da pasta `schema`, como foi especificado.

O arquivo esquema do ponto de extensão guarda mais informações sobre o ponto de extensão e sua estrutura além das informações apresentadas no arquivo manifesto. Estas informações ficam armazenadas num arquivo XML esquema com o sufixo *.exsd*. Informações como: a classe que deve ser implementada para se estender o *plug-in*, que fica armazenada no atributo `implementation`; uma descrição do ponto de extensão; os atributos que devem ser declarados ao se utilizar o ponto de extensão; dentre outras informações.

Ao se utilizar o ponto de extensão `wirelessToolkitTest` deve-se declarar os atributos `name`, `id` e `implementation`, que encontram-se declarados no arquivo *wirelessToolkit.exsd*. No atributo `implementation` encontra-se a classe que irá implementar a interface `Test` , responsável por definir como serão executados os casos de teste através do método `run`. Se houver necessidade de tornar mais específico como os testes devem ser executados este ponto de extensão deve ser utilizado. A Figura 39 ilustra um exemplo de como a extensão a este ponto de extensão deve ser declarada dentro do arquivo manifesto.

```
<extension
       point= "eclipseJ2MEUnit.framework.wirelessToolkitTest">
      <wirelessToolkitTest
            name="Nokia Wireless Toolkit Test"
            implementation= "projeto.nokia.Testador"
            id=" projeto.unitTest.NokiaWTK">
      </wirelessToolkitTest>
</extension>
```

Figura 39. Exemplo de uso do ponto de extensão `wirelessToolkitTest`.

No ponto de extensão `newTestRunner` a classe que deve ser implementada é a `IBaseTestRunner`, interface que declara métodos como o `processArguments`, `getTest` e `setLoading`, que respectivamente são responsáveis por iniciar a execução dos testes, retornar o `Test` correspondente a uma dada *suite* de teste e configurar o comportamento da execução dos testes. Assim como no ponto de extensão `wirelessToolkitTest`, deve-se declarar os atributos `name`, `id` e `implementation`, definidos no arquivo XML esquema *newTestRunner.exsd*.

A Figura 40 mostra um exemplo de como estender esse ponto de extensão dentro do arquivo manifesto. A classe `TestRunner` irá implementar a interface `IBaseTestRunner` e definir o comportamento de seus métodos.

```
<extension
 point= "eclipseJ2MEUnit.runner.newTestRunner">
      <newTestRunner
            name="new TestRunner"
            implementation= "projeto.TestRunner"
            id=" projeto.testRunner">
      </newTestRunner>
</extension>
```

Figura 40. Exemplo do uso do ponto de extensão `newTestRunner`.

A *tag* `extension` declara as extensões que serão usadas no *plug-in*, para cada extensão deve-se declarar uma *tag* deste tipo. O atributo `point` especifica o ponto de extensão pelo qual o *plug-in* será acoplado a um outro *plug-in*. A *tag* `newTestRunner` foi definida dentro do arquivo *newTestRunner.exsd* e deve ser declarada ao se utilizar o ponto de extensão `newTestRunner`, os atributos `name`, `id` e `implementation` já foram explicados anteriormente e pertencem a *tag* `newTestRunner`.

Esses pontos de extensão são declarados dentro do arquivo manifesto do *plug-in* EclipseJ2MEUnit, Apêndice A.

Após a construção dos desses *plug-ins*, EclipseJ2MEUnit e EclipseJ2MEUnit_ui, será possível que desenvolvedores de aplicativos J2ME testem suas classes e métodos dentro do ambiente Eclipse, ambiente este que já permite o desenvolvimento de aplicativos J2ME. Além de que, a interface na qual os resultados dos testes são exibidos facilita o trabalho do desenvolvedor para encontrar os pontos onde houve erros ou falhas. Os pontos de extensão criados irão permitir que outros trabalhos sejam desenvolvidos a partir deste.

Capítulo 5

Estudo de Caso

Este Capítulo contém classes com aplicativos J2ME e as respectivas classes de teste desses aplicativos. Estas classes foram desenvolvidas para melhor entendimento do funcionamento do *plug-in* EclipseJ2MEUnit e também foram utilizados como uma forma de validar o *plug-in*.

5.1 Exemplos

Nesta Seção apresentamos exemplos de classes que utilizaram o EclipseJ2MEUnit para implementar as classes de testes. Foram implementadas duas classes J2ME, a classe `Calculadora` e a classe `Language` e as classes de testes dessas aplicações.

A classe `Calculadora` ilustrada na 0 é uma aplicação J2ME que faz as operações básicas de matemática, soma, subtração, divisão e multiplicação, com dois operadores definidos pelo usuário.

Esta classe é composta por várias telas, a primeira tela, `initialScreenHandler`, contém um campo para digitar o primeiro operando da operação e é invocada ao iniciar o aplicativo pelo método `startApp`. A segunda tela, `secondScreenHandler`, contém um campo para o usuário inserir o segundo operando. Logo após o usuário definir os operandos, a tela `resultScreenHandler` é exibida para fazer-se a escolha de qual operação será executada, esta troca de telas é gerenciada pelo método `commandAction`, que recebe um comando para poder trocar a tela, este comando pode ser o clique em algum dos botões do menu. A 0 ilustra alguns métodos da classe `Calculadora`.

A classe `BaseCalculadora` definida como um parâmetro é chamada pela classe `Calculadora`, pois é responsável por realizar as operações básicas do aplicativo, esta classe é mostrada na Figura 42.

```
public class Calculadora extends MIDlet implements CommandListener {
  public BaseCalculadora bc;
  ...
      private Screen resultScreenHandler() {
        int resultado;
        resultado=bc.calculador(bc.getOperador1(),
        bc.getOperador2(), list.getSelectedIndex());
        tb = new TextBox("Resultado", "= " + resultado, 10, 0);
        tb.addCommand(exitCommand);
        tb.addCommand(selectCommand);
        tb.setCommandListener(this);
        return tb;
      }

      protected void startApp() {
        initialScreen = initialScreenHandler();
        display.setCurrent(initialScreen);
      }

      protected void pauseApp() {
      }

      protected void destroyApp(boolean cond) {
      }

      public void commandAction(Command com, Displayable disp) {
            if (disp == initialScreen) {
               if (com == exitCommand) {
                   destroyApp(false);
                  notifyDestroyed();
               } else {
                  secondScreen = secondScreenHandler();
                  display.setCurrent(secondScreen);
               }
            ...
      }
}
```

Figura 41. Parte da classe `Calculadora`.

A classe `BaseCalculadora` realiza operações básicas da classe `Calculadora` como a operação `calculador` que recebe dois operandos e um inteiro que simboliza a operação a ser realizada, e faz o cálculo da operação desejada, e as operações `getOperador1` e `setOperador1`, que retorna e modifica o valor `operador1`, respectivamente. A Figura 42 contém trechos dessa classe.

Esta é a classe que testamos utilizando a classe de teste `BaseCalculadoraTest`. Como ainda não há definido o teste para interfaces de aplicações J2ME, apenas as operações responsáveis pela execução do aplicativo são testadas.

```
public class BaseCalculadora {

   public final String[] itensMenu = { "Adição", "Subtração",
"Multiplicação","Divisão" };

   public int calculador(int op1, int op2, int operacao){
        int resultado;
        if (operacao==0){
            resultado = op1 + op2;  }
        else if (operacao==1){
            resultado = op1 - op2;  }
        else if (operacao ==2 ){
            resultado = op1 * op2;  }
        else {
            resultado = op1 / op2;  }

        return resultado;
        }

        public int getOperador1() {
            return operador1;
        }

        public void setOperador1(int operador1) {
            this.operador1 = operador1;
        }
        ...
}
```

Figura 42. Parte da classe `BaseCalculadora`.

A Figura 43 ilustra trechos de código da classe `BaseCalculadoraTest` responsável por realizar o teste da classe `BaseCalculadora`. Como se pode observar, esta classe estende a classe `TestCase` e implementa os métodos para execução dos testes. O atributo do tipo `BaseCalculadora` é inicializado dentro do método `setUp`, responsável por configurar o ambiente para execução do teste. Dentro deste método também é modificado os atributos `operador1` e `operador2` para efetuarmos a execução dos testes.

Os métodos `testCalculadorSoma`, `testCalculadorSubtracao` e `testOperador1`, são os métodos que contém as assertivas para testar a classe `BaseCalculadora`. Estes métodos testam a operação de soma do método `calculador`, a operação de subtração e as operações que retornam e modificam o `operador1`, respectivamente. O método `suite` cria a *suite* de teste com os métodos que realizam os testes do aplicativo.

```
public class BaseCalculadoraTest extends TestCase {
  public BaseCalculadora bc;

  public Test suite()
  {
      TestSuite aSuite = new TestSuite();
      aSuite.addTest(new BaseCalculadoraTest
      ("testCalculadorSoma", new TestMethod() { public void run(TestCase
      tc) {((BaseCalculadoraTest)     tc).testCalculadorSoma(); }}));
  ...
  return aSuite;
  }

  public void setUp() {
      bc = new BaseCalculadora();
      bc.setOperador1(2);
      bc.setOperador2(3);
  }

  public void testCalculadorSoma() {
      assertEquals("Teste Soma", 4, bc.calculador(1, 2, 0));
  }

  public void testCalculadorMultiplicacao() {
      assertEquals("Teste Multiplicação", 3, bc.calculador(1, 2, 2));  }

  public void testOperador2() {
      assertEquals("Operador 2", 2, bc.getOperador2());  }
  ...
}
```

Figura 43. Trechos de código da classe de teste `BaseCalculadoraTest`.

Após a execução da classe de teste `BaseCalculadoraTest` uma interface AWT é exibida mostrando o resultado do teste e se houver alguma falha ou erro, o caminho onde esse problema foi encontrado também é mostrado.

Como observamos na Figura 44, foram encontradas falhas nos métodos de teste: `testCalculadorSoma`, `testCalculadorMultiplicacao`, `testCalculadorDivisao` e `testOperador2`, os valores esperados por estes métodos também são apresentados. Por exemplo, no método `testCalculadorSoma` o valor esperado é quatro, mas foi encontrado três, isso porque a soma foi feita entre os operandos um e dois, como ilustra a Figura 43.

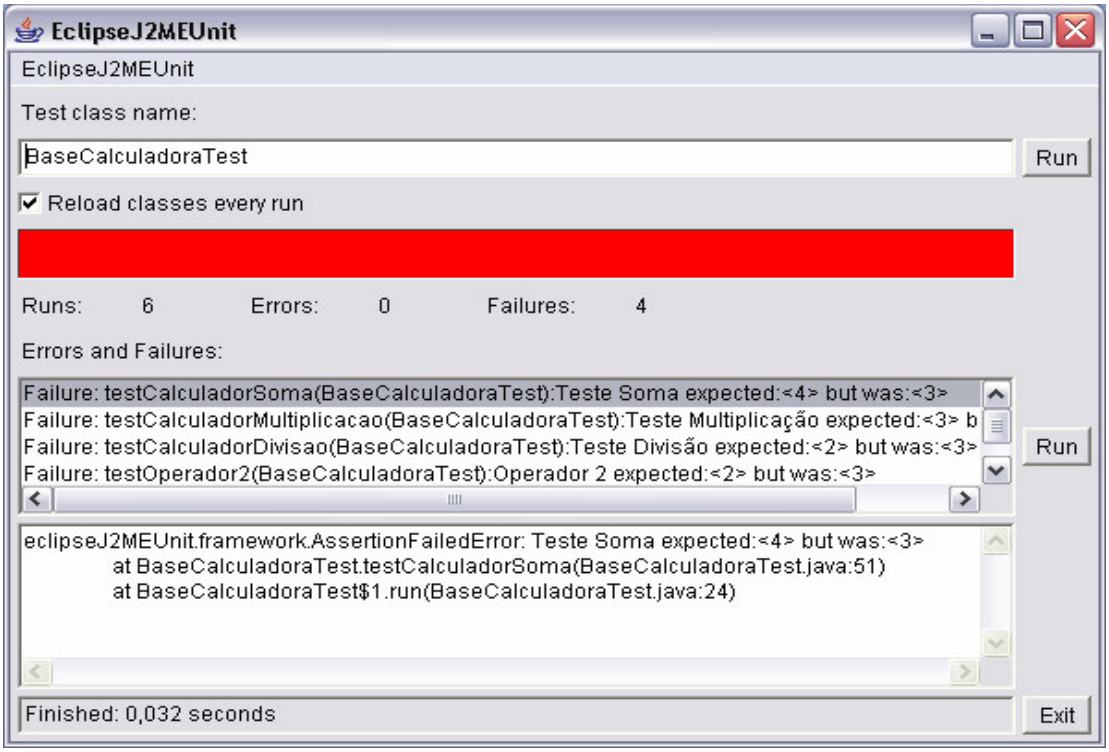

Figura 44. Tela com resultados da classe de teste `BaseCalculadoraTest`.

A classe `Language`, Figura 45, foi criada para podermos introduzir uma classe que executa todos os testes da classe `Calculadora` e `Language` ao mesmo tempo. Esta classe contém uma tela com três opções de linguagem, o usuário seleciona alguma delas e o texto "Olá Mundo!" é exibido na linguagem desejada.

O método `commandAction` recebe um `Command` que indica quando o usuário selecionou alguma das teclas do menu do celular e chama o método `linguagem` da classe base para executar a operação de selecionar a mensagem de retorno para ser exibida na tela.

```
public class Language extends MIDlet implements CommandListener {
...
   public void commandAction(Command c, Displayable d) {
        alerta = new Alert("Atenção");
        if (c == selecioneCommand) {
                String mensagem =
                bl.linguagem(list.getSelectedIndex());
                alerta.setString(mensagem);
                display.setCurrent(alerta);
        } else if (c == sairCommand) {
                destroyApp(false);
                  notifyDestroyed();
        }
}
```

Figura 45. Trechos de código da classe `Language`.

A classe `BaseLanguage`, assim como a classe `BaseCalculadoraTest` contém as operações básicas executadas pelo aplicativo, que não dependem da interface gráfica, será esta a classe testada pelo *plug-in* EclipseJ2MEUnit e é apresentada na Figura 46 abaixo. O método

`linguagem` recebe como parâmetro um inteiro que indica qual linguagem o usuário escolheu. Este método é chamado na classe `Language` ao ser executado o método `commandAction`.

```
public class BaseLanguage {
   public String opcoes[] = {"Ingles","Portugues","Espanhol"};
   ...
   public String linguagem(int comando) {
      String mensagem="";
      if (comando == 0) {
            mensagem = "Hello World!";
      } else if (comando == 1) {
            mensagem = "Olá Mundo!";
      } else {
            mensagem = "Hola Mundo!";}
      return mensagem;
      }
}
```

Figura 46. Parte da classe `BaseLanguage`.

A classe de teste `BaseLanguageTest` testa a classe `BaseLanguage`, vários métodos de teste são criados para testar o método `linguagem`. Um exemplo desses métodos de teste é o `testLinguagemPort` que contém uma assertiva para testar se a mensagem que será exibida na tela será mesmo a mensagem correta. A Figura 47 apresenta esta classe.

```
public class BaseLanguageTest extends TestCase {
   public BaseLanguage bl;
   public Test suite()
   {
      TestSuite aSuite = new TestSuite();
      aSuite.addTest(new BaseLanguageTest ("testLinguagemPort", new
TestMethod()      {  public void run(TestCase tc) {((BaseLanguageTest)
tc).testLinguagemPort(); } }));
   ...
   return aSuite;
   }

   public void setUp() {
      bl = new BaseLanguage();      }

   public void testLinguagemPort(){
      assertEquals("Linguagem: Português", "Olá World!",  bl.linguagem(1));
}
...
}
```

Figura 47. Parte da classe `BaseLanguageTest`.

Os resultados da execução da classe `BaseLanguageTest` estão apresentados na Figura 48, o método `testLinguagemPort` esperava como saída a string “Olá World!” e não “Olá Mundo!” como foi a saída.

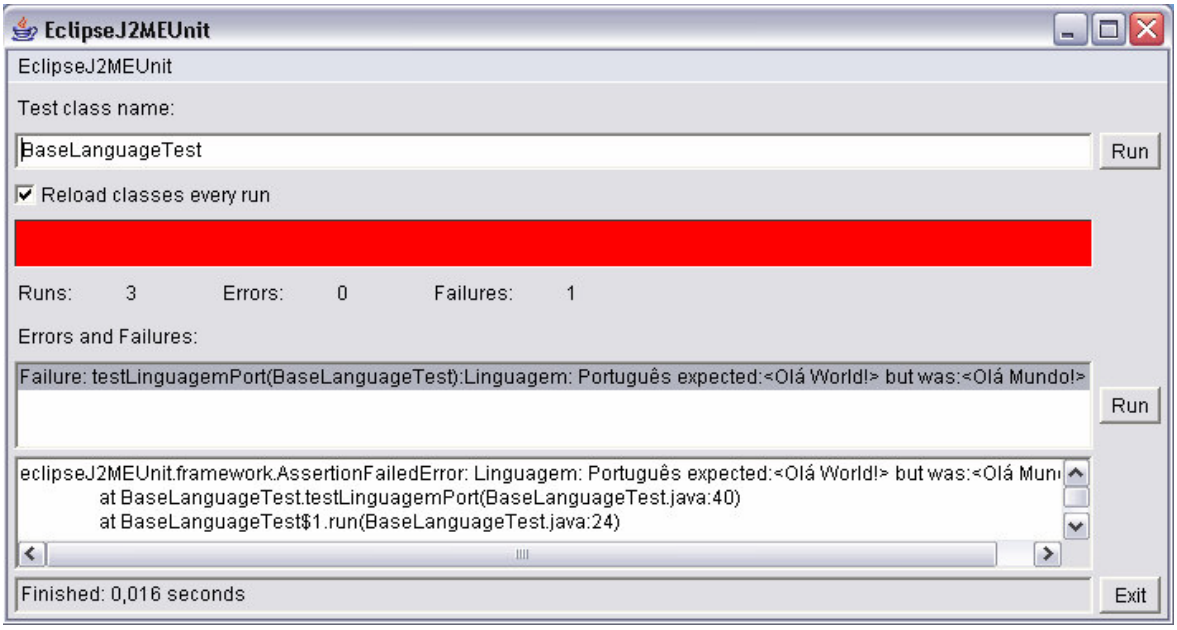

Figura 48. Tela com resultados da classe de teste `BaseLanguageTest`.

A classe `TodoTest` cria uma *suite* de teste com todas as *suites* das classes que irão executar testes, neste caso as classes `BaseLanguageTest` e `BaseCalculadoraTest`, a Figura 49 mostra trechos dessa classe.

Dentro do método `suite` são invocados as *suite*s de teste das outras classes, e todos os testes podem ser executados ao mesmo tempo e os resultados exibidos numa única tela de resultados.

```
public class TodoTest extends TestCase {
      ...
public Test suite()
       {
              TestSuite suite = new TestSuite();
              suite.addTest(new BaseLanguageTest().suite());
              suite.addTest(new BaseCalculadoraTest().suite());
              return suite;
       }
}
```

Figura 49. Trechos de código da classe `TodoTest`.

Os resultados da execução da classe `TodoTest` estão exibidos na Figura 50, e podemos observar que todos os resultados das classes de testes estão apresentados nesta tela. A execução desta classe agiliza o processo de teste, pois os teste são executados juntamente.

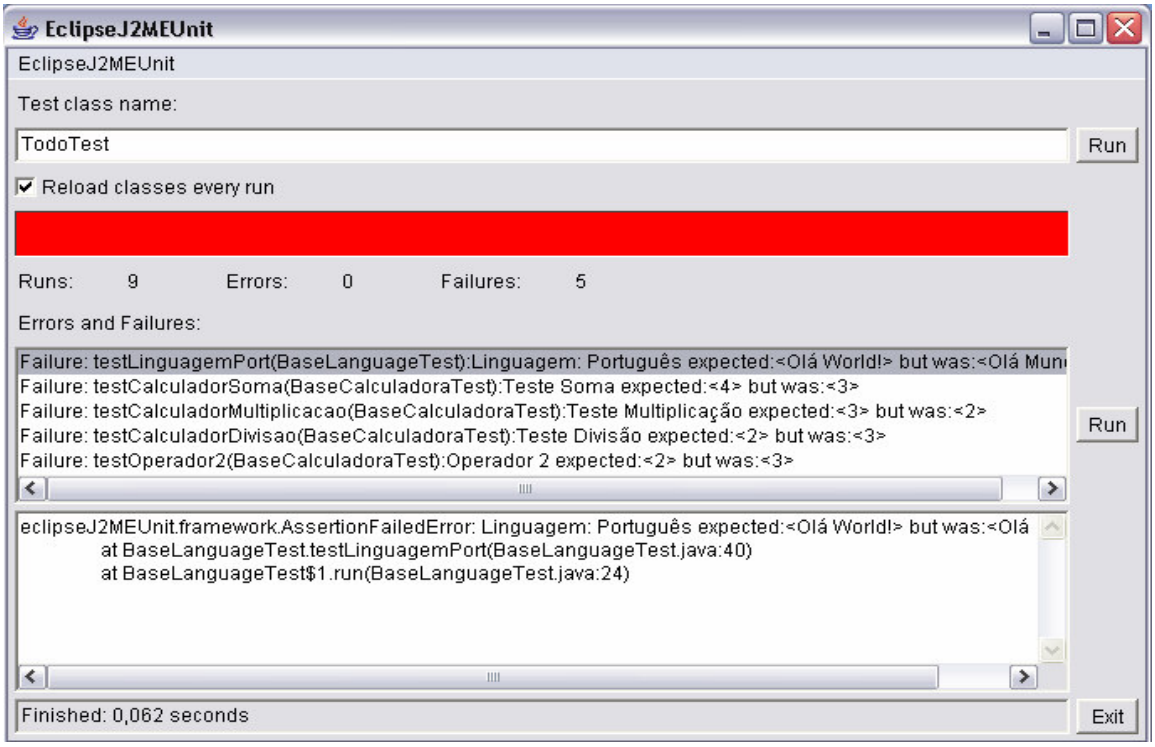

Figura 50. Tela com resultados da classe de teste `TodoTest`.

A implementação dessas classes de teste permite o fácil entendimento da utilização do *plug-in* EclipseJ2MEUnit. O código completo dessas classes pode ser encontrado no Apêndice B.

Estas classes de teste assim como as telas geradas após a execução dos testes permitem validar os *plug-ins* desenvolvidos neste projeto. É possível observar que quando há algum erro ou falha na classe de teste ou da aplicação, estes problemas são identificados pelo *plug-in* EclipseJ2MEUnit e exibidos na interface gráfica mostrada nas figuras acima.

Após a utilização dos *plug-ins* desenvolvidos neste projeto para executar os testes dessas classes de aplicação citadas nos exemplos, podemos afirmar que os *plug-ins* se comportam corretamente, satisfazendo o escopo deste projeto.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste Capítulo apresentaremos as contribuições e problemas encontrados durante a realização do projeto. Também será apresentado direções para trabalhos futuros.

A Seção 6.1 contém uma breve descrição dos principais conceitos necessários durante o desenvolvimento dos *plug-ins* deste trabalho, o EclipseJ2MEUnit e o EclipseJ2MEUnit_ui, as atividades que foram desenvolvidas neste projeto e os resultados obtidos ao final do trabalho.

Na Seção 6.2 apresentamos alguns exemplos de trabalhos futuros que podem ser desenvolvidos a partir deste projeto. Com a definição dos pontos de extensão no EclipseJ2MEUnit, esta tarefa torna-se mais clara e fácil de ser realizada.

6.1 Contribuições

O processo de teste de *software* não é mais tratado de forma desorganizada, depois de analisados os problemas ocorridos com *softwares* desenvolvidos sem as atividades de teste, como o alto custo de manutenção ou as funcionalidades definidas que não eram implementadas, o processo de teste é hoje uma das fases mais importantes do processo de desenvolvimento. O processo de teste de *software* tem o objetivo de assegurar a qualidade do *software* no que diz respeito, por exemplo, à corretude do mesmo, e verificação das funcionalidades requisitadas.

A atividade de teste pode ser aplicada do início do projeto até sua finalização, como também somente após a codificação do sistema. O processo de *software* define como será executado o processo de teste, a seqüência de atividades necessárias para o desenvolvimento, os documentos que devem ser entregues, dentre outras atividades.

Existem vários aplicativos desenvolvidos por programadores Java, dentre esses aplicativos encontramos aqueles que se baseiam na plataforma de desenvolvimento J2ME. São eles celulares, PDAs, *pagers*, dentre outros aplicativos. Devido à grande procura dos clientes, celulares são desenvolvidos com cada vez mais aplicativos para facilitar atividades dos usuários, e com tecnologias diversas.

Aplicativos para dispositivos móveis precisam ser testados antes de serem entregues para o usuário final. Devido à grande quantidade de diferentes aplicativos para celulares que são desenvolvidos, o processo de teste para aplicativos J2ME tem grande importância.

O JUnit é um *framework* que facilita o teste de aplicativos Java e pode ser utilizado como um *plug-in* do Eclipse. O Eclipse é um ambiente de programação bastante utilizado por desenvolvedores já que esta IDE permite fácil codificação de sistemas, permite que vários *plug-*

ins sejam acoplados ao Eclipse e que sejam utilizados ao mesmo tempo pelo desenvolvedor. Assim como o JUnit, o J2MEUnit foi desenvolvido para facilitar o teste de *software*, porém, neste caso, voltado para testar aplicativos J2ME. Ao contrário do JUnit, o J2MEUnit não foi projetado como um *framework* que pode ser utilizado como um *plug-in* no Eclipse.

Neste trabalho desenvolvemos um *framework* que auxilia o desenvolvedor a testar aplicativos J2ME. Este *framework* foi desenvolvido com base no J2MEUnit; algumas de suas classes foram alteradas para permitir que o resultado do teste fosse exibido tanto no console do Eclipse, através da classe `eclipseJ2MEUnit.textui.TestRunner`, quanto numa interface AWT através da classe `eclipseJ2MEUnit.awtui.TestRunner`, a qual facilita a visualização dos erros e mostra a linha da classe na qual houve falhas ou erros.

Após a construção do *plug-in* EclipseJ2MEUnit, desenvolvemos o *plug-in* EclipseJ2MEUnit_ui para auxiliar na execução dos testes desenvolvidos com o EclipseJ2MEUnit.

Com o desenvolvimento dos *plug-ins* EclipseJ2MEUnit e EclipseJ2MEUnit-ui, oferecemos suporte ao processo de teste de aplicativos J2ME dentro do Eclipse, oferecendo a visualização dos erros através da exibição de uma interface.

Os *plug-ins* que desenvolvemos podem ser úteis para desenvolvedores J2ME que utilizam o *plug-in* EclipseME, resultando em um ambiente com um *plug-in* de desenvolvimento e outro de teste, facilitando o processo de desenvolvimento, pois o programador não precisa ficar alternando entre ferramentas para desenvolver e testar os aplicativos.

6.2 Trabalhos Futuros

Ao desenvolvermos novos *plug-ins* é possível definirmos pontos de extensão deste *plug-in*, que consistem em pontos do *plug-in* através do qual outros *plug-ins* podem ser acoplados a ele implementando a classe definida pelo ponto de extensão.

Ao desenvolvermos o EclipseJ2MEUnit foram definidos dois pontos de extensão, o `wirelessToolkitTest` e o `newTestRunner`. O primeiro ponto de extensão foi criado para permitir que outros *plug-ins* implementem a interface `Test`, responsável por definir a estrutura dos casos de teste e o comportamento deles. Um possível trabalho a ser feito é utilizar este ponto de extensão para definir maneiras específicas de como os testes devem ser executados. Um exemplo seria definir a execução dos testes para *wireless toolkits* que o EclipseME disponibiliza, como o da Motorola, da Sun, da Nokia, dentre outros, para, a partir daí, ser possível testar a interface gráfica destes dispositivos.

O outro ponto de extensão, `newTestRunner`, foi definido para capacitar que outros *plug-ins* possam estender o *plug-in* EclipseJ2MEUnit para que outras formas de se executar os testes do *plug-in* sejam desenvolvidas. Neste projeto foram criadas três formas de se executar os testes, através do `TestRunner` do pacote *textui*, que imprimi os resultados no console do Eclipse, do pacote `awtui`, que imprime os resultados numa interface AWT como foi mostrado na Figura 31 da Seção 4.2, do pacote `midletui`, que implementa o `TestRunner` como um MIDlet para poder ser executado como um emulador WTK (*Wireless ToolKit*). Poderia ser criado um novo `TestRunner` através da implementação da interface `IBaseTestRunner` que permitisse a exibição dos resultados do teste numa interface *Swing*, por exemplo.

Outra forma de se utilizar deste *plug-in* seria fazer com que o mesmo, fosse acoplado ao EclipseME, *plug-in* destinado a facilitar o desenvolvimento de aplicativos J2ME. Para isto, seria necessário criar um ponto de extensão para o EclipseME que permitisse o acoplamento dos *plug-ins* deste projeto ao EclipseME no Eclipse, resultando numa nova versão do EclipseME. Conseqüentemente, teríamos um único plug-in capaz de ampliar a realização das tarefas

necessárias durante o desenvolvimento de aplicativos J2ME. Além disso, o tempo destinado à instalação e configuração de ferramentas seria reduzido.

Bibliografia

[1] PRESSMAN, R. S. “Engenharia de Software”. 5 ed., McGraw-Hill, 2002
[2] SOMMERVILLE, R. “Engenharia de *Software*”. 6 ed. Addison Wesley, 2003
[3] MAHMOUND, Q. “Testing *Wireless Java Applications*”, novembro, 2002
[4] JUnit: Http://www.junit.org, visitado em 10 de abril de 2005.
[5] Eclipse: http://www.eclipse.org, visitado em 30 de abril de 2005.
[6] J2ME: Http://java.sun.com/j2me, visitado em 01 de março de 2005.
[7] J2MEUnit: Http://j2meunit.sourceforge.net, visitado em 10 de maio de 2005.
[8] Extremme Programming: www.extremeprogramming.org, visitado em 10 de abril de 2005.
[9] KANER, C.; FALK, J. e NGUYEN, H.. “*Testing Computer Software*”. 2 ed. Wiley, 1999.
[10] Standish Group. CHAOS: A Recipe for Success. Disponível em http://www.standishgroup.com/sample_research/PDFpages/chaos1999.pdf. Visitado em: 23 de junho de 2005.
[11] SOARES, M. "Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software". Universidade Presidente Antônio Carlos.
[12] COPELAND, L. “*A Practitioner’s Guide to Software test Design*”. Artech House, Incorporated, 2003.
[13] GAMMA, E.; BECK, K. “Contributing to Eclipse: Principles, Patterns, and Plug-ins”. Addison-Wesley, 2004.
[14] Eclipse Platform Technical Overview: http://www.eclipse.org/whitepapers/eclipse-overview.pdf, 2003.
[15] *Notes on the Eclipse Plug-in Architecture*: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, visitado em 05 de abril de 2005.
[16] BECK,KENT e JOHNSON,RALPH “*Patterns Generate Architectures*”, 8ª European Conference, ECOOP 94, Itália.
[17] VAZ, R. "JUnit - *Framework* para Testes em Java". Centro Universitário Luterano de Palmas, 2003.
[18] BECK, KENT “*Smalltalk Best Patterns*” , Pretice Hall, 1996.
[19] DOSHI, G. "J2MEUnit Tutorial". Instrumentalservices.com: http://www.instrumentalservices.com, 2004.
[20] MUCHOW, Jonh W. Core J2ME. “Tecnologia e MIDP”. Jonh W. Muchow. São Paulo: Pearson Makron Books, 2004.
[21] CAVALCANTE, W. “Tecnologia Java para Sistemas Embarcados". UFPE, Cin, 2001.
[22] CLDC: http://java.sun.com/products/cldc/wp, visitado em 07 de fevereiro de 2005.
[23] Java: http://java.sun.com, visitado em 07 de fevereiro de 2005.
[24] EclipseME: http://eclipseme.org, visitado em 15 de maio de 2005.
[25] The Launching Framework in Eclipse: http://www.eclipse.org/articles/Article-Launch-Framework/launch.html, visitado em 15 de maio de 2005.
[26] Help Eclipse: http://help.eclipse.org, visitado em 30 de abril de 2005.

Apêndice A

Arquivos manifesto dos *plug-ins*

Apresentaremos aqui os arquivos manifestos desenvolvidos nos plug-ins EclipseJ2MEUnit e EclipseJ2MEUnit_ui.

A.1 EclipseJ2MEUnit

Apresentaremos neste apêndice o arquivo manifesto do *plug-in* EclipseJ2MEUnit.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="EclipseJ2MEUnit"
   name="EclipseJ2MEUnit Plug-in"
   version="1.0.0"
   provider-name=""
   class="eclipseJ2MEUnit.internal.TesPlugin">

   <runtime>
      <library name="EclipseJ2MEUnit.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="eclipseme.core"/>
      <import plugin="org.eclipse.ui"/>
   </requires>

   <extension-point id="wirelessToolkitTest" name="Test WirelessToolkit"
   schema="schema/wirelessToolkitTest.exsd"/>
   <extension-point id="newTestRunner" name="New TestRunner"
   schema="schema/newTestRunner.exsd"/>
</plugin>
```

A.2 EclipseJ2MEUnit_ui

Apresentaremos neste apêndice o arquivo manifesto do *plug-in* EclipseJ2MEUnit_ui.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id=" EclipseJ2MEUnit_ui"
   name="EclipseJ2MEUnit_ui Plug-in"
   version="1.0.0"
   provider-name=""
   class=" EclipseJ2MEUnit_ui.Plugin">

   <runtime>
      <library name=" EclipseJ2MEUnit_ui.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
    <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.debug.ui"/>
      <import plugin="org.eclipse.debug.core"/>
      <import plugin="org.eclipse.jdt.core"/>
      <import plugin="org.eclipse.jdt"/>
      <import plugin="org.eclipse.jdt.debug.ui"/>
      <import plugin="org.eclipse.jdt.launching"/>
      <import plugin="org.eclipse.core.resources"/>
   </requires>

   <extension
            point="org.eclipse.debug.core.launchConfigurationTypes">
      <launchConfigurationType
            name="EclipseJ2MEUnit Application"
            delegate=
            "eclipseJ2MEUnit.launch.EclipseJ2MEUnitLaunchConfigurations"
            modes="run"
            id="EclipseJ2MEUnitconfigurationType">
      </launchConfigurationType>
      </extension>
      <extension
      point="org.eclipse.debug.ui.launchConfigurationTabGroups">
      <launchConfigurationTabGroup
            type="EclipseJ2MEUnitconfigurationType"
            class="eclipseJ2MEUnit.launch.EclipsemeUnitTabGroup"
            id="BetaTabGroup">
      </launchConfigurationTabGroup>
      </extension>

</plugin>
```

Apêndice B

Exemplos apresentados no Capítulo 5

O código completo dos exemplos apresentados no capítulo 5 estão descritos neste apêndice.

B.1 Calculadora

A classe Calculadora está apresentada abaixo.

```
package calculadora;
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;

public class Calculadora extends MIDlet implements CommandListener {
      public Display display;
      public static Command selectCommand, exitCommand;
      public static TextField op1, op2;
      public TextBox tb;
      public Form form;
      public Screen initialScreen, secondScreen, listScreen, resultScreen;
      public ChoiceGroup list;
      public BaseCalculadora bc;

      public Calculadora() {
            bc = new BaseCalculadora();
            display = Display.getDisplay(this);
            selectCommand = new Command(bc.softKeyR, Command.ITEM, 1);
            exitCommand = new Command(bc.softKeyL, Command.EXIT, 1);
      }

      public Screen initialScreenHandler() {
            op1 = new TextField("Operando 1", "", 10, 1);
            form = new Form(bc.titulo);
            form.append(op1);
            form.addCommand(exitCommand);
            form.addCommand(selectCommand);
            form.setCommandListener(this);
            return form;
      }

      private Screen secondScreenHandler() {
```

```
        if (!op1.getString().equals("")) {
              bc.setOperador1(Integer.parseInt(op1.getString()));   }

        form = new Form(bc.titulo);
        op2 = new TextField("Operando 2", "", 10, 1);
        form.append(op2);
        form.addCommand(exitCommand);
        form.addCommand(selectCommand);
        form.setCommandListener(this);
        return form;
}

private Screen listScreenHandler() {
        if (!op2.getString().equals("")) {
              bc.setOperador2(Integer.parseInt(op2.getString()));   }

        form = new Form(bc.titulo);
        list = new ChoiceGroup("Menu", ChoiceGroup.EXCLUSIVE,
        bc.itensMenu,null);
        form.append(list);
        form.addCommand(exitCommand);
        form.addCommand(selectCommand);
        form.setCommandListener(this);
        return form;
}

private Screen resultScreenHandler() {
        int resultado;
        resultado = bc.calculador(bc.getOperador1(), bc.getOperador2(),
        list.getSelectedIndex());
        tb = new TextBox("Resultado", "= " + resultado, 10, 0);
        tb.addCommand(exitCommand);
        tb.addCommand(selectCommand);
        tb.setCommandListener(this);
        return tb;
}

protected void startApp() {
        initialScreen = initialScreenHandler();
        display.setCurrent(initialScreen);  }

protected void pauseApp() {   }

protected void destroyApp(boolean cond) {   }

public void commandAction(Command com, Displayable disp) {
        if (disp == initialScreen) {
              if (com == exitCommand) {
                    destroyApp(false);
                    notifyDestroyed();
              } else {
                    secondScreen = secondScreenHandler();
                    display.setCurrent(secondScreen);         }
        } else if (disp == secondScreen) {
              if (com == exitCommand) {
                    destroyApp(false);
                    notifyDestroyed();
              } else {
                    listScreen = listScreenHandler();
                    display.setCurrent(listScreen);           }
```

```
            } else if (disp == listScreen) {
                  if (com == exitCommand) {
                        destroyApp(false);
                        notifyDestroyed();
                  } else {
                        resultScreen = resultScreenHandler();
                        display.setCurrent(resultScreen);         }
            } else {
                  if (com == exitCommand) {
                        destroyApp(false);
                        notifyDestroyed();
                  } else {
                        initialScreen = initialScreenHandler();
                        display.setCurrent(initialScreen);        }
            }
      }
}
```

B.2 BaseCalculadora

A classe BaseCalculadora está apresentada logo a seguir.

```
public class BaseCalculadora {
      public final String[] itensMenu = { "Adição", "Subtração",
      "Multiplicação","Divisão" };
      public final String titulo = "Calculadora";
      public final String softKeyL = "Exit";
      public final String softKeyR = "Select";
      public int operador1;
      public int operador2;

      public BaseCalculadora(){
      }

      public int calculador(int op1, int op2, int operacao){
            int resultado;
            if (operacao==0){
                  resultado = op1 + op2;        }
            else if (operacao==1){
                  resultado = op1 - op2;        }
            else if (operacao ==2 ){
                  resultado = op1 * op2;        }
            else {
                  resultado = op1 / op2;        }
            return resultado;
      }

      public int getOperador1() {
            return operador1; }

      public void setOperador1(int operador1) {
            this.operador1 = operador1;   }

      public int getOperador2() {
            return operador2; }

      public void setOperador2(int operador2) {
```

```
            this.operador2 = operador2;   }
}
```

B.3 BaseCalculadoraTest

A classe BaseCalculadoraTest, classe de teste, será apresentada logo a seguir

```
import eclipseJ2MEUnit.framework.Test;
import eclipseJ2MEUnit.framework.TestCase;
import eclipseJ2MEUnit.framework.TestMethod;
import eclipseJ2MEUnit.framework.TestSuite;

public class BaseCalculadoraTest extends TestCase {
      public BaseCalculadora bc;

      public BaseCalculadoraTest() {
            super("null");
      }

      public BaseCalculadoraTest(String sTestName, TestMethod rTestMethod)
      {
            super(sTestName, rTestMethod);      }

      public Test suite()  {

      TestSuite aSuite = new TestSuite();
      aSuite.addTest(new BaseCalculadoraTest("testCalculadorSoma", new
      TestMethod(){ public void run(TestCase tc) {((BaseCalculadoraTest)
      tc).testCalculadorSoma(); } }));

      aSuite.addTest(new BaseCalculadoraTest("testCalculadorSubtracao", new
      TestMethod() { public void run(TestCase tc) {((BaseCalculadoraTest)
      tc).testCalculadorSubtracao(); } }));

      aSuite.addTest(new BaseCalculadoraTest("testCalculadorMultiplicacao",
      new TestMethod() { public void run(TestCase tc) {((BaseCalculadoraTest)
      tc).testCalculadorMultiplicacao(); } }));

      aSuite.addTest(new BaseCalculadoraTest("testCalculadorDivisao", new
      TestMethod() { public void run(TestCase tc) {((BaseCalculadoraTest)
      tc).testCalculadorDivisao(); } }));

      aSuite.addTest(new BaseCalculadoraTest("testOperador1", new TestMethod()
      { public void run(TestCase tc) {((BaseCalculadoraTest)
      tc).testOperador1(); } }));

      aSuite.addTest(new BaseCalculadoraTest("testOperador2", new TestMethod()
      { public void run(TestCase tc) {((BaseCalculadoraTest)
      tc).testOperador2(); } }));

            return aSuite;
      }

      public void setUp() {
            bc = new BaseCalculadora();
            bc.setOperador1(2);
            bc.setOperador2(3);
```

```
	}

	public void testCalculadorSoma() {
		assertEquals("Teste Soma", 4, bc.calculador(1, 2, 0));
	}

	public void testCalculadorSubtracao() {
		assertEquals("Teste Subtração", 2, bc.calculador(5, 3, 1));
	}

	public void testCalculadorMultiplicacao() {
		assertEquals("Teste Multiplicação", 3, bc.calculador(1, 2, 2));
	}

	public void testCalculadorDivisao() {
		assertEquals("Teste Divisão", 2, bc.calculador(6, 2, 3));
	}

	public void testOperador1() {
		assertEquals("Operador 1", 2, bc.getOperador1());
	}

	public void testOperador2() {
		assertEquals("Operador 2", 2, bc.getOperador2());
	}
}
```

B.4 Language

A classe Language está apresentada a seguir.

```
package calculadora;
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import calculadora.BaseLanguage;

public class Language extends MIDlet implements CommandListener {
	private Display display;
	public Alert alerta;
	private Command selecioneCommand, sairCommand;
	private List list;
	private BaseLanguage bl;

	public Language() {
		bl = new BaseLanguage();
		display = Display.getDisplay(this);
		selecioneCommand = new Command(bl.softKeyR, Command.ITEM, 1);
		sairCommand = new Command(bl.softKeyL, Command.EXIT, 1);
		list = new List("Menu", List.IMPLICIT, bl.opcoes, null);
		list.addCommand(selecioneCommand);
		list.addCommand(sairCommand);
```

```
                list.setCommandListener(this);
        }

        protected void startApp() {
                display.setCurrent(list);  }

        protected void pauseApp() {   }

        protected void destroyApp(boolean cond) {  }

        public void commandAction(Command c, Displayable d) {
                alerta = new Alert("Atenção");
                if (c == selecioneCommand) {
                        String mensagem =
                        bl.linguagem(list.getSelectedIndex());
                        alerta.setString(mensagem);
                        display.setCurrent(alerta);
                } else if (c == sairCommand) {
                        destroyApp(false);
                        notifyDestroyed();
                }
        }
}
```

B.5 BaseLanguage

A classe BaseLanguage será apresentada a seguir.

```
public class BaseLanguage {
      public String opcoes[] = { "Ingles", "Portugues", "Espanhol" };
      public final String softKeyL = "Exit";
      public final String softKeyR = "Select";

      public BaseLanguage(){
      }

      public String linguagem(int comando) {
            String mensagem="";
            if (comando == 0) {
                  mensagem = "Hello World!";
            } else if (comando == 1) {
                  mensagem = "Olá Mundo!";
            } else {
                  mensagem = "Hola Mundo!";
            }
            return mensagem;
      }
}
```

B.6 BaseLanguageTest

A classe BaseLanguageTest, classe de teste, será apresentada logo a seguir

```
import eclipseJ2MEUnit.framework.Test;
import eclipseJ2MEUnit.framework.TestCase;
import eclipseJ2MEUnit.framework.TestMethod;
import eclipseJ2MEUnit.framework.TestSuite;

public class BaseLanguageTest extends TestCase {
	public BaseLanguage bl;

	public BaseLanguageTest() {
		super("null");
	}

	public BaseLanguageTest(String sTestName, TestMethod rTestMethod)
	{
		super(sTestName, rTestMethod);
	}

	public Test suite() {
	TestSuite aSuite = new TestSuite();

	aSuite.addTest(new BaseLanguageTest("testLinguagemPort", new
	TestMethod() { public void run(TestCase tc) {((BaseLanguageTest)
	tc).testLinguagemPort(); } }));

	aSuite.addTest(new BaseLanguageTest("testLinguagemEsp", new
	TestMethod() { public void run(TestCase tc) {((BaseLanguageTest)
	tc).testLinguagemEsp(); } }));

	aSuite.addTest(new BaseLanguageTest("testLinguagemIng", new TestMethod()
	{ public void run(TestCase tc) {((BaseLanguageTest)
	tc).testLinguagemIng(); } }));

		return aSuite;
	}

	public void setUp() {
		bl = new BaseLanguage();
	}

	public void testLinguagemPort(){
		assertEquals("Linguagem: Português", "Olá World!",
		bl.linguagem(1));
	}

	public void testLinguagemEsp(){
		assertEquals("Linguagem: Espanhol", "Hola Mundo!",
		bl.linguagem(2));
	}

	public void testLinguagemIng(){
		assertEquals("Linguagem: Inglês", "Hello World!",
		bl.linguagem(0));
	}
}
```

B.7 TodoTest

A classe TodoTest contém todas as *suite*s de testes da classes de teste anteriores.

```
import eclipseJ2MEUnit.framework.Test;
import eclipseJ2MEUnit.framework.TestCase;
import eclipseJ2MEUnit.framework.TestSuite;

public class TodoTest extends TestCase{

	public TodoTest()
	{
		super("null");
	}

	public TodoTest(String name)
	{
		super(name);
	}

	public static void main(String[] args)
	{
		String[] runnerArgs = new String[] { "TodoTest" };
		eclipseJ2MEUnit.textui.TestRunner.main(runnerArgs);
	}

	public Test suite()
	{
		TestSuite suite = new TestSuite();
		suite.addTest(new BaseLanguageTest().suite());
		suite.addTest(new BaseCalculadoraTest().suite());
		return suite;
	}
}
```