UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

CURSO CIÊNCIA DA COMPUTAÇÃO

TURMA 2005.1

TRABALHO DE GRADUAÇÃO

COMPONENTIZAÇÃO DE SOFTWARE EM JAVA™ 2 MICRO EDITION

UM *FRAMEWORK* PARA DESENVOLVIMENTO DE INTERFACE GRÁFICA PARA DISPOSITIVOS MÓVEIS

Autor: Leandro Marques do Nascimento

Orientador: Silvio Romero de Lemos Meira

Recife
Agosto de 2005

Dedico este trabalho a meus pais

Agradecimentos

Agradeço primeiramente a Deus que meu deu forças para chegar até aqui e me permitiu ultrapassar obstáculos aparentemente intransponíveis.

A meus pais, Maria Luiza e Pedro Marques, que, com muito carinho e amor, me incentivaram a concluir o curso de graduação. Para eles eu dedico este trabalho como presente por tudo que eles já fizeram por mim e ainda podem fazer. Amo vocês.

À minha grande amada, Andréa Pimentel, que se mostrou muito compreensiva ao entender que este trabalho deveria contar com uma grande dedicação da minha parte e, por isso, eu não teria muito tempo para lhe dar atenção. Agradeço também a ela por todo o amor carinho que me foi dado durante todo o período em que estivemos juntos.

A Eduardo Almeida, que participou ativamente de toda a realização deste trabalho, revisando os capítulos e apresentando sugestões, as quais foram de grande importância para a conclusão com sucesso deste.

Aos meus colegas de trabalho Bruno Jamir e Bruna Bunzen, que se mostraram presentes em momentos de dúvida, ajudando a esclarecê-las. A Neildes Vieira, gerente do projeto em que estou alocado atualmente, pela compreensão em disponibilizar alguns dias de expediente para que eu concluísse este trabalho.

A todos os amigos do Centro de Informática pelos momentos de descontração durante todo o período da graduação.

Ao professor Silvio Meira pelas críticas e sugestões apresentadas.

Resumo

Diante do constante crescimento do mercado voltado para tecnologia *Java 2 Micro Edition* (J2ME), a necessidade de desenvolvimento de aplicações com maior qualidade em um menor espaço de tempo se tornou primordial para a obtenção de sucesso. Em contrapartida, o rápido progresso dos dispositivos móveis que utilizam essa tecnologia faz com que essas aplicações possam ser desenvolvidas de várias maneiras possíveis, principalmente no que diz respeito à interface gráfica com o usuário.

Este trabalho descreve o desenvolvimento de um *framework* responsável pela criação de interface gráfica com o usuário em J2ME, diminuindo, assim, o esforço de codificação do desenvolvedor e deixando o aspecto visual da aplicação mais atrativo. Para criação do *framework*, foi utilizado um método de Desenvolvimento Baseado em Componentes e, durante aplicação do método, são sugeridas adaptações para o contexto de J2ME.

Abstract

Toward the constant growth of the Java 2 Micro Edition (J2ME) market, the need of application development with more quality in a reduced time has become an essential point to reach success. However, the fast progress of the mobile handsets which use this technology causes that the applications may be developed in many ways, mainly in aspects related to graphical user interface.

This work describes the development of a framework responsible for creating graphical user interface in J2ME, reducing the codification effort for the developer and producing a more attractive visual aspect of the application. For the creation of this framework, it has been used a Component Based Software Development (CBSD) method and, during the method execution, some suggestions are made to adapt it to J2ME context.

Sumário

Índice de Figuras

Índice de Tabelas

Capítulo 1

Introdução

1.1. Contexto

Dispositivos móveis já estão fazendo parte da vida cotidiana de uma forma muitas vezes invisível. São aparelhos pequenos que têm poder computacional limitado e apresentam muitas funcionalidades tais como: telefone celular, tocador de *mp3*, *palmtop* entre outras. Uma poderosa característica também muito presente nestes dispositivos é a capacidade de carregar e executar softwares em Java 2 Micro Edition [1] – versão da Linguagem de Programação Java projetada para este tipo de dispositivo.

O mercado de aplicações em J2ME está em constante crescimento. Segundo o site *3G Americas* [9], que apura as estatísticas da tecnologia 3G[1] nas Américas, um percentual de 25% dos usuários habilitados para Java aumentam entre cinco e dez dólares a sua conta devido ao uso das aplicações. Este percentual resulta em um bom negócio para as operadoras. Conseqüentemente, a quantidade de aplicações desenvolvidas e utilizadas tende a aumentar.

Diante deste quadro encorajador, o desenvolvimento de novas aplicações para dispositivos móveis se fortalece. Telas com melhores definições de cores e diferentes tamanhos estão sendo usadas em grande escala. Mais aplicações são desenvolvidas em menos tempo, necessitando, assim, que a reutilização de código seja praticada com maior freqüência.

Uma característica intrínseca da API[2] nativa de J2ME é manter uma menor quantidade possível de componentes e funcionalidades não deixando o dispositivo sobrecarregado. Logo, se uma aplicação usa somente a API nativa disponível, ela não

---

[1] 3G: Tecnologia responsável pelo uso da rede de telefonia celular para transmissão de dados e voz ao mesmo tempo com altas velocidades de transmissão.

[2] *Application Program Interface*: conjunto de rotinas, protocolos e ferramentas para construção de sistemas computacionais.

deverá apresentar um aspecto visual rico e atrativo. Para atingir este objetivo, o software deve usar componentes gráficos de baixo nível que se baseiam em desenhar a tela pixel a pixel ou, no máximo, utilizando primitivas gráficas tais como: reta, quadrado ou círculo, por exemplo. Assim sendo, o desenvolvimento torna-se trabalhoso e demorado.

Este trabalho se propõe a criar um *framework* que encapsule componentes gráficos básicos e libere o desenvolvedor do trabalho de utilizar a API de baixo nível. Desta forma, as aplicações poderiam ser desenvolvidas em menos tempo com base em reutilização e apresentariam um aspecto visual atrativo.

1.2. Motivação

Dois pontos principais motivaram a realização deste trabalho:

- Poder observar o resultado deste trabalho aplicado em projetos de desenvolvimento reais em J2ME: projetos reais têm a necessidade de construir interface gráfica a partir de componentes prontos para reduzir o tempo de desenvolvimento e aumentar a qualidade do produto final. Isto permite que o resultado deste trabalho seja observado na prática; e
- Uma pesquisa realizada com 20 colaboradores do C.E.S.A.R. [54] e 2 alunos do Centro de Informática – UFPE [55] (ver Apêndice A) permitiu observar que, considerando o contexto de projetos em J2ME: 43% dos participantes acreditam que o uso de componentes aumentou entre 20% e 40% a produtividade no projeto; 45% consideram a criação de interface com o usuário a mais difícil das camadas; 53% dizem que entre 40% e 60% do tempo dos projetos que participaram foi gasto na criação de interface com o usuário; 80% afirmam que é viável a criação de interface com o usuário em J2ME utilizando componentes prontos, sem alteração de código fonte, em algumas partes do projeto. Essa pesquisa constatou que a criação de um *framework* que agregasse um conjunto de componentes para criação de interface gráfica poderia ter grande utilidade em projetos J2ME.

1.3. Visão geral do trabalho

Este trabalho está organizado em cinco capítulos e um apêndice. O primeiro capítulo contém esta introdução.

O Capítulo 2 apresenta uma visão geral sobre o Desenvolvimento Baseado em Componentes (DBC), discutindo, detalhadamente, alguns dos vários conceitos de componente presentes na academia. O capítulo também mostra métodos de DBC, fazendo uma análise deles e escolhendo qual é melhor aplicável para contexto deste trabalho. Por fim, o capítulo descreve qual é o processo a ser seguido para viabilizar a aplicação do método escolhido.

O Capítulo 3 expõe um panorama geral da plataforma J2ME, apresentando as outras edições de Java e mostrando qual é o ambiente de desenvolvimento utilizado para o contexto do trabalho. O capítulo também enumera algumas boas práticas de programação e detalha como se constrói aplicações, mais precisamente, interface gráfica na plataforma.

O Capítulo 4 apresenta o estudo de caso, que consiste na criação de um *framework* para desenvolvimento de interface gráfica em J2ME e mostra como foi feita a aplicação do método de DBC escolhido no Capítulo 2, detalhando cada fase do processo e sugerindo adaptações para o contexto de J2ME. Em seguida, são expostos os resultados obtidos após aplicação do método e quais são os passos a serem seguidos para criar novas aplicações a partir do *framework* desenvolvido. Por fim, é feita uma análise do estudo de caso indicando pontos fortes e fracos, limitações do processo e lições aprendidas.

O Capítulo 5 apresenta as considerações finais sobre este trabalho, ressaltando suas contribuições e trabalhos futuros.

Capítulo 2

Desenvolvimento Baseado em Componentes (DBC)

Em inglês, a expressão *Component Based Software Development* (CBSD) tornou-se uma palavra muito utilizada na área de engenharia de software, nos últimos anos. A indústria de software e a academia se propõem a aceitar que a idéia de construir sistemas grandes, a partir de peças menores (componentes), já anteriormente desenvolvidos, pode aumentar a produtividade durante o período de desenvolvimento, além de também aumentar a qualidade do produto final. Essa idéia se encaixa em um dos princípios básicos para solução de problemas computacionais: dividir para conquistar. Neste contexto, este capítulo apresenta os conceitos essenciais discutidos na área de Desenvolvimento Baseado em Componentes (DBC).

O capítulo está organizado do seguinte modo: a Seção 2.1 apresenta os conceitos básicos sobre reutilização de software. Em seguida, a Seção 2.2 discute algumas definições de componentes de software, acompanhada da Seção 2.3 que apresenta a definição a ser adotada neste trabalho. A Seção 2.4 e 2.5 discutem, respectivamente, os principais atributos de componentes e as taxonomias para sua classificação. Na Seção 2.6 são apresentados alguns métodos de DBC e uma breve análise é realizada, mostrando qual método mais se aplica para este projeto. Por fim, a Seção 2.7 apresenta o método *UML Components*, o qual será o foco deste trabalho.

2.1. Reutilização de software

O conceito de reutilização de software não é novo. Desde que começaram a ser desenvolvidos os primeiros sistemas computacionais, havia um pensamento de reutilizar partes do código em outros sistemas de modo a reduzir a complexidade, os custos e aumentar a qualidade, uma vez que os artefatos reutilizados já foram bem testados [22].

A reutilização pode ser feita de diversas formas dentro de diferentes contextos. Por exemplo, a reutilização pode ser realizada utilizando-se padrões de projeto [6] e estruturando da mesma forma sistemas que se propõem a resolver problemas semelhantes. Outro exemplo de reutilização é a Engenharia de Domínio (ED). Segundo Pressman [26], a intenção da Engenharia de Domínio é identificar, construir, catalogar e disseminar um conjunto de componentes de software que possua aplicabilidade para

software existente ou futuro, dentro de um particular domínio de aplicações. Assim sendo, a ED estabelece mecanismos que sistematizem a busca e a representação de informações do domínio, de tal forma a facilitar ao máximo a sua reutilização.

Uma outra forma de reutilização de software e talvez a mais conhecida é a utilização de componentes. Se for considerado que um software é composto por um conjunto de partes, logo, todo software é composto por componentes, já que esses resultam da decomposição do problema, uma técnica padrão para solução de problemas computacionais [13] (dividir para conquistar). Neste contexto, componentes e reutilização se complementam, ou seja, a partir do momento em que componentes são usados para a construção de um sistema, isso automaticamente indica a reutilização de software.

Atualmente, a grande necessidade de manutenção em sistemas de software tornou-se um fator crucial para empresas, uma vez que aproximadamente cinqüenta por cento dos custos de desenvolvimento são envolvidos com tarefas de manutenção [25]. Grandes sistemas construídos em meados da década de 1970 e 1980, por exemplo, precisaram ser completamente remodelados e refeitos para que atendessem aos mesmos requisitos e outros que surgiram durante o tempo de uso do sistema. Esse quadro encorajou engenheiros e arquitetos de software a projetar os sistemas de tal forma a facilitar a manutenção e fazer com que partes ou "pedaços" pudessem ser substituídos ou adaptados atendendo às novas necessidades do mercado. Assim, o uso de componentes vem a ser uma das soluções mais indicadas, uma vez que, ao invés de substituir todo o sistema com o passar dos anos, por exemplo, seriam adicionados, removidos ou substituídos componentes, adaptando-o às mudanças de requisitos ou às novas exigências.

De acordo com Sametinger [16] (pág. 05): "*Componentes de software reusáveis e adaptáveis no lugar de aplicações grandes e monolíticas são o ponto-chave para o sucesso de fábricas de software*".

2.2. Componentes de software

Assim como o conceito de reutilização de software, explicado na Seção 2.1, a idéia de componentes também não é nova. Na literatura, podem ser encontradas diversas definições para o termo componente de software. Em 1968, diante do surgimento dos conceitos de reutilização de software e componentes, McIlroy [22] propôs que componentes de software pudessem ser largamente aplicados em diferentes máquinas e para diferentes usuários e deveriam estar disponíveis em categorias de famílias, conforme sua precisão, robustez, generalidade e performance. Para ele, componentes poderiam manter a produção de software em massa industrialmente, como em uma linha de montagem.

Trinta anos depois, Sametinger [16] definiu que: "*Componentes são artefatos autocontidos que nós identificamos claramente em nossos sistemas. Eles têm uma interface, descrevem e/ou executam funções específicas, são documentados separadamente e apresentam um status de reutilização bem definido*".

Outra definição bem aceita na literatura é a de Heineman [24], em 2001, na qual ele diz que: "*Um componente de software é um elemento de software que está de acordo com um modelo de componente e pode ser implantado independentemente sem alterações*". Para Heineman, "*um elemento de software contém uma seqüência de instruções que descrevem computações a serem executadas por uma máquina*". Ele também define um modelo de componente como sendo "*uma definição específica de padrões de interação e composição utilizados pelos componentes*".

Uma definição satisfatória é a apresentada por Szyperski [23], em 2002, que diz: "*Um componente de software é uma unidade de composição com interfaces especificadas contratualmente e dependências explícitas somente de contexto. Um componente de software pode ser implantado independentemente e é usado para composição com terceiros.*"

É notável que mesmo hoje, trinta e sete anos após a definição dada por McIlroy, conceituar componentes não é uma tarefa fácil. Em um dos principais livros sobre a área, Heineman [24] relata que o esforço empregado para a parte de definição de componentes, considerando outras definições, foi de aproximadamente 18 meses. Esta última definição de Szyperski, citada no parágrafo anterior, apresenta-se atualmente mais próxima do consenso dentro da academia. Este trabalho utiliza esta definição e na seção a seguir irá apresentá-la com maiores detalhes.

2.3. Definição detalhada de componentes segundo Szyperski

Segundo Szyperski, os termos **componente** e **objeto** são muitas vezes usados com o mesmo intuito. Assim, uma diferenciação torna-se extremamente essencial de modo a evitar problemas futuros.

De acordo com Szyperski, um componente:

i. **É uma unidade que pode ser implantada independentemente** – isso implica que um componente pode ser implantado separadamente do seu ambiente e de outros componentes e leva a consideração de que um componente também não pode ser parcialmente implantado;

ii. **É uma unidade de composição de terceiros** – para que um componente faça parte da composição de terceiros, é necessário que ele seja autocontido. Isso também remete a necessidade do componente vir atrelado a uma documentação

que especifique claramente o que ele precisa e o que ele provê. Em outras palavras, um componente precisa encapsular a sua implementação e interagir com o ambiente onde está inserido através de interfaces bem definidas; e

iii. **Não apresenta estado observável externamente** – isto significa que um componente não pode ser distinguido por cópias dele mesmo. Ou seja, componentes não são instanciados como objetos o são.

2.3.1. Objetos

Os conceitos de instância, identidade e encapsulamento levam ao conceito de objeto. Em contraste com o conceito de componente, citado na seção anterior, um objeto [23]:

i. **É uma unidade de instanciação que tem um identificador único** – este identificador aponta unicamente para o objeto independente de mudanças em seu estado durante todo o seu ciclo de vida. É o caso, por exemplo, de um carro que teve seus quatro pneus trocados, mas continua sendo um carro;

ii. **Pode ter seu próprio estado e este pode ser observado externamente** – o objeto permite a observação de seu estado interno através de sua classe; e

iii. **Encapsula seu estado e seu comportamento** – a classe do objeto também permite o encapsulamento do seu estado e comportamento.

2.3.2. Componentes e Objetos

Componentes e objetos estão intrinsecamente interligados. Um componente tende a existir através de objetos ou poderia normalmente consistir de uma ou mais classes ou protótipos imutáveis de objetos. Além disso, não existe a necessidade de um componente ser composto somente de classes. Ele pode ter funções, procedimentos ou até mesmo variáveis globais (estáticas), mas que não venham a ser observáveis externamente.

Neste contexto, um questionamento poderia ser: qual seria então a diferença entre o estado mantido por um objeto criado por um componente e o estado mantido por um componente? Um estado mantido por um objeto tem como abstração a sua referência. Por outro lado, um componente que não mantém um estado observável externamente não pode nem manter referências para objetos criados internamente. Assim, uma referência para o componente (seu próprio nome completo) não pode ser usada para recuperar objetos encapsulados por ele.

2.3.3. Interfaces

A definição de Szyperski diz que "*Um componente de software é uma unidade de composição com **interfaces especificadas contratualmente**...*". Neste contexto, interfaces são, resumidamente, os pontos de acesso aos serviços providos por um componente. Naturalmente, um componente pode apresentar múltiplas interfaces

correspondendo a múltiplos pontos de acesso. Cada ponto de acesso pode ser acessado por diferentes clientes de acordo com sua necessidade. É importante enfatizar o aspecto contratual no conceito de interface, já que os clientes e os componentes são desenvolvidos completamente separados e por causa desse contrato firmado entre eles é que a interação funciona.

2.3.4. Dependências explícitas de contexto

O conceito introduzido por Szyperski na definição de componente de software diz respeito ao ambiente em que o componente será inserido. Em outras palavras, a definição necessita da especificação do que o ambiente de implantação do componente precisará prover para que ele funcione corretamente. Essas necessidades são chamadas de *dependências de contexto*, referenciando ao contexto de composição com outros componentes e implantação. O modelo de componente define as regras de composição e a plataforma do componente define as regras de implantação, instalação e ativação do mesmo.

2.4. Atributos de componentes

Para melhor classificar componentes, é necessário esclarecer alguns atributos inerentes à definição. Segundo Sametinger [16], seis atributos são essenciais para o desenvolvimento baseado em componentes: *Funcionalidade*, *Interatividade*, *Concorrência*, *Distribuição*, *Formas de Adaptação* e *Controle de qualidade*. As seções seguintes apresentam cada atributo em detalhes.

2.4.1. Funcionalidade

A funcionalidade de um componente é essencial para sua reutilização em determinados contextos. Dependendo da quantidade de operações executadas pelo componente, este pode ser muito específico para um determinado contexto ou muito geral, ou até mesmo incompleto. O conceito de funcionalidade é completado com os conceitos de *aplicabilidade*, *generalidade* e *completude*.

i. A *aplicabilidade* de um componente indica o quão interessante ele se apresenta para um determinado domínio de aplicação.

ii. A *generalidade* demonstra se um componente tem sua funcionalidade mais específica ou não. Por exemplo, um componente que ordena números tem menor generalidade do que um componente que ordena objetos. Ou seja, quanto maior a generalidade de um componente, maior a sua aplicabilidade.

iii. A *completude* descreve se o componente oferece a funcionalidade que se espera que ele ofereça dentro dos cenários previstos. Um exemplo claro de um componente incompleto é uma tela de *menu* que precisa ser desenhada no

dispositivo, como um celular, e não implementa o método *paint* para mostrar o seu conteúdo.

2.4.2. Interatividade

Componentes interativos recebem de outros ambientes, que podem ser componentes ou sistemas, entradas diferentes em situações diferentes. Para entender melhor o conceito de interatividade, pode-se traçar um paralelo entre funções e objetos, mostrando claramente como componentes podem ser ou não ser interativos.

Funções transformam um sistema de um estado inicial para um estado final através de alguma computação. Elas não têm memória, ou seja, não mantêm estado e para uma mesma entrada a resposta será sempre a mesma. Já os objetos, reagem a mensagens recebidas por outros objetos realizando alguma computação e reenviando mensagens para esses objetos. Um objeto, ao contrário de funções, mantém seu estado interno e pode reagir de formas diferentes às mesmas entradas de acordo com esse estado.

Um atributo que influencia a reutilização de um componente é a interatividade. Um componente pode interagir com outros componentes ou com o usuário. A principal meta a ser atingida quando se fala em componentes interativos é: alta coesão e baixo acoplamento. Isso significa que um componente deve ter um alto grau de unidade conceitual, isto é, ele, por si só, já encapsula todo o seu significado e não deve apresentar dependências de outros componentes. Um exemplo de um componente coeso e com baixo acoplamento pode ser a classe *Frame* de J2SE [2]. Esta classe representa uma moldura que é apresentada na tela para o usuário e nela podem ser inseridos outros componentes com suas funcionalidades, como, por exemplo, *Button* ou *TextField*.

2.4.3. Concorrência

A execução paralela de componentes no decorrer do tempo chama-se de concorrência. Um componente concorrente torna-se não determinístico, ou seja, pode apresentar diferentes resultados para as mesmas entradas, justamente por não saber exatamente qual a seqüência de instruções que está sendo executada. Dessa forma, essa propriedade é utilizada para solucionar problemas não determinísticos por não precisar de execução seqüencial. A concorrência é usada também para diminuir o tempo de execução, reduzindo o tempo inativo do processador.

2.4.4. Distribuição

Sistemas distribuídos, por definição, são sistemas fisicamente ou logicamente separados, o que aumenta o poder de processamento e flexibiliza a capacidade de expansão [27]. Com a popularização da Internet, muitas aplicações funcionam tendo como base de processamento outros computadores remotos, como, por exemplo, *web services* [30]. Um componente que tem esse atributo de distribuição deve suportar sua inserção em ambientes distribuídos. A grande preocupação quando se utilizam componentes em

ambientes distribuídos está no fato de que esses ambientes necessitam de grande controle de concorrência, o que pode vir a diminuir a reusabilidade.

2.4.5. Formas de adaptação

Antes de um componente ser reusado de fato dentro de um sistema, é necessário que ele passe por algumas adaptações. Essa fase pode ser chamada de *otimização* de um componente, na qual ele vai ser submetido a uma série de alterações que não afetam o seu funcionamento essencial. Posteriormente, um componente pode passar por uma *modificação*, o que não é recomendado porque, na maioria dos casos, diminui-se a reusabilidade, já que a sua funcionalidade primária pode estar sendo alterada.

Um exemplo simples de otimização é usar um componente que monta a tela de menu para um aplicativo rodando em um celular e, como as telas dos celulares variam de tamanho, esse componente receberia uma otimização para funcionar bem naquele determinado tamanho de tela. Nesse exemplo, não haveria mudança no código.

Já um exemplo de modificação seria copiar o código fonte de um componente e alterá-lo para deixá-lo compatível com a aplicação em que será inserido. O ponto negativo desta ação é que várias versões deste componente seriam criadas podendo trazer inconsistências ao código.

2.4.6. Controle de qualidade

Garantir a qualidade de um componente é uma tarefa extremamente complexa. Por outro lado, verificar essa qualidade é ainda mais difícil. O processo é tão complexo que não existe nenhuma verificação formal até mesmo para componentes pequenos. A grande variedade deles, escritos em várias linguagens de programação e sistemas operacionais diferentes, ainda não torna possível a avaliação de componentes de tal forma a colocá-los um selo ou carimbo atestando sua qualidade. Alguns trabalhos nesta linha vêm investigando o assunto e apresentando resultados significativos, como o trabalho proposto por Álvaro [28][29]. Neste trabalho, o objetivo é definir um processo de certificação para componentes de software. No entanto, nos métodos e processos atuais, a idéia é montar um sistema tolerante à falha, mesmo que ele seja composto de componentes falíveis.

Alguns artifícios são usados para assegurar que um componente também seja tolerante à falha. Alguns deles são: pré-condições, pós-condições, constantes e exceções [16]:

- **Pré-condições** são expressões *booleanas* que checam se os argumentos dados como entrada são válidos e se o objeto encontra-se num estado aceitável para execução daquela tarefa;

- **Pós-condições** testam, após o término da tarefa executada, se ela foi completada com sucesso de acordo com o contrato que o método se propõe a executar;
- **Constantes** são usadas toda vez que um componente ou objeto passa o controle para outro; e
- **Exceções** representam outra forma de garantia de qualidade para um componente. Elas são utilizadas para manipular situações atípicas que acontecem durante a execução do código. Por exemplo, uma exceção pode ser levantada no momento em que o sistema requisita a leitura de um arquivo e o mesmo encontra-se corrompido.

2.5. Taxonomia para classificação de componentes

Após o entendimento dos principais conceitos sobre componentes de software, uma questão chave que emerge diz respeito à taxonomia de componentes. Uma taxonomia facilita na classificação e recuperação de componentes e também auxilia na certificação dos mesmos, que objetiva atrelar a eles um "selo de qualidade" depois que tenham passado por um processo que os certifique. Este processo atribui níveis de qualidade ou de maturidade aos componentes, garantindo a qualidade de um produto final que utilizou DBC.

Este trabalho apresenta duas taxonomias propostas por dois grandes pesquisadores na área.

2.5.1. Taxonomia segundo Sametinger

Antes de entender como está estruturada a taxonomia de componentes de acordo com as idéias de Sametinger [16], é fundamental entender o conceito de composição, que é o processo de construir aplicações conectando componentes através de suas interfaces. Alguns tipos de composição são:

- **Textual**: um exemplo prático são os componentes parametrizáveis, usados como *templates* por outros componentes que atribuem valores aos parâmetros;
- **Funcional**: pode ser considerado o mais famoso mecanismo de composição. Baseia-se em funções que recebem parâmetros. Esses parâmetros são passados no momento da chamada da função;
- **Modular** (ou de pacotes): acontece quando funcionalidades diferentes do sistema são divididas em módulos distintos. Cada módulo pode estar dentro de um pacote, daí a interligação dos conceitos. Ou seja, um módulo ou pacote pode agregar vários componentes juntos;

- **Orientada a objetos**: tem como alicerce os conceitos de orientação a objetos como herança, encapsulamento e polimorfismo. Um componente pode herdar de outro sem perder compatibilidade. Ou então, ativar diferentes componentes através de uma única chamada de função devido ao polimorfismo;
- **Subsistemas**: são sistemas menores que interligados formam um sistema maior. Um subsistema pode teoricamente "trabalhar" sozinho sem interdependências, podendo ter vários módulos e conseqüentemente vários componentes;
- **Modelo de objetos**: para permitir que a composição de componentes funcione entre plataformas diferentes, um modelo de objetos pode ser usado, ou seja, uma arquitetura para organização dos componentes. Um bom exemplo para este caso é a especificação CORBA [17];
- **Plataforma específica**: a composição ou integração de componentes funciona em uma única plataforma, como, por exemplo, somente na plataforma Windows; e
- **Plataforma aberta**: a composição ou integração de componentes funciona em qualquer plataforma.

A Figura 1 mostra uma possível taxonomia aplicável a componentes de software:

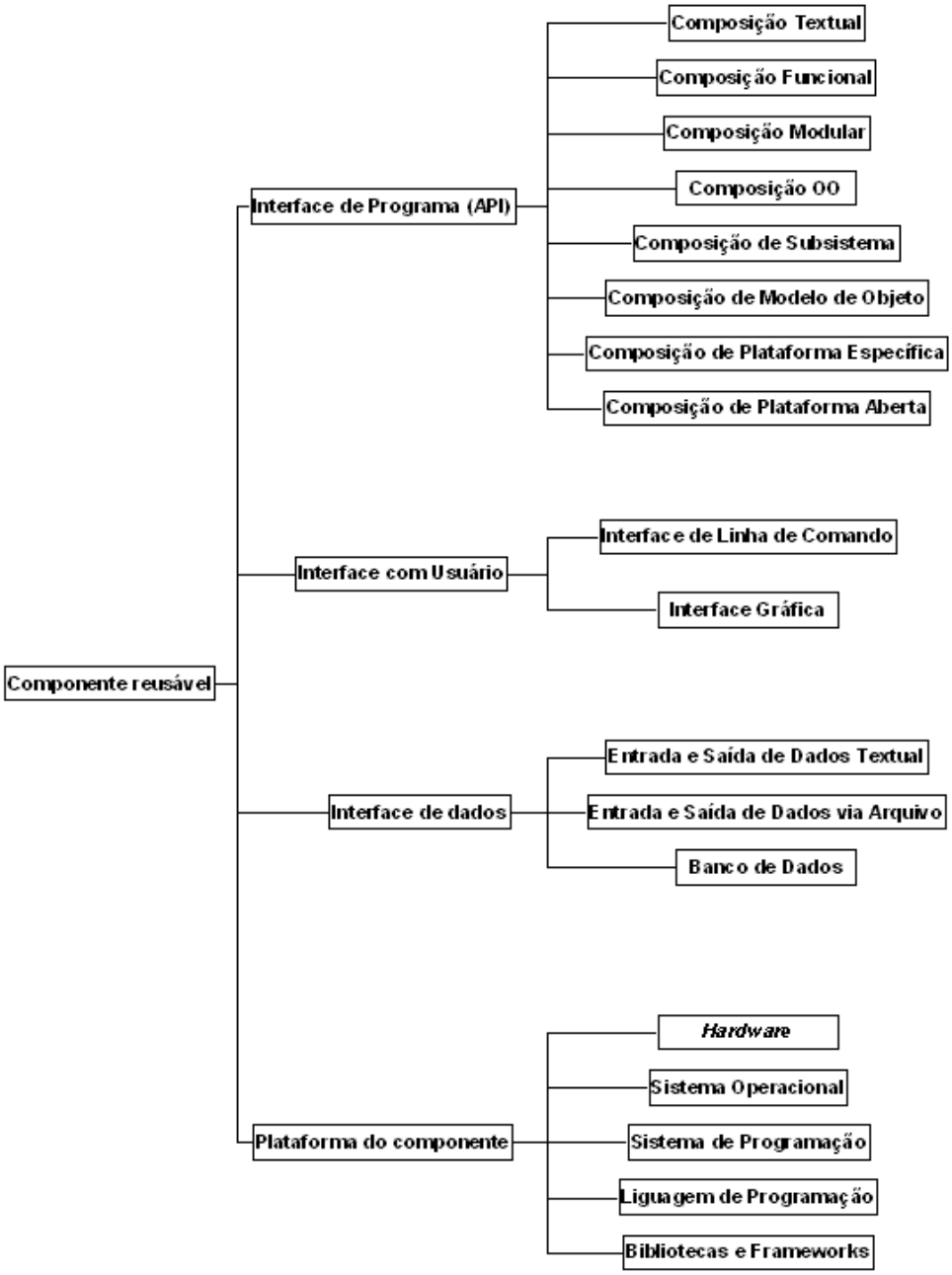

Figura 1: Taxonomia de Componentes, segundo Sametinger [16].

2.5.2. Taxonomia segundo Heineman

Segundo Heineman [24], os componentes não são criados de forma idêntica. Eles diferem em complexidade, escopo, nível de funcionalidade, conjunto de habilidades necessárias para usá-los e infra-estrutura requerida. Para facilitar na diferenciação, o autor divide componentes em três categorias, como mostra a Figura 2.

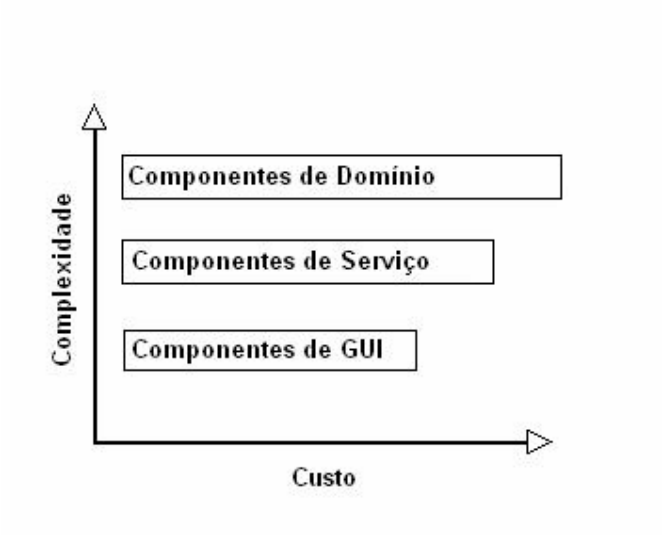

Figura 2: Taxonomia de Componentes, segundo Heineman [24].

Esta definição diferencia os componentes através de custo e complexidade. A seguir cada um dos tipos é explicado em detalhes, seguindo a ordem crescente de complexidade:

- **Componentes de GUI**: no mercado são os componentes mais encontrados, já que são os mais simples de se construir e apresentam um nível de complexidade baixo. Exemplos simples podem ser: botões, janelas, listas, campos de texto. Reusar este tipo de componente pré-fabricado é facilitado e apresenta um retorno rápido. O aumento de produtividade pode chegar a até 40 % reusando-se componentes desta categoria [24];
- **Componentes de Serviço**: esta categoria apresenta um pouco mais de complexidade. Estes componentes provêem serviços comuns necessários a aplicações. Eles incluem acesso à banco de dados, acesso a serviços de mensagens, transações e integração de sistemas. Uma característica peculiar aos componentes de serviço é que eles usam uma infra-estrutura adicional para executar suas funções, considerando também que eles podem integrar sistemas diferentes e por isso têm o custo de implantação elevado. O aumento de produtividade com o reuso de componentes de serviço pode chegar a 150 %, mas esse fator não pode desconsiderar o alto custo de implantação [24]; e
- **Componentes de Domínio**: é o tipo de componente mais difícil de construir e reusar. Heineman considera componentes de domínio somente aqueles que realmente podem ser reutilizados, já que os componentes de domínio não reutilizáveis irão apresentar o mesmo impacto dos componentes de serviço na organização. Por exemplo, uma aplicação para uma seguradora pode incluir módulos de Pagamentos, Políticas e Recompensa, os quais dificilmente seriam reusados em outros domínios além desse. Se for construído um diagrama de

classes da aplicação, pode-se observar que os componentes de domínio são a chave da abstração. Eles não são geralmente monolíticos, pois fazem interação com outras abstrações, por exemplo, num sistema de ponto de venda o componente Pedido interage com Item de Venda. Usar componentes de domínio é consideravelmente mais custoso, já que eles necessitam de grande infra-estrutura para serem implantados, porém, apresentam um ganho de até 1000 % de produtividade [24].

2.6. Métodos de DBC

Desenvolver componentes exige uma abordagem sistemática, assim como qualquer outro processo de software. Assim, utilizar métodos ou processos para esse tipo de desenvolvimento facilita na construção de sistemas mais complexos que conectam todas as "partes". Neste cenário, pode-se justificar o grande número de pesquisas desta área resultando em alguns métodos voltados para DBC [14][31][32]. Alguns desses métodos são extensões de outros já existentes, os quais utilizam orientação a objetos. Dentre estes métodos e abordagens podem ser destacados: *Catalysis* [19][21], *PECOS* (*Pervasive Component Systems*) [20], *Rational Unified Process* (*RUP*) [32][36], *Select Perspective* [32][34][35], *UML Components* [11][14]. As seções a seguir descrevem, resumidamente, cada um dos métodos e apresenta uma breve análise dos mesmos. A ordem dos métodos apresentados é simplesmente alfabética.

2.6.1. Catalysis

Catalysis é uma iniciativa de pesquisa desenvolvida na universidade de Brighton, Inglaterra, por D'Souza e Wills [21], que resultou num método integrado de técnicas para construir Sistemas Distribuídos Orientados a Objetos. De acordo com os autores, Catalysis é um método de desenvolvimento baseado em componentes que cobre todas as fases de desenvolvimento de um componente, desde a sua especificação à sua implementação.

Este método é baseado num conjunto de princípios [21] para o desenvolvimento de software. Dentre eles, pode-se destacar: a **abstração**, **precisão**, **refinamento**, **componentes "plug-in"** e algumas **leis de reutilização**.

O princípio da abstração guia o desenvolvedor na busca de aspectos essenciais do sistema, dispensando detalhes que não são relevantes para o contexto do mesmo. O princípio da precisão objetiva descobrir erros e inconsistências na modelagem. Refinamentos sucessivos entre as transições de uma fase para outra auxilia na obtenção de artefatos cada vez mais precisos e propensos à reutilização. O princípio componentes "plug-in" suporta a reutilização de componentes, com o intuito de construir outros. Por fim, a principal lei de reutilização do método Catalysis é não reutilizar código sem reutilizar os modelos de especificações desses códigos [14].

O processo de desenvolvimento de software utilizando Catalysis se divide em três níveis: **domínio do problema** – elaborando "o quê" o sistema faz para resolver o problema; **especificação dos componentes** – onde o comportamento do sistema é descrito; **projeto interno dos componentes** – onde os detalhes internos de implementação são especificados [14].

2.6.2. PECOS (Pervasive Component Systems)

PECOS teve seu projeto iniciado em 2000 e finalizado em 2002. É um método voltado para DBC em sistemas embarcados, onde existe uma dificuldade maior em atingir os requisitos não-funcionais, tais como, resposta em tempo real. Nenhum outro método de DBC antes deste tinha sido aplicado para sistemas embarcados por algumas razões e duas delas podem ser consideradas as principais: boa parte dos colaboradores de TI não provê "produtos de prateleira" (*Commercial-off-the-shelf*) tais como sistemas operacionais para sistemas embarcados; e as características limitantes dos sistemas embarcados (potência consumida, memória, etc.) [20].

O objetivo de PECOS é prover um ambiente que suporte a especificação, composição, checagem de configuração e implantação de sistemas embarcados construídos a partir de componentes de software. O método apresenta os mesmos pontos-chave de outros métodos de DBC, porém respeita as características de sistemas embarcados. Esses pontos-chave são:

- Um **Modelo de Componente** para sistemas embarcados levando em consideração a especificação de seu comportamento, propriedades não-funcionais e limitações;
- Uma **Linguagem de Composição** baseada no modelo, para especificar componentes e suas composições, a qual suporta e apresenta uma interface de ligação com o ambiente de composição;
- Um **Ambiente de Composição** para montar aplicações embarcadas a partir de componentes, validando limitações funcionais (interfaces) e não-funcionais (potência consumida, tamanho do código), gerando a aplicação executável para dispositivos embarcados e monitorando sua execução; e
- Um **Ambiente de Componentes Leve** para instalar, rodar e testar aplicações de sistemas embarcados com recursos limitados montadas em cima de componentes, permitindo o seu gerenciamento.

2.6.3. Rational Unified Process (RUP)

O RUP é um processo de engenharia de software desenvolvido pela Rational Software Corporation e tem como base as idéias de Jacobson [36]. O método é interativo, orientado a objetos, controlado e suportado por ferramentas, podendo ser aplicado a todo

tipo de desenvolvimento de software. O RUP alcançou grande popularidade na indústria de software, especialmente entre os usuários de ferramentas Rational que oferecem suporte a todas as fases do processo [14].

De acordo com Jacobson, o RUP é mais do que um simples processo de software. Ele serve como *framework* de processo genérico, podendo ser especializado para diferentes áreas de aplicação, tipos de organização e tamanhos de sistema.

Todo o processo pode ser dividido em quatro fases: concepção, elaboração, construção e transição, com um número arbitrário de iterações. Cada fase é estruturada baseada em um conjunto de *workflows*, os quais agrupam diferentes tipos de atividades. O RUP não introduz nenhum conceito novo de modelagem, já que é completamente apoiado pela UML [38].

O conceito principal do RUP é a definição das atividades através do desenvolvimento do ciclo de vida do software, tais como especificação de requisitos, análise e projeto, implementação e testes. Dentro do RUP, o suporte a DBC é encorajado, mas ainda é extremamente influenciado pela notação UML. A abordagem do método em cima do conceito de componente ainda é limitada a pacotes de código, baseando-se em componentes em UML e diagramas de implantação. Isto pode ser provado pela própria definição de componente do RUP: “uma parte não trivial, quase independente, e substituível de um sistema que realiza uma função clara no contexto de uma arquitetura bem definida”. RUP usa o conceito de UML de subsistema com o propósito de modelagem de componentes, mas não apresenta detalhes de como esse processo é realizado [14][32].

2.6.4. Select Perspective

Originalmente criado em 1994-1995 a partir da combinação do *Object Modelling Technique* (OMT) [34] e do método *Use Case Driven Objectory* [35] com base em princípios *Rapid Application Development* (RAD), este método somente suportava a modelagem de negócio, casos de uso, classes, modelagem de interação dos objetos e modelagem de estados. Em seguida, depois do crescente interesse e importância do paradigma DBC, o método foi estendido com o intuito de prover mecanismos para modelagem de componentes. O *Select Perspective* inclui a integração da tecnologia orientada a objetos com a modelagem de negócios e de dados.

Além da modelagem de componentes, este método utiliza a notação CSC Catalyst [37] para modelagem do negócio, mantendo a ligação entre processos de negócio e casos de uso e classes associados. Para a modelagem de dados o método utiliza *Entity-Relationship Diagrams* (ERD) e provê mapeamento entre diagramas UML e modelos de dados. *Select* trata o conceito de componente usando o conceito de pacote, definido em UML como “*um mecanismo de propósito geral para organizar elementos em grupos*” [38].

Dois estereótipos básicos para pacotes podem ser diferenciados: um pacote de serviço, que contém classes com um alto nível de interdependência e servem como proposta comum de oferecer um conjunto consistente de serviços; um pacote de componentes, que representa um componente executável, ou seja, seu código fonte [32].

2.6.5. *UML Components*

O *UML Components*, extensão de UML, representa um processo para a especificação de software baseado em componentes. Neste processo, os sistemas são estruturados em quatro camadas distintas: interface com o usuário, comunicação com o usuário, serviços de sistemas e serviços de negócios. Esta estrutura representa a arquitetura do sistema. Além disso, *UML Components* propõe a utilização de UML para modelar todas as fases do desenvolvimento de sistemas baseado em componentes, contendo as atividades de definição de requisitos, identificação e descrição das interfaces entre componentes, especificação e modelagem, além de implementação e montagem.

O método utiliza uma forma simples de estender UML que é utilizar estereótipos nas entidades. Dessa forma, o propósito de UML, já bem difundido em se tratando de modelagem, pode ser seguido pelo analista. Uma preocupação que os autores desse método tiveram foi oferecer uma solução sem especificar uma plataforma de desenvolvimento.

2.6.6. Análise dos métodos de DBC

Como já foi mencionado anteriormente na Seção 1.1, este trabalho visa construir componentes para o desenvolvimento de interface gráfica para dispositivos móveis utilizando J2ME [1]. Para alcançar este objetivo, será utilizado um método de DBC que facilite o processo de análise, projeto e construção desses componentes. Analisando esses métodos existentes, optou-se por utilizar o método *UML Components* por alguns motivos descritos a seguir:

i. **Documentação**: o método *UML Components* apresenta vasta documentação de fácil acesso;

ii. **Fácil integração com UML**: o método apresenta fácil integração com UML por causa do uso de estereótipos. Alem disso, várias ferramentas disponíveis no mercado que permitem modelagem em UML suportam esta funcionalidade [39][40];

iii. **Processo flexível e simples**: o método somente adiciona o conceito de seis estereótipos e por isso torna-se simples para modelagem de componentes;

iv. **Específico para DBC**: ao contrário do RUP, por exemplo, que é um processo genérico para desenvolvimento de software, o *UML Components* se apresenta voltado exclusivamente para DBC;

v. **Plataforma aberta**: *UML Components* pode ser usado para especificar componentes em qualquer plataforma de desenvolvimento;

vi. **Bem aceito na indústria e academia**: a idéia deste método já vem sendo bem usada tanto pela indústria de software quanto pela academia [41].

Os outros métodos também foram considerados, no entanto, apresentaram alguns pontos negativos. O *PECOS* é um projeto completamente voltado para a área de sistemas embarcados e não se aplica a este projeto, o qual têm como objetivo desenvolver componentes em J2ME, não apresentando como resultado nenhum software embarcado. O *Catalysis* têm características mais voltadas para sistemas distribuídos e, além disso, apresenta-se de forma muito genérica, sem especificar, em detalhes, suas atividades, entradas e saídas. Da mesma forma o *Select Perspective* poderia ser uma solução viável, entretanto, este método pode ser melhor utilizado em aplicações maiores, que utilizam-se de banco de dados, o que não é o caso específico deste projeto. Por fim, O RUP não é viável para este trabalho, por ser um processo de software muito amplo e não voltado para o desenvolvimento baseado em componentes.

A necessidade deste projeto é de um método de DBC simples e de fácil documentação, que possa ser usado em aplicações sem muitas regras de negócios, todavia, com uma necessidade de especificação dos componentes bem estruturada. Neste contexto, o método *UML Components* se apresentou mais adequado. Na Seção 2.7 seguinte, serão detalhados o método e seus estereótipos.

2.7. UML Components

A UML foi proposta com a idéia de permitir a modelagem de sistemas e dar ao analista e ao desenvolvedor uma visão do todo, quando não há, ainda, nenhuma implementação do que será feito. No entanto, UML já possui o conceito de “componente”, então um questionamento chave é: qual o intuito de *UML Components*? De acordo com Cheeseman e Daniels [11], existem vários motivos para utilizar *UML Components*, dentre os quais podem ser destacados:

- UML foi originalmente projetada como uma linguagem de análise e projeto orientada a objeto, isto é, assume-se que haverá uma implementação orientada a objeto. O ponto principal de *UML Components* não chega a esse nível de detalhe, deixando livre a forma como os componentes serão implementados;
- *UML Componentes* apresenta um foco baseado na técnica e não em diagramas como UML; e
- UML também foi projetada com o intuito de ser estendida podendo receber a semântica desejada. O método *UML Components* estende a linguagem

adicionando estereótipos que ajudam na modelagem. A Seção 2.7.1 apresenta quais são os estereótipos que podem ser utilizados.

É possível que, no futuro, UML venha a ter um suporte completo ao desenvolvimento baseado em componentes.

2.7.1. Extensão de UML com estereótipos

Existe um grande número de mecanismos diferentes de extensão para UML, mas, provavelmente, a mais utilizada, de acordo com Cheeseman e Daneils [11] é utilizar estereótipos. Em sua versão original [38], UML permite que qualquer elemento tenha um estereótipo anexado, facilitando a extensão da linguagem. Ele é encapsulado pelo símbolo "<< >>" permitindo ao usuário criar novos elementos de modelagem.

Um estereótipo pode também ser aplicado até entre as ligações dos elementos, como, por exemplo, em uma implementação de interface. Nas próximas seções, serão apresentados os estereótipos utilizados no método *UML Components*.

2.7.1.1. *Type*

Estereótipo: `<<type>>`. Usado para representar uma classe no modelo de negócios de um componente em nível de especificação. Vale ressaltar que esse conceito não é o mesmo que classe em uma linguagem de programação como Java.

2.7.1.2. *Datatype*

Estereótipo: `<<datatype>>`. Possui a mesma representação de *Type*, mas é usado para classes que utilizam persistência.

2.7.1.3. *Interface type*

Estereótipo: `<<interface type>>`. Representa uma interface em nível de especificação. Por convenção, o nome das classes desse tipo são iniciadas com um "I". A ressalva deste estereótipo é não confundir com o estereótipo: `<<interface>>` já presente em UML. Este último serve para modelagem orientada a objetos e isso é o que exatamente se deseja evitar quando for feita a modelagem ou especificação de componentes.

2.7.1.4. *Component Specification*

Estereótipo: `<<comp spec>>`. Usado para indicar uma especificação de um componente. Este estereótipo, normalmente, está associado a um outro estereótipo `<<interface type>>` através de uma ligação de `<<offers>>`, que será explicado na próxima seção.

2.7.1.5. *Offers*

Estereótipo: `<<offers>>`. Liga uma especificação de um componente à sua interface. Este estereótipo é análogo a `<<realize>>` de UML. Ou seja, ao invés de um

componente implementar uma determinada interface, o conceito muda, de tal forma que, a especificação de um componente oferece uma interface.

2.7.1.6. *Core*

Estereótipo: <<core>>. O "*core*" subentende um "*type*" sem ocorrência de dependências, sendo assim, o tipo "*core*" pode ser encontrado a partir de um refinamento do modelo de um componente, observando quais são os tipos que não apresentam dependência com outros componentes.

2.8. Visão geral do processo de *UML Components*

Cheeseman e Daniels [11] definem um *workflow* para desenvolvimento de componentes com base no RUP, que utiliza a idéia de uma seqüência de atividades encadeadas, onde o resultado de uma atividade serve como entrada para a próxima [33]. A Figura 3 apresenta o *workflow* em questão.

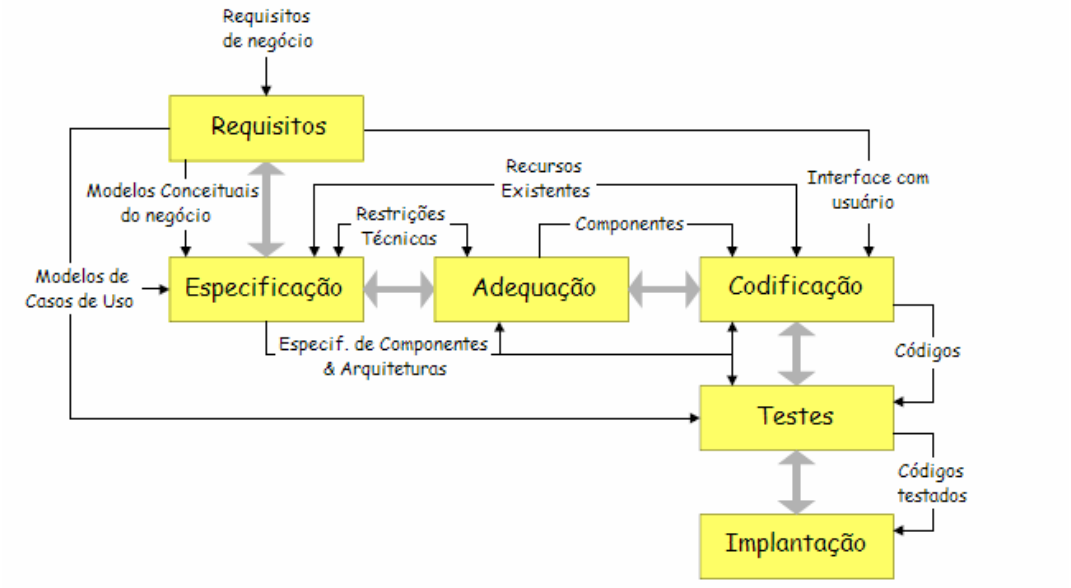

Figura 3: *Workflow* para desenvolvimento de componentes [11].

Comparando o *workflow* da Figura 3 com o encontrado no RUP, Requisitos, Testes e Implantação correspondem diretamente às atividades com o mesmo nome no RUP e, portanto, não serão abordadas em detalhes neste trabalho. As atividades de Especificação, Adequação e Codificação substituem diretamente as atividades de Análise e Projeto e Implementação do RUP. Assim, eles serão abordados em detalhes para demonstrar o processo de desenvolvimento utilizando o *UML Components*.

A Figura 4 mostra como estão subdivididas as atividades do processo.

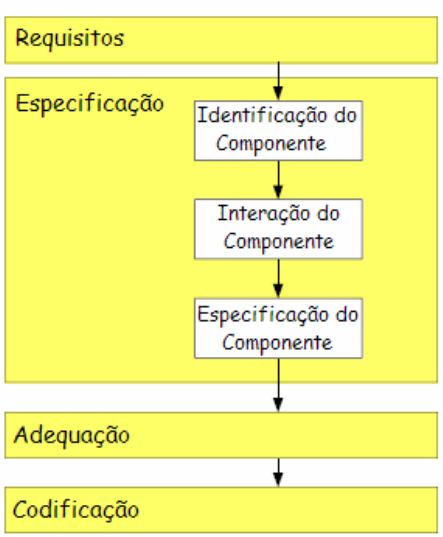

Figura 4: Detalhamento das atividades do processo de desenvolvimento baseado em componentes utilizando *UML Components* [11].

O foco central deste processo é a atividade de Especificação, onde se encontram os conceitos de *UML Components* aplicados. Esta atividade se divide em três fases: **Identificação do Componente**, **Interação do Componente**, **Especificação do Componente**. Ela recebe como entrada da atividade de Requisitos o modelo de casos de uso e o modelo conceitual do negócio. Podem ser usadas também informações previamente existentes, como, por exemplo, sistemas legados, pacotes de código, banco de dados, restrições técnicas, tais como arquiteturas ou ferramentas específicas. A atividades de Especificação gera como saída um conjunto de arquiteturas e especificações de componentes.

Estas saídas serão usadas pela atividade de Adequação. Esta atividade garante que os componentes necessários estarão disponíveis, sejam eles comprados, construídos do zero, integrados com outros componentes, reutilizados de outros componentes ou outro software. A atividade também inclue testes unitários [11] dos componentes antes da Codificação.

A Codificação integra todos os componentes com os possíveis softwares já existentes, utilizando uma interface com o usuário apropriada, formando uma aplicação que atenda ás necessidades do negócio, ou seja, aos requisitos levantados [11].

As seções que se seguem apresentam com mais detalhes as três fases da atividade de Especificação. A descrição dessas fases propostas por Cheeseman e Daniels [11] é voltada para sistemas desenvolvidos em camadas, mais especificamente para a camada de negócios, a qual apresenta-se independente da interface com o usuário. Os exemplos apresentados dentro de casa fase serão baseados em um sistema de informação

responsável por fazer o gerenciamento das operações de um hotel. Esse sistema receberá o nome fictício de SGH (Sistema de Gerenciamento de Hotel).

A seguir, é apresentada uma descrição resumida do sistema [41]:

> ”Um sistema de reserva para hotéis que permita que a reserva seja feita para qualquer hotel da cadeia. O sistema deve oferecer quartos alternativos se o hotel desejado estiver lotado. Cada hotel tem um administrador de reservas responsável por controlar as reservas. O tempo para conseguir realizar uma reserva por telefone ou mesmo pessoalmente é, em média, três minutos. Para acelerar o processo, detalhes sobre clientes antigos serão armazenados e estarão disponíveis”.

2.8.1. Identificação do componente

Esta é a primeira fase da atividade de Especificação do componente e recebe como entrada o modelo conceitual do negócio e o modelo de casos de uso, originados da atividade de definição de Requisitos. A Figura 5 apresenta o modelo conceitual do negócio.

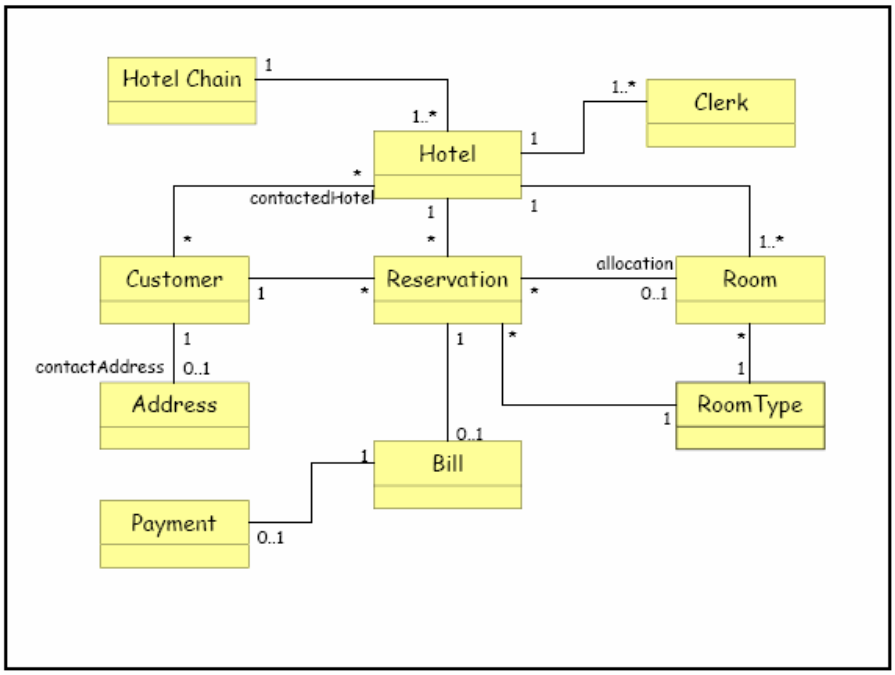

Figura 5: Modelo conceitual do negócio [11].

O objetivo da identificação do componente é criar um conjunto inicial de interfaces e especificações de componentes. Esta fase também cria um ambiente com um conjunto de artefatos que vão servir como entrada para as próximas fases.

Este conjunto é constituído por quatro artefatos: modelo de tipos do negócio (*business type model*); especificação inicial de interfaces; especificação da arquitetura de componentes; especificação inicial de componentes.

Cheeseman e Daniels [11] propõem alguns passos a serem seguidos para se ter como resultado os artefatos citados: **identificar interfaces e operações do sistema**; **identificar interfaces de negócio**; **criar especificação inicial de interfaces**; **especificar arquitetura de componentes**.

Para **identificar interfaces e operações do sistema** é necessário que os casos de uso sejam observados a cada passo de execução e, a partir deles, pode-se obter a informação de quais operações o sistema (camada de negócios) deve prover para atingir as responsabilidades definidas nos requisitos. Essas operações não são mapeadas diretamente com os passos do caso de uso numa relação de um pra um. Cada passo pode ter zero, uma ou mais operações providas pelo sistema para atender à necessidade. Como exemplo para este passo, pode-se usar um caso de uso “Fazer Reserva” do SGH. Neste exemplo uma interface pode ser definida com as operações de: “obter detalhes sobre o hotel”, “obter informações do quarto” e “fazer reserva”.

A **identificação de interfaces do negócio**, que são abstrações da informação manipulada pelo sistema, é feita a partir do refinamento do modelo conceitual, dando assim origem ao modelo de tipos do negócio. Deste refinamento, são identificados os tipos `<<core>>` (ver Seção 2.7.1.6) e criadas interfaces de negócio para cada tipo. A Figura 6 apresenta o refinamento do modelo conceitual, que resulta no modelo de tipos do negócio do sistema fictício SGH.

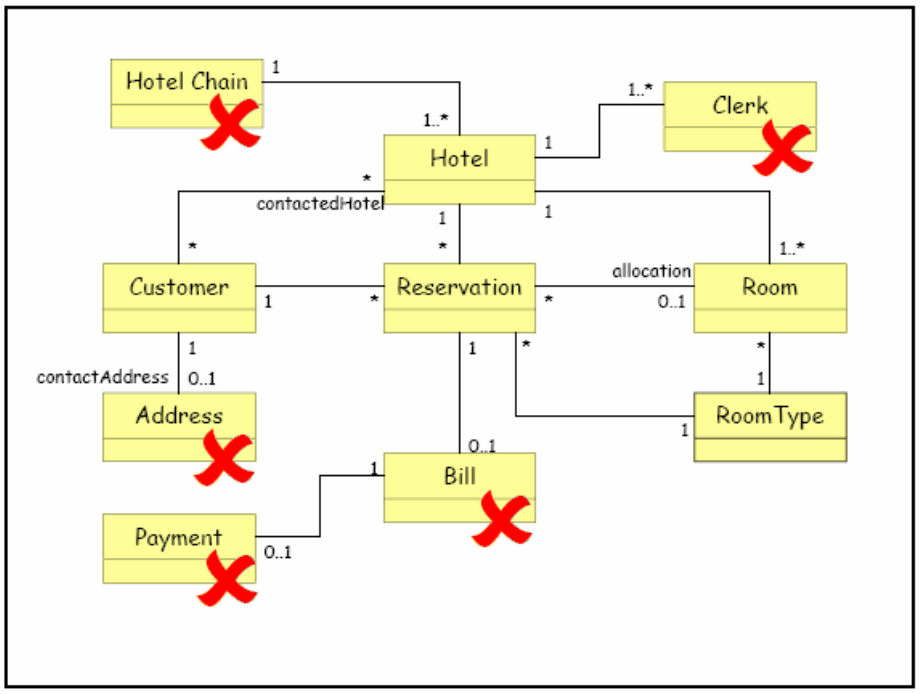

Figura 6: Refinamento do modelo conceitual do negócio resultando no modelo de tipos do negócio [11].

As eliminações de tipos feitas com um "X" na Figura 6 foram realizadas a partir de uma análise detalhada dos requisitos e casos de uso do SGH em questão. O tipo *HotelChain* foi eliminado porque a descrição do sistema não especifica que o sistema deve suportar mais de uma cadeia de hotéis. Os tipos *Payment* e *Bill* foram eliminados porque o SGH não dá suporte a operações de pagamentos e contas. Os tipos *Address* e *Clerk* foram eliminados para deixar o exemplo mais simples.

A Figura 7 apresenta um modelo inicial do diagrama de tipos do negócio como resultado do refinamento do modelo conceitual do negócio.

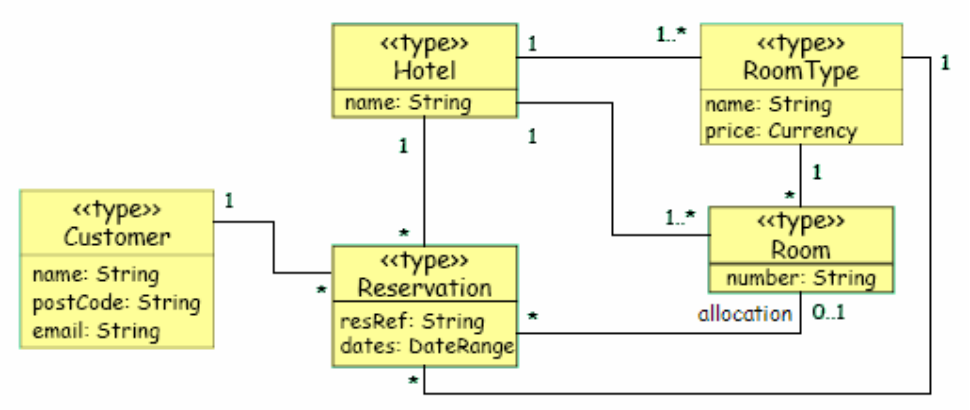

Figura 7: Diagrama inicial de tipos do negócio [11].

A partir deste modelo da Figura 7, pode se identificar os tipos `<<core>>`. O propósito maior dessa identificação é começar a pensar em quais informações são dependentes das outras e quais delas podem existir independentes de qualquer outra. Essas últimas podem ser consideradas do tipo `<<core>>`. Como UML não permite que uma entidade tenha mais de um estereótipo, o tipo `<<core>>` subentende o `<<type>>`. A análise do diagrama da Figura 7 mostra que os tipos *Hotel* e *Customer* podem ser considerados `<<core>>`. A Figura 8 apresenta os detalhes.

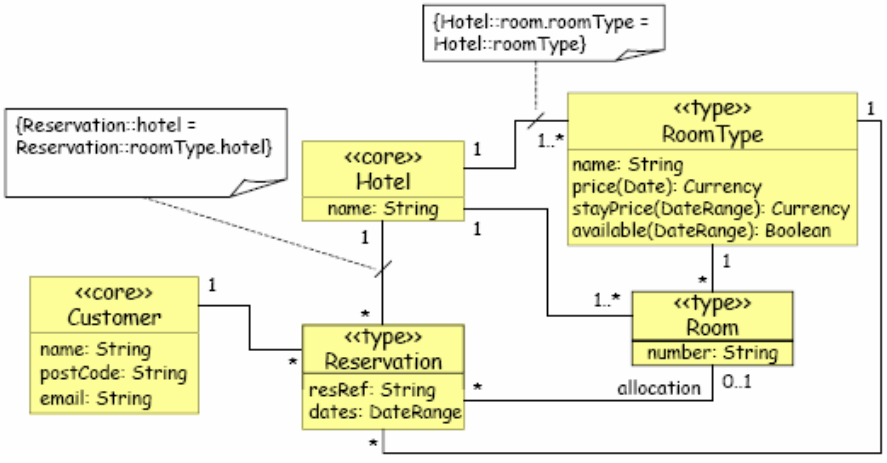

Figura 8: Identificação dos tipos core [11].

A partir da especificação dos tipos *core*, é possível **criar uma especificação inicial de interfaces** (Figura 9) destes tipos e por conseqüência **especificar a arquitetura de componentes** (Figura 10).

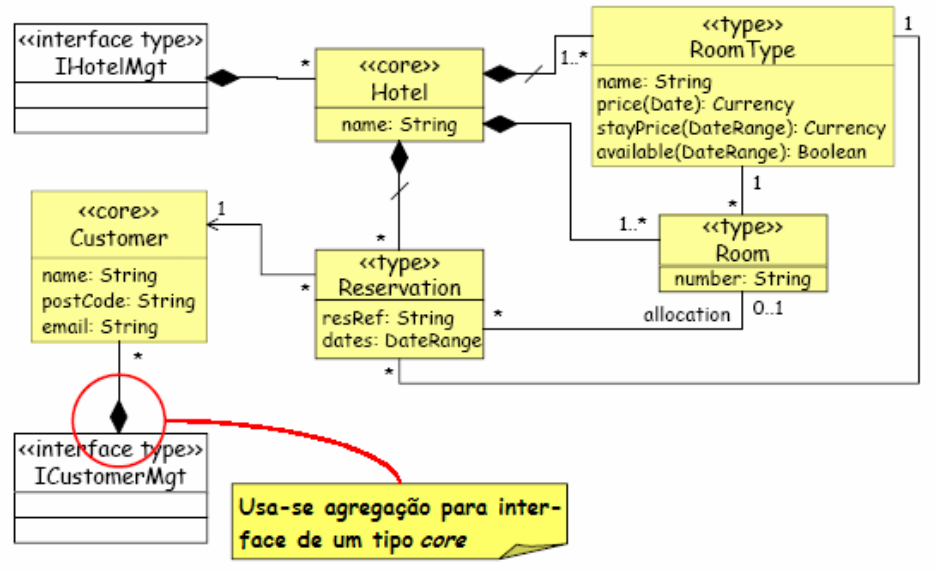

Figura 9: Especificação inicial de interfaces [11].

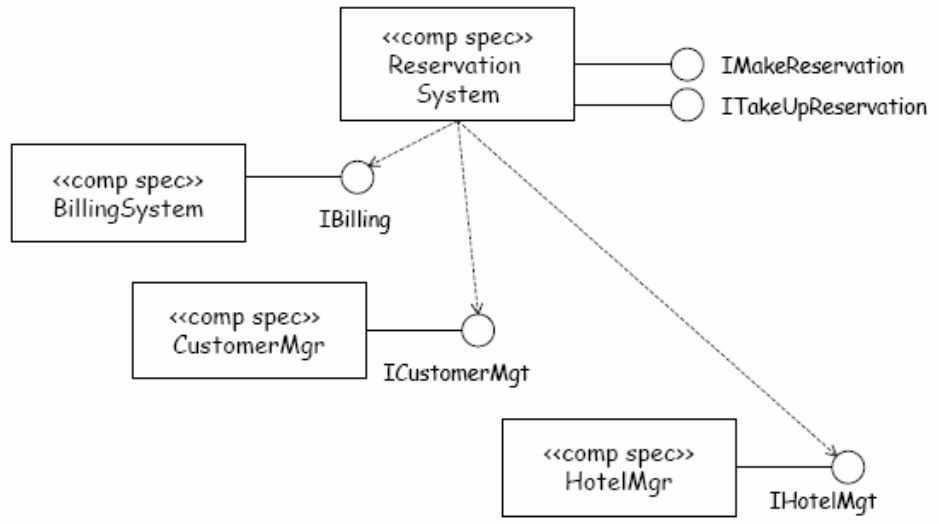

Figura 10: Arquitetura de componentes [11].

2.8.2. Interação do componente

Depois de realizada uma especificação inicial de interfaces na fase de identificação do componente, este é o momento em que as interações entre os componentes do sistema são identificadas. O resultado desta fase será um refinamento da especificação inicial de interfaces, com definição das assinaturas das operações, e do diagrama de arquitetura de componentes.

Para atingir o objetivo desta fase, alguns passos devem ser seguidos, na ordem em que são apresentados:

- **Desenvolver modelos de interação para cada operação do sistema**: isso significa analisar cada caso de uso um a um e verificar quais passos precisam de uma operação correspondente no sistema. Para auxiliar no descobrimento dessas operações, diagramas de colaboração em UML podem ser utilizados;
- **Descobrir operações das interfaces do sistema e suas assinaturas**: a partir dos diagramas de colaboração, é possível identificar as operações necessárias e, com isso, especificar as assinaturas de cada operação. Não necessariamente todas as operações precisam ser identificadas nesta fase, ou seja, os artefatos podem passar por outros refinamentos;
- **Refinar responsabilidades**: a partir do momento em que as operações das interfaces e suas assinaturas são encontradas, pode haver a necessidade de refinar as responsabilidades dos tipos do negócio. Por exemplo, um refinamento do SGH pode ser criar uma nova interface chamada *ITakeUpReservation*, responsável por fazer as reservas no hotel. Assim sendo, a interface *IHotelMgt* (ver Figura 9) não teria mais essa responsabilidade; e
- **Definir restrições necessárias**: esse é um passo importante para definir as regras de negócio do sistema. Uma restrição que pode ser ressaltada no SGH, por exemplo, é que um cliente somente pode fazer uma reserva se for cadastrado no Hotel. Essas restrições vão levar à definição de pré e pós-condições das operações do sistema na próxima fase de especificação do componente.

2.8.3. Especificação do componente

Nesta fase última da atividade de Especificação, é definido precisamente como os contratos de uso e realização são especificados. Ao final da fase, deve-se obter a especificação completa do componente e suas interfaces.

É necessário compreender os conceitos de **contrato de uso** e **contrato de realização**. Um **contrato de uso** é definido por uma especificação de interface, representado pelo estereótipo `<<interface type>>`, e pode ser entendido como um contrato entre a interface de um componente e os outros objetos que a utilizam. Um **contrato de realização** é definido por uma especificação de componente, representado pelo estereótipo `<<comp spec>>`, e pode ser entendido como um contrato entre a interface do componente e sua realização ou implementação.

Para se obter a especificação completa do componente, os seguintes passos devem ser seguidos:

- **Especificar o modelo de informação da interface.** Define um conjunto de operações e informações que são assumidas para serem manipuladas pelo objeto que oferece a interface, para propósito de especificação somente. Na implementação, o objeto que realiza a interface apresenta todas as informações e operações definidas por ela. O modelo é suficiente para mostrar os efeitos das operações. A Figura 11 exemplifica o modelo citado;

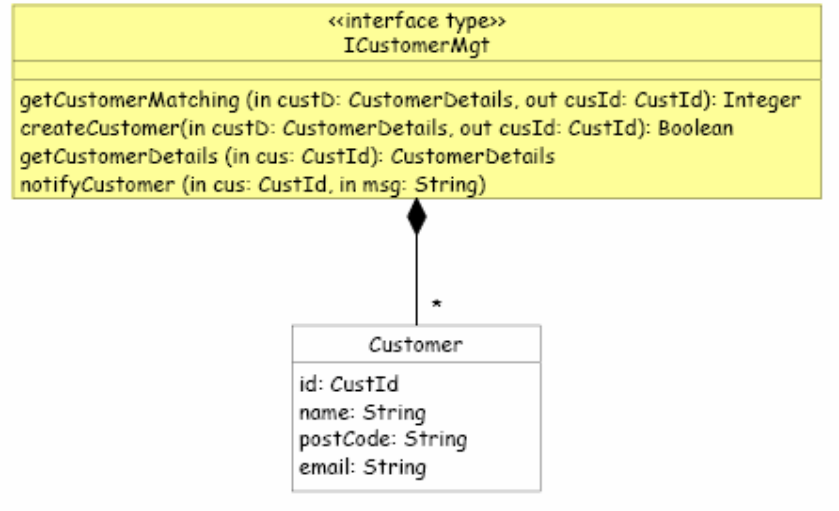

Figura 11: Modelo de informação de interface [41].

- **Definir pré e pós-condições.** Essas pré e pós-condições são usadas para definir precisamente quais as operações de cada interface com suas respectivas assinaturas. Elas podem ser escritas em linguagem natural ou utilizando uma linguagem do tipo OCL (*Object Constraint Language*). UML contém um subconjunto que consiste em uma OCL, a qual permite aos desenvolvedores adicionar restrições a modelos de objetos [43]; e
- **Montar especificação do componente.** Depois que todas as interfaces do sistema, com suas respectivas realizações ou implementações, estão especificadas, deve-se montar o modelo arquitetural final que indicará a especificação do componente. A Figura 12 apresenta um exemplo de como ficaria a especificação para o SGH.

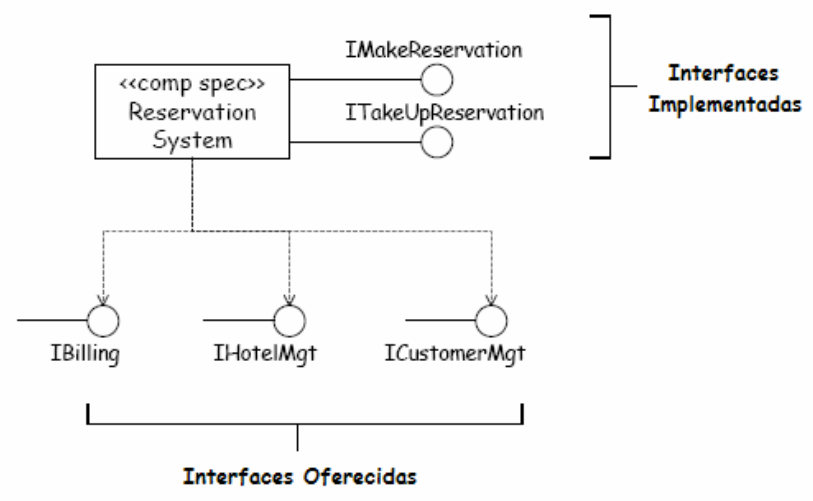

Figura 12: Especificação de componentes do Sistema de Gerenciamento de Hotel [11].

2.9. Resumo

Este capítulo apresentou os conceitos básicos sobre reutilização de software e desenvolvimento baseado em componentes. Foram mostradas algumas das principais definições de componente desde 1968, proposta por McIlroy [22], até 2002, proposta por Szyperski [23]. Esta última foi a definição utilizada para este trabalho e abordada com mais detalhes. Além disso, foram discutidos alguns dos principais atributos de componentes segundo Sametinger [16].

Uma vez explorados os principais conceitos sobre componentes, foram relatadas duas possíveis taxonomias para classificação dos mesmos. Por fim, discutiu-se sobre os métodos de desenvolvimento baseado em componentes, com especial ênfase no método *UML Components*, o qual será usado como base neste trabalho.

O capítulo seguinte apresenta um panorama geral da tecnologia J2ME, que será usada no desenvolvimento de componentes de interface gráfica para dispositivos móveis.

Capítulo 3

Java 2 Micro Edition

Após o surgimento da linguagem Java, em 1995 [3], a programação orientada a objetos recebeu um novo impulso. A linguagem surgiu com o princípio de ser portável para vários sistemas operacionais, tendo como base uma máquina virtual para processar os códigos-fonte pré-compilados. Inicialmente, antes de sua criação, em meados da década de 1990, Java tinha como foco principal criar software para sistemas embarcados. Entretanto, o time de desenvolvimento da Sun [8], empresa criadora da tecnologia, desviou seus esforços para as grandes oportunidades geradas pela Internet.

Depois que essas oportunidades foram atendidas, ao final da década de 1990, uma nova gama de oportunidades surgiu impulsionada pelo surgimento de aparelhos portáteis de comunicação [4]. Telefones celulares se expandiram, assim como dispositivos eletrônicos portáteis, como, por exemplo, *Personal Digital Assistants* (PDAs). Neste momento, a atenção do time da Sun voltou-se para estes dispositivos. Entretanto, ao invés de eles adicionarem novas APIs à linguagem, resolveram desmontar a linguagem e a máquina virtual e reduzi-las ao mínimo suficiente para prover inteligência aos sistemas embarcados. Isso foi necessário devido às restrições de recurso destes dispositivos. O resultado desse esforço foi a *Java 2 Micro Edition* (J2ME), uma versão reduzida da máquina virtual e da API de Java projetada para funcionar em dispositivos com limitação de recursos.

Este capítulo trata da plataforma J2ME, a qual serve como base para o desenvolvimento de componentes apresentados no próximo capítulo. A Seção 3.1 apresenta uma visão geral da tecnologia, com detalhes sobre as edições de Java e as configurações e perfis existentes em J2ME. A Seção 3.2 apresenta o ambiente de desenvolvimento para J2ME, seguida da Seção 3.3, que mostra algumas boas práticas de programação ao usar a tecnologia. Na Seção 3.4, serão apresentados os conceitos básicos da programação em J2ME e, por fim, a Seção 3.5 expõe os conceitos básicos para criação de interface gráfica com o usuário.

3.1. Visão geral da tecnologia

Nesta seção, serão apresentadas basicamente quais as edições de Java e onde J2ME se encaixa nesse panorama. Além disso, também será mostrado como J2ME está estruturada com configurações e perfis.

3.1.1. Edições de Java

As plataformas Java atualmente disponíveis são:

- **Standard Edition** (J2SE): projetada para execução em máquinas simples de computadores pessoais e estações de trabalho;
- **Enterprise Edition** (J2EE): suporte interno para *servlets*, JSP (*Java Server Pages*) e XML (*Extensible Markup Language*). Esta edição é destinada a aplicativos baseados no servidor; e
- **Micro Edition** (J2ME): projetada para dispositivos com memória, vídeo e poder de processamento limitados.

É importante observar que, em 1998, a Sun apresentou o nome “Java 2” (J2) para coincidir com o lançamento do Java 1.2. Essa nova convenção de nomes se aplica a todas as edições de Java citadas.

A Figura 13 apresenta o panorama geral de toda a plataforma Java:

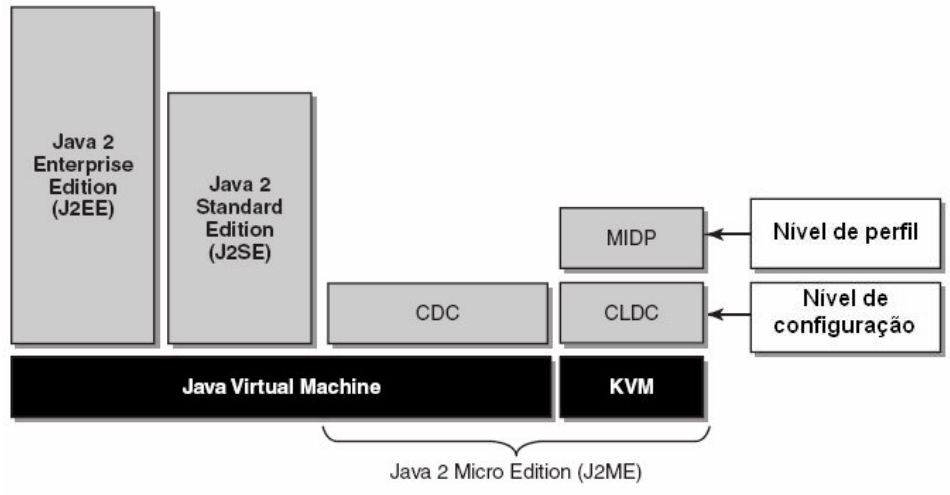

Figura 13: As várias edições de Java [3].

As seções a seguir apresentam os conceitos de configuração e perfil e mostram como a máquina virtual Java está inserida neste contexto.

3.1.2. Configurações

Para suportar a ampla variedade de produtos que se encaixam dentro do escopo de J2ME, a Sun introduziu o conceito de Configuração.

Segundo Muchow [3] "*Uma **Configuração** define uma plataforma Java para uma ampla variedade de dispositivos. Ela está intimamente vinculada a uma máquina virtual Java (JVM, Java Virtual Machine) e define os recursos da linguagem e as bibliotecas básicas da JVM para essa configuração em particular*". Em J2ME utiliza-se a sigla KVM para simbolizar a JVM com um tamanho total que pode ser medido em *kilobytes*.

A linha divisória entre as configurações existentes é, de modo geral, baseada na memória, no vídeo, na conectividade da rede (ou limitações disso) e no poder de processamento. A seguir são apresentadas as duas configurações atualmente disponíveis. Embora a divisão das configurações pareça bastante clara, nem sempre esse é o caso. À medida que a tecnologia oferecer mais recursos, a sobreposição entre essas categorias se tornará maior.

3.1.2.1. Configuração de Dispositivo Conectado (CDC)

Os dispositivos que implementam a CDC utilizam uma arquitetura de 32 bits, têm, no mínimo, dois *megabytes* de memória disponível e implementam uma completa e funcional JVM. Alguns exemplos destes dispositivos são: *set-top boxes*, sistema de navegação e posicionamento, pontos de venda (PDVs) e *smart phones*.

As características a seguir descrevem a CDC:

- 512 *kilobytes* (no mínimo) de memória para executar o Java;
- 256 *kilobytes* (no mínimo) de memória para alocação de memória em tempo de execução; e
- Conectividade de rede, largura de banda possivelmente persistente e alta.

3.1.2.2. Configuração de Dispositivo Conectado Limitado (CLDC)

Os dispositivos que implementam a CLDC utilizam uma arquitetura de 16 ou 32 bits com quantidades reduzidas de memória, geralmente entre 128KB e 512KB, e são alimentados por bateria. Eles também utilizam uma conexão de rede sem fio de banda estreita e podem não apresentar interface com o usuário. Dispositivos implementam uma versão reduzida da máquina virtual Java chamada de KVM (*KJava Virtual Machine*). Alguns exemplos destes dispositivos são: *pagers*, PDAs, telefones celulares e terminais dedicados.

As características a seguir descrevem a CLDC:

- 128 *kilobytes* de memória para executar o Java;
- 32 *kilobytes* de memória para alocação de memória em tempo de execução;
- Interface com o usuário restrita;
- Baixa potência, geralmente alimentado por bateria; e

- Conectividade de rede, normalmente dispositivos sem fio com largura de banda baixa e acesso intermitente.

3.1.3. Perfis

"*Um perfil é uma extensão de uma configuração. Ele oferece bibliotecas para o desenvolvedor escrever aplicativos para um tipo em particular de dispositivo*" [3]. Um exemplo de perfil é o MIDP (*Mobile Information Device Profile*, perfil de dispositivo de informação móvel). Ele é usado em conjunto com a CLDC e define APIs para componentes, entrada e tratamento de eventos de interface com o usuário, armazenamento persistente, interligação em rede e cronômetros, levando em consideração as limitações de tamanho de tela e memória dos dispositivos móveis. Atualmente, a grande maioria deles que têm suporte a Java utilizam o MIDP.

Segundo Keogh [4], pode-se definir ainda outros seis perfis: *Foundation Profile* (Perfil Base), *Game Profile* (Perfil de Jogos), *PDAP – Personal Digital Assistant Profile* (Perfil de PDA), *Personal Profile* (Perfil Pessoal), *Personal Basis Profile* (Perfil Pessoal Básico) e *RMI Profile* (Perfil RMI).

- O ***Foundation Profile*** é usado em conjunto com a CDC (ver Seção 3.1.2.1) e é a base para quase todos os outros perfis que usam CDC, porque este perfil contém as classes Java básicas;
- O ***Game Profile***, como o *Foundation Profile*, é também usado em conjunto com a CDC e contém as classes necessárias para o desenvolvimento de jogos para dispositivos móveis;
- O ***PDAP*** é usado com a CLDC e contém classes que usam os recursos sofisticados dos PDAs. Esses recursos incluem telas melhores e maior memória em comparação com telefones celulares que utilizam MIDP, por exemplo;
- O ***Personal Profile*** é usado em conjunto com a CDC e o *Foundation Profile* e contém classes que implementam uma interface com o usuário complexa, capaz de apresentar várias janelas ao mesmo tempo;
- O ***Personal Basis Profile*** é similar ao *Personal Profile* no que diz respeito à conjunção com a CDC e o *Foundation Profile*. Entretanto, este perfil provê classes que implementam uma interface com o usuário mais simples, capaz de apresentar somente uma janela por vez; e
- O ***RMI Profile*** é usado junto com a CDC e o *Foundation Profile* para prover às classes básicas do *Foundation Profile* a capacidade de chamada remota de métodos (RMI – *Remote Method Invocation*).

Como o objetivo deste trabalho é construir componentes para desenvolvimento de interface gráfica em J2ME para dispositivos móveis e estes implementam o perfil *Mobile*

(MIDP), o resultado apresentado está todo baseado no MIDP. A seção a seguir apresenta em detalhes a arquitetura deste perfil.

3.1.3.1. Arquitetura MIDP

Para facilitar a compreensão da arquitetura MIDP, precisa-se começar com o nível mais baixo, que, no caso, é o hardware. Um nível acima deste, encontra-se o sistema operacional nativo. Alguns aplicativos nativos rodam em cima do sistema operacional nativo, como, por exemplo, o programa que permite a configuração de hora e data do dispositivo. No canto superior direito da Figura 14, encontra-se a referência para um aplicativo nativo.

A CLDC é instalada no sistema operacional e é a base para MIDP. Observa-se que os aplicativos que usam MIDP têm acesso tanto às bibliotecas de CLDC como às de MIDP.

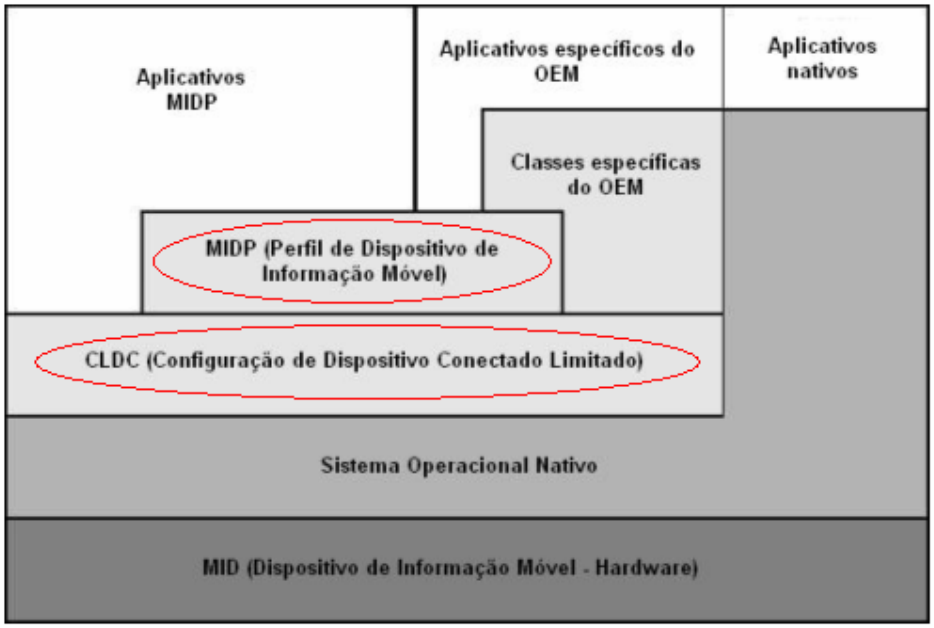

Figura 14: Arquitetura MIDP [3].

As classes específicas de OEM (*Original Equipment Manufacturer*, fabricante original do equipamento) são fornecidas pelo fabricante do dispositivo. Estas classes permitem acesso a funcionalidades específicas do aparelho, tais como: responder às chamadas recebidas ou pesquisar entradas na agenda telefônica. Os aplicativos OEM utilizam as classes OEM em conjunto com algumas classes do MIDP. O grande problema dos aplicativos OEM que utilizam classes específicas do dispositivo é que, por elas serem específicas para o dispositivo, a portabilidade da aplicação é notavelmente reduzida.

A partir da Figura 14, pode-se observar que os aplicativos MIDP têm acesso tanto às bibliotecas do perfil MIDP, como às classes básicas da CLDC. Este trabalho visa construir

componentes para facilitar a construção de interface gráfica com o usuário para este tipo de aplicativo.

3.2. Ambiente de desenvolvimento

Muitos fabricantes de dispositivos móveis fornecem diferentes ferramentas e simuladores de aplicações para desenvolvimento em J2ME, criando assim, ambientes diferentes de acordo com cada fabricante. Entretanto, a Sun fornece um kit de ferramentas gratuito[1] que permite a criação, execução (através de simuladores) e gerenciamento de aplicações J2ME.

3.2.1. MIDlet

"*Um MIDlet é um aplicativo Java projetado para ser executado em um dispositivo móvel*" [3]. Mais especificamente, um *MIDlet* tem como base a CLDC e o MIDP. Um conjunto de *MIDlets* consiste em um ou mais deles empacotados em um arquivo JAR (*Java Archive*). Uma aplicação em J2ME pode ser normalmente constituída por somente um *MIDlet* e todas as outras classes acessadas por ele estariam dentro do JAR. O *MIDlet* comunica-se com o Gerenciador de Aplicativos do dispositivo, responsável por instalar, executar e remover aplicações J2ME. Ele tem a capacidade de informar ao Gerenciador se a aplicação foi finalizada, paralisada, no caso de o dispositivo receber uma chamada telefônica, por exemplo, ou se está sendo solicitada para reativar-se após uma paralisação. Por outro lado, o Gerenciador tem a possibilidade de se comunicar com o *MIDlet* solicitando a sua ativação, paralisação ou finalização.

Um dos métodos mais importantes dentro da classe *MIDlet* é o método *startApp()*, o qual é chamado imediatamente após o início da aplicação. É esse método quem dispara todo o processamento necessário para a execução da aplicação. Ele deve ser escrito com determinado cuidado para não sobrecarregar a carga inicial da mesma (ver Seção 3.3).

A Figura 15 mostra um exemplo com os conceitos apresentados:

[1] Disponível em http://java.sun.com/products/j2mewtoolkit

```
1  import javax.microedition.midlet.MIDlet;
2  public class ExampleMIDlet extends MIDlet {
3      public ExampleMIDlet() {
4          //Construtor público que é chamado somente pelo Gerenciador de
5          //Aplicativos
6      }
7      protected void startApp() {
8          //Este método é chamado quando o Gerenciador de Aplicativos inicia
9          //a aplicação, depois de instanciar o MIDlet chamando seu construtor.
10         //OBSERVAÇÃO: O objeto MIDlet nao pode ser instanciado diretamente.
11     }
12     protected void pauseApp() {
13         //Método chamado pelo Gerenciador de Aplicativos logo após a
14         //suspensão da aplicação devido ao recebimento de uma chamada ou
15         //fechamento do flip no celular, por exemplo.
16     }
17     public void destroyApp(boolean unconditional){
18         //Método chamado quando a aplicação é encerrada.
19     }
20     public void exitApp() {
21         //A boa prática de programação demonstra que o método notifyDestroyed()
22         //deve ser chamado logo após o método destroyApp() é chamado para
23         //informar ao Gerenciador de Aplicativos que a aplicação foi
24         //corretamente finalizada.
25         this.destroyApp(true);
26         this.notifyDestroyed();
27     }
28 }
```

Figura 15: Exemplo de código de um MIDlet.

3.2.2. Arquivo JAR (*Java Archive*)

Uma aplicação Java geralmente consiste de muitos arquivos, tais como, classes, imagens, dados de aplicativos (conhecidos como recursos). Esses arquivos são empacotados em uma única entidade chamada arquivo JAR.

Além desses arquivos, o JAR contém um arquivo conhecido como manifesto. Ele descreve o conteúdo do JAR, tem o nome de `manifest.mf` e é armazenado como parte do próprio JAR. O Gerenciador de Aplicativos do dispositivo, onde está sendo instalada a aplicação J2ME, lê o arquivo manifesto e a partir dele inicia o *MIDlet* corretamente. Se o manifesto não apresentar os atributos essenciais para correta instanciação do *MIDlet*, o Gerenciador se recusa a executar o JAR. A Tabela 1 mostra quais são os possíveis atributos do arquivo manifesto e quais deles são obrigatórios ou não.

ATRIBUTOS DO ARQUIVO MANIFESTO		
Atributo	***Objetivo***	***Exigido***
MIDlet-Name	Nome do conjunto de *MIDlets*	Sim
MIDlet-Version	Número de versão do *MIDlet*	Sim
MIDlet-Vendor	Quem desenvolveu o *MIDlet*	Sim
MIDlet-<n>	Referência a um *MIDlet* dentro de um conjunto de *MIDlets*. Esse atributo contém até três informações (separadas por vírgula): 1. Nome do *MIDlet* 2. Ícone desse *MIDlet* (opcional) 3. Nome da classe que o Gerenciador de Aplicativos chamará para carregar esse *MIDlet*	Sim

MicroEdition-Profile	Qual o perfil de J2ME que é exigido pelo(s) *MIDlet(s)*	Sim
MicroEdition-Configuration	Qual o configuração de J2ME que é exigida pelo(s) *MIDlet(s)*	Sim
MIDlet-Icon	Ícone usado pelo Gerenciador de Aplicativos. Mostrado junto do MIDlet-Name no dispositivo. Esse arquivo deve ser uma imagem no formato PNG	Não
MIDlet-Description	Texto descrevendo o *MIDlet*	Não
MIDlet-Info-URL	URL que pode ter mais informações sobre o *MIDlet* e/ou sobre o fornecedor	Não

Tabela 1: Atributos do arquivo de manifesto [3].

```
1 MIDlet-Name: ExampleMIDlet
2 MIDlet-Version: 2A.01.17
3 MIDlet-Vendor: CIN
4 MIDlet-1: ExampleMIDlet, icon.png, package.ExampleMIDlet
5 MicroEdition-Profile: MIDP-2.0
6 MicroEdition-Configuration: CLDC-1.1
7 MIDlet-Icon: icon.png
8 MIDlet-Description: ExampleMIDlet
```

Figura 16: Exemplo do arquivo manifesto.

3.2.3. Arquivo JAD (Java Application Descriptor)

O arquivo JAD também deve estar disponível como parte do conjunto de *MIDlets* para fornecer informações sobre o(s) *MIDlet(s)* dentro do arquivo JAR. O JAD tem basicamente duas funcionalidades:

- Fornecer informações para o Gerenciador de Aplicativos sobre o conteúdo do arquivo JAR, como por exemplo, o tamanho em *kilobytes* do JAR; e
- Fornecer uma forma eficiente de passar parâmetros para o(s) *MIDlet(s)* da aplicação sem ter que fazer alterações no arquivo JAR. O JAD pode ser acessado a qualquer momento da execução de uma aplicação J2ME e também pode ser usado como fonte para entrada de parâmetros lidos a partir do código.

Vale salientar que o arquivo JAD deve ter a extensão "`.jad`". Uma aplicação somente pode ser instalada em um dispositivo se tiver o arquivo JAR e JAD devidamente estruturados.

A Tabela 2 mostra quais são todos os possíveis atributos do JAD e quais deles são obrigatórios ou não.

ATRIBUTOS DO ARQUIVO JAD (JAVA APPLICATION DESCRIPTOR)		
Atributo	***Objetivo***	***Exigido***
MIDlet-Name	Nome do conjunto de *MIDlets*	Sim
MIDlet-Version	Número de versão do *MIDlet*	Sim
MIDlet-Vendor	Quem desenvolveu o *MIDlet*	Sim
MIDlet-<n>	Ver Tabela 1	Sim

MIDlet-Jar-URL	URL do arquivo JAR	Sim
MIDlet-Jar-Size	Tamanho do arquivo JAR em bytes	Sim
MIDlet-Data-Size	O número mínimo de bytes exigido para armazenamento de dados persistentes	Não
MIDlet-Description	Texto descrevendo o *MIDlet*	Não
MIDlet-Delete-Confirm	Mensagens mostradas para um usuário confirmar um pedido de exclusão da aplicação	Não
MIDlet-Install-Notify	URL para receber relatórios sobre o status da instalação da aplicação	Não

Tabela 2: Atributos do arquivo JAD [3].

```
1 MIDlet-Jar-URL: ExampleMIDlet.jar
2 MIDlet-Jar-Size: 10254
3 MIDlet-Name: ExampleMIDlet
4 MIDlet-Version: 2A.01.17
5 MIDlet-Vendor: CIN
6 MIDlet-1: ExampleMIDlet, icon.png, package.ExampleMIDlet
7 MIDlet-Icon: icon.png
8 MIDlet-Description: ExampleMIDlet
```

Figura 17: Exemplo de um arquivo JAD.

3.2.4. Kit de ferramentas J2ME

Com o propósito de ajudar a gerenciar os projetos em J2ME, a Sun criou o *J2ME Wireless Toolkit* (J2ME WTK) [42]. O *Toolkit* auxilia na hierarquia de projetos com diferentes diretórios para código-fonte, arquivos de classe e de recursos, como, por exemplo, imagens e sons. Ele apresenta uma interface gráfica que permite fazer alterações no arquivo de manifesto e no arquivo JAD. O *Toolkit* também inclui as versões de CLDC e MIDP, recursos suficientes para criação de aplicações em J2ME. A Figura 18 apresenta um exemplo de uma aplicação executada a partir do WTK, que apresenta diferentes modelos como padrão para execução das aplicações.

Figura 18: Exemplo de aplicação executada a partir do WTK [42].

3.3. Boas práticas de programação

Desenvolver aplicações em J2ME exige um pouco mais de cuidado do que em outras plataformas, já que o dispositivo alvo apresenta limitações computacionais. Restrições de memória, por exemplo, muitas vezes não permite que a aplicação nem chegue a ser executada. Devido a este cuidado especial dado às aplicações J2ME, algumas boas práticas de programação podem ser usadas. Segundo Keogh [4], algumas dessas boas práticas são:

- **Manter as aplicações simples e pequenas na medida do possível.** O tamanho de uma aplicação em J2ME é crucial para a sua implantação. Uma boa dica é manter o projeto da aplicação simples e evitar adicionais desnecessários, os quais se tornaram um costume constante em aplicações para *desktops*;
- **Limitar o uso de memória.** Três dicas podem ser consideradas para limitar o uso de memória na aplicação: usar tipos escalares, como inteiro e *boolean*, ao invés de usar tipos objetos; otimizar o uso do *Garbage Collection*[1], somente usando os objetos estritamente necessários e logo, após o uso, colocar as referências deles para *null*; sempre reutilizar objetos ao invés de criar novos;
- **Evitar computações pesadas.** Uma boa forma de se evitar computações pesadas é fazer uma comunicação com um servidor que desempenhe este

[1] *Garbage Collection*: é uma forma de desocupar a memória que não está sendo mais utilizada pela aplicação, mas que já tinha sido alocada anteriormente.

papel. Numa aplicação de informações sobre o trânsito numa cidade, por exemplo, um servidor poderia fornecer todo o status do trânsito e aplicação somente seria responsável por apresentá-lo;

- **Gerenciar conexões de rede, minimizando o tráfego de pacotes.** Num dispositivo móvel, não se pode ter certeza da disponibilidade da rede, ou seja, a comunicação pode ser quebrada a qualquer momento. Para evitar que este fato tenha impacto direto no usuário, uma boa prática é manter as transmissões de dados pequenas, abrindo o menor número de conexões possível, já que, abrir uma conexão de dados é consideravelmente custoso para um dispositivo com recursos reduzidos;
- **Simplificar interface com o usuário.** É importante lembrar que a forma de interação do usuário com o dispositivo móvel é consideravelmente limitada em relação a aplicações *desktop*, que mantêm um número grande de componentes gráficos para construção de interfaces e permite interação de várias formas, como teclado, mouse ou tela de toque, por exemplo. Deixar o visual da aplicação J2ME simples se torna crucial para manter o usuário interessado em usá-la;
- **Evitar usar variáveis globais, sincronização e concatenar *Strings*.** Estas práticas diminuem o processamento e memória exigidos na aplicação; e
- **Ter precauções com o conteúdo do método *startApp()*.** Como foi citado na Seção 3.2.1, um *MIDlet* tem o método *startApp()* que é chamado imediatamente após o início da execução da aplicação e também após a reativação da aplicação depois de uma pausa. Logo, se este método contiver muitas linhas de código ou computações pesadas, a aplicação irá demorar a ser iniciada. Uma boa prática é escrever o código estritamente necessário para execução sem erros da aplicação e, à medida do que for preciso, realizar chamadas a outros métodos sob demanda.

A experiência e convivência na área de programação em J2ME também estimulam a utilização de outras boas práticas, tais como:

- **Evitar pinturas desnecessárias da tela.** Quando se desenvolve interface com o usuário para dispositivos móveis em J2ME, o programador pode controlar a pintura de objetos na tela. Em algumas ocasiões de interação do usuário com a aplicação, a tela precisa repintada. Porém, em muitas dessas ocasiões, não existe a necessidade de repintar a tela inteira, mas sim somente a parte que foi atualizada após a interação. Ou seja, quanto menos a tela for repintada, menor será o processo computacional a ser realizado pelo dispositivo; e
- **Utilizar memória persistente do dispositivo com cautela.** Os dispositivos móveis apresentam um espaço reservado para aplicações J2ME armazenarem

informações persistentes. Entretanto, esse espaço é compartilhado entre todas as aplicações instaladas naquele dispositivo e tem uma capacidade de armazenamento consideravelmente pequena. Logo, a aplicação deve apresentar um bom comportamento em situações onde não há mais espaço de armazenamento. Neste caso, um bom comportamento seria simplesmente não travar ou apresentar uma mensagem de erro compatível.

3.4. Construindo uma aplicação J2ME

Nesta seção, serão apresentados os conceitos básicos para a criação de execução de uma aplicação J2ME, incluindo detalhes sobre a classe *MIDlet* e o objeto *Display*, além de como são tratados os eventos da interface gráfica no dispositivo móvel e qual a hierarquia geral das classes do MIDP.

3.4.1. Objeto *MIDlet*

O objeto *MIDlet*, como já foi citado na Seção 3.2.1, é a principal classe de uma aplicação J2ME. É ele que mantém a comunicação com o Gerenciador de Aplicativos do dispositivo. Este objeto pode passar por três estados durante o ciclo de vida da aplicação.

- **Ativo**: o *MIDlet* está em execução;
- **Pausado**: o *MIDlet* foi colocado em estado de espera pelo Gerenciador de Aplicativos, após ter sido instanciado e iniciado. Ele pode alternar entre o estado de Ativo e Pausado qualquer número de vezes; e
- **Destruído**: o *MIDlet* libera todos os recursos que lhe foram alocados para a sua execução e assim é desligado.

Para criar um objeto *MIDlet*, é necessário fazer com que a classe em desenvolvimento herde da classe *MIDlet* da API nativa de J2ME no pacote `javax.microedition.midlet.MIDlet`. Os métodos da classe são:

- **`abstract void destroyApp(boolean unconditional)`**: o Gerenciador de Aplicativos solicita o desligamento do *MIDlet*;
- **`abstract void pauseApp()`**: o Gerenciador de Aplicativos solicita a pausa do *MIDlet*;
- **`abstract void startApp()`**: o Gerenciador de Aplicativos solicita a ativação do *MIDlet*;
- **`final void notifyDestroyed()`**: o *MIDlet* solicita para ser finalizado;
- **`final void notifyPaused()`**: o *MIDlet* solicitado para ser pausado;
- **`final void resumeRequest()`**: o *MIDlet* solicita para se tornar ativo após uma pausa; e

- **`final String getAppProperty(String key)`**: obtém atributos dos arquivos JAR e/ou JAD a partir da chave passada como parâmetro.

Os métodos abstratos (*abstract*) necessitam obrigatoriamente que sejam implementados quando se está construindo um novo *MIDlet*. Esses métodos são base da comunicação entre o Gerenciador de Aplicativos e a aplicação J2ME.

3.4.2. Objeto Display

Este objeto é usado para controlar o que está sendo mostrado na tela do dispositivo. Seus métodos farão mais sentido após a apresentação da classe `Displayable` (ver Seção 3.4.2.1), da qual se originam todas as classes a serem usadas na interface gráfica. Resumidamente, o objeto `Display` é obtido através do *MIDlet* (somente um `Display` por *MIDlet*) e contém métodos que configuram qual objeto `Displayable` estará sendo mostrado na tela, podendo ser mostrados mais de um deles. Os métodos da classe `Display` do pacote `javax.microedition.lcdui.Display` são:

- **`static Display getDisplay(MIDlet m)`**: obtém o objeto `Display` do *MIDlet* passado como parâmetro;
- **`Displayable getCurrent()`**: obtém o objeto `Displayable` corrente;
- **`void setCurrent(Alert alert, Displayable nextDisplayable)`**: mostra um alerta seguido de um próximo objeto `Displayable`;
- **`void setCurrent(Displayable nextDisplayable)`**: apresenta na tela o objeto passado como parâmetro;
- **`boolean isColor()`**: informa se o dispositivo suporta cores;
- **`int numColors()`**: informa a quantidade de cores suportadas pelo dispositivo; e
- **`void callSerially(Runnable r)`**: pede para que um objeto executável seja chamado após a repintura.

3.4.2.1. Objeto `Displayable`

Como o nome já descreve, este objeto é tudo aquilo que pode ser apresentado na tela do dispositivo. É uma classe abstrata que, no MIDP, serve como base para duas subclasses: `Screen` e `Canvas`. Segundo Muchow [3], os objetos da subclasse `Screen` (`TextBox`, `List`, `Form`, `Alert`) são todos componentes de alto nível da interface com o usuário, pois sua implementação e apresentação no dispositivo são manipuladas pelo desenvolvedor. Já o objeto `Canvas` é usado para elementos gráficos personalizados, tratamento de eventos de baixo nível e exige um pouco mais de cuidado da parte do desenvolvedor para atualizar a tela. A Seção 3.5 apresenta detalhes sobre essas duas

subclasses. Neste trabalho, o foco maior será dado à classe `Canvas` já que os componentes desenvolvidos têm como base esta classe.

A classe `Displayable` apresenta quatro métodos:

- **`void addCommand(Command cmd)`**: adiciona `Command` n;o objeto `Displayable`;
- **`boolean removeCommand(Command cmd)`**: remove o objeto `Command` do `Displayable`;
- **`void setCommandListener(CommandListener l)`**: adiciona `CommandListener` ao objeto `Displayable`; e
- **`boolean isShown()`**: informa se o objeto `Displayable` está visível na tela.

Mais detalhes sobre o uso das classes `Command` e `CommandListener`, podem ser encontrados na seção seguinte.

3.4.3. Tratamento de eventos

Ao passo que existe algum tipo de interação com o usuário, o processamento de eventos é essencial para quase todo *MIDlet*. "*O tratamento de eventos nada mais é do que reconhecer quando um evento ocorre e adotar uma ação baseada nesse evento. Por exemplo, reconhecer que um botão de ajuda foi pressionando e exibir uma mensagem pertinente*." [3].

Basicamente, duas classes são levadas em consideração quando se deseja fazer o tratamento de eventos utilizando a API nativa de J2ME: `Command` e `CommandListener`.

O objeto `Command` contém informações sobre um evento. Uma boa associação a ser feita é pensar neste objeto como sendo um "botão". O `CommandListener` é uma interface que serve como receptor para os `Commands`. Para permitir o processamento de eventos na classe que está sendo desenvolvida, é necessário que seguir alguns passos:

1. Criar um objeto `Command` para encapsular as informações sobre um evento (Figura 19 – linha 13).
2. Adicionar esse objeto em outro objeto que `Form`, `TextBox`, `List` ou `Canvas` (Figura 19 – linha 17).
3. Adicionar um receptor no objeto `Form`, `TextBox`, `List` ou `Canvas` (Figura 19 – linha 18).

O `CommandListener` apresenta um método cuja assinatura é "`void commandAction(Command c, Displayable d)`" e é chamado quando o objeto "`Command c`" no objeto "`Displayable d`" inicia um evento (Figura 19 – linhas 29 à 34).

```
1  import javax.microedition.lcdui.*;
2  import javax.microedition.midlet.*;
3  public class AccessingCommands extends MIDlet implements CommandListener {
4      private Display myDisplay;
5      private Form myForm;
6      private Command exitCommand;
7      public AccessingCommands(){
8          //Inicializando o objeto Display passando como referência este MIDlet
9          myDisplay = Display.getDisplay(this);
10
11         //Inicializando o exitCommand com label "EXIT", o tipo de Command.EXIT
12         //e a prioridade de 1 (Canto esquerdo da tela é de prioridade 1).
13         exitCommand = new Command("EXIT", Command.EXIT, 1);
14
15         //Criando um Form que irá receber o Command criado anteriormente
16         myForm = new Form("My Form");
17         myForm.addCommand(exitCommand); //Adicionando o Command ao Form
18         myForm.setCommandListener(this);//Adicionando um receptor (Listener) ao Form
19     }
20     protected void startApp() {
21         myDisplay.setCurrent(myForm);   //Apresentando o Form na tela
22     }
23     protected void pauseApp() { }
24     protected void destroyApp(boolean arg0) { }
25     /**
26      * Este método trata as ações de comando recendo o comando que está sendo
27      * executado e o objeto Displayable de onde o comando é acionado
28      */
29     public void commandAction(Command c, Displayable d) {
30         if (c == exitCommand) {
31             destroyApp(false);
32             notifyDestroyed();
33         }
34     }
35 }
```

Figura 19: Exemplo de código para tratamento de eventos.

3.4.4. Hierarquia de classes MIDP

Para se ter uma visão geral de como estão estruturadas as classes no perfil MIDP, a Figura 20 apresenta um Diagrama de Classes com notação UML que agrega as principais classes responsáveis pela criação de interfaces gráficas.

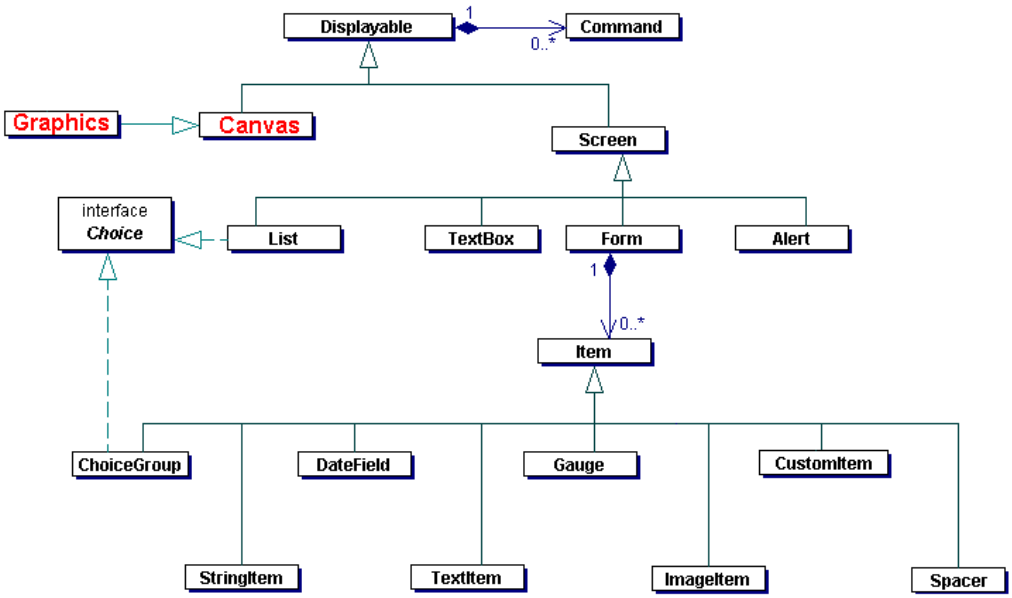

Figura 20: Hierarquia de classes MIDP [3].

Através da Figura 20, pode-se observar que existem basicamente duas formas de se construir interface em J2ME. Uma delas é utilizando a classe `Screen` e suas subclasses. A outra forma é utilizando a classe `Canvas` e os métodos da classe `Graphics`, com suas primitivas gráficas. A seção a seguir apresenta mais detalhes sobre essas duas formas de construção de interface gráfica.

3.5. Construindo interface com o usuário em J2ME

O MIDP dispõe de duas possibilidades de construção de interface gráfica com o usuário:

i. ***Screen***: essa é a forma de mais alto nível para construção de interface gráfica, onde a classe `Screen` é a "classe mãe" de todas as outras. Utilizar esta classe implica em utilizar um conjunto limitado de componentes, originalmente presente no MIDP, onde o desenvolvedor não irá se preocupar em saber como o componente está sendo desenhado na tela. Assim sendo, cada componente teria sua função específica e não poderia ser usado de forma mais customizada para uma aplicação qualquer; e

ii. ***Canvas***: essa é a forma de mais baixo nível para desenvolvimento de interface gráfica onde a classe `Canvas` é a classe base para toda a programação. Esta classe existe em conjunto com a classe `Graphics`, a qual permite a pintura na tela de primitivas gráficas tais como reta, quadrado e círculo. Utilizando esta forma de construção, o desenvolvedor deve se preocupar com todos os detalhes da

interface pixel a pixel. Isso faz com o esforço seja maior, porém o resultado é também de considerável qualidade.

3.5.1. Interface de alto nível: *Screen*

Como este trabalho se propõe a desenvolver componentes gráficos personalizados que se baseiam em desenhar a tela pixel a pixel em um nível maior de detalhes, com a intenção de se obter uma interface com o usuário mais rica, a classe `Canvas` é a mais apropriada para o proposto. Logo, esta seção não entrará em muitos detalhes sobre a classe `Screen`.

A classe `Screen` apresenta um conjunto de subclasses suficientes para produzir rapidamente uma aplicação J2ME simples. A partir da Figura 21, pode-se observar a hierarquia de classes:

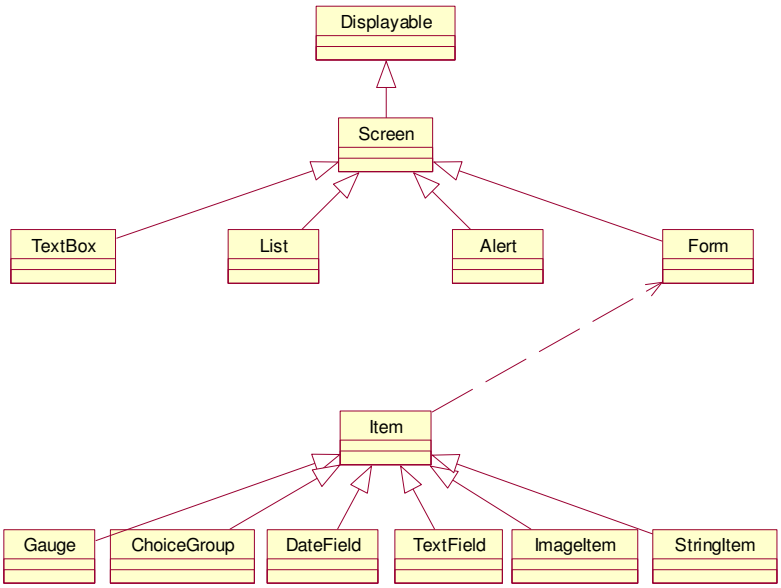

Figura 21: Hierarquia de classes para `Screen` [3].

- **`Screen`**: a classe em si não é algo que possa ser visto. Ela serve como mãe para os outros componentes e têm uma aparência e um comportamento na tela;
- **`TextBox`**: caixa de entrada de texto que pode conter várias linhas e permite filtrar a entrada do usuário para somente texto ou somente números, por exemplo. Este componente pode receber uma barra de rolagem se a quantidade de caracteres dados como entrada excederem o tamanho da tela. Este componente herda diretamente de `Screen` e por isso pode ser apresentado na tela sem nenhuma outra dependência;
- **`List`**: é usada para mostrar uma lista de itens na tela e permite a seleção desse grupo de itens. Este componente apresenta a mesma funcionalidade do

ChoiceGroup, porém, esse último herda de Item e List herda de Screen. Ou seja, assim como TextBox, List pode ser apresentada na tela sem nenhuma outra dependência;

- **Alert**: representa uma caixa de diálogo mostrada na tela para alertar o usuário sobre um possível erro na aplicação ou simplesmente para transmitir uma mensagem. Esta mensagem apresentada pode ser intermitente, onde um comando do usuário é esperado para sair da tela, ou temporária, com um tempo predefinido para ser apresentada;
- **Form**: este objeto é um contêiner que pode ter qualquer número de componentes subclasses de Item. Esta implementação permite a exibição de vários itens na tela e fornece rolagem quando for necessário acomodar os componentes. Apresenta métodos para anexar, substituir, inserir e remover componentes;
- **Item**: é um componente que pode ser adicionado num objeto Form São subclasses de um de Item: ChoiceGroup, DateField, Gauge, ImageItem, StringItem e TextField;
- **ChoiceGroup**: esta classe implementa a interface Choice, a qual define métodos relacionados a manipulação de vários tipos de seleções predefinidas. O ChoiceGroup representa um grupo de seleção que pode ser múltiplo ou exclusivo. Este componente é semelhante a *RadioButtons* e *CheckButtons* numa aplicação *desktop*;
- **DateField**: este componente permite manipular um objeto Date (conforme definido em java.util.Date), através das telas e/ou botões de software de um dispositivo móvel;
- **TextField**: apresenta exatamente a mesma funcionalidade do TextBox, porém este componente deve estar contido num Form, justamente por herdar da classe Item, não podendo ser mostrado sem estar atrelado a um Form;
- **Gauge**: permite montar um medidor de progresso semelhante a um indicador de porcentagem exibido durante um *download*;
- **ImageItem**: permite encapsular a maneira como se quer que uma imagem, originada da classe javax.microedtion.lcdui.Image seja apresentada na tela; e
- **StringItem**: apresenta um rótulo estático que não pode ser editado com uma mensagem de texto.

3.5.2. Interface de baixo nível: *Canvas*

A seção anterior apresentou os componentes básicos para criação de interface gráfica com o usuário. Esses componentes liberam o desenvolvedor de pensar como a tela está sendo desenhada. Entretanto, poder desenhar na tela aumenta consideravelmente as possibilidades de criação de interface e é essa a principal capacidade da classe `Canvas`, utilizada em conjunto com a classe `Graphics`.

A classe `Canvas` forma a tela de fundo do dispositivo com altura e largura determinadas e nela é desenhado o que o usuário final verá. Ela também apresenta uma forma de mais baixo nível para tratamento de eventos, um pouco diferente da citada na Seção 3.4.3. Para desenhar o que se deseja na tela, representada pela classe `Canvas`, utiliza-se a classe `Graphics`. Esta contém métodos responsáveis por desenhar linhas, retângulos e texto, assim como, também permite especificar cor e preferências de fonte.

De acordo com Muchow [3], o primeiro passo para a criação de interfaces personalizadas é entender a classe `Canvas`. Esta classe é abstrata e é subclasse de `Displayable` (ver Seção 3.4.2.1). Ou seja, para criar uma instância da classe `Canvas`, é necessário criar uma subclasse dela que implemente o seu método abstrato `protected void paint(Graphics g)`. Este método é automaticamente chamado pela máquina virtual Java quando se executa o método `Display.setCurrent(Displayable d)`. A Figura 22 apresenta um trecho de código para facilitar o entendimento da classe:

```
1  import javax.microedition.lcdui.Canvas;
2  import javax.microedition.lcdui.Graphics;
3  public class MyCanvas extends Canvas{
4      protected void paint(Graphics arg0) {
5          //Escrever o corpo do método especificando o que deve ser
6          //desenhado na tela. O método paint não é chamado diretamente
7          //pelo desenvolvedor. Ele é chamado pelo KVM a partir do momento
8          //que o método repaint() é chamado. Sendo assim, a KVM se
9          //responsabiliza por passar como parâmetro uma instância de Graphics
10     }
11 }
12 //Para mostrar esta classe na tela deve-se executar os seguintes
13 //passos:
14 MyCanvas myCanvas = new MyCanvas();
15 //...
16 display.setCurrent(myCanvas);
```

Figura 22: Exemplo de código utilizando a classe `Canvas`.

É importante observar que o objeto `Graphics` não é instanciado explicitamente. Ele é passado como parâmetro pela própria máquina virtual quando o método `paint` é chamado.

As seções a seguir apresentam mais detalhes de como funciona o sistema de coordenadas na classe `Canvas`, como é feito o tratamento de eventos de baixo nível e mais detalhes sobre a API da mesma.

3.5.2.1. Sistema de coordenadas

Para facilitar o posicionamento na tela, a API do MIDP define um sistema de coordenadas `x,y` cuja origem se localiza no canto superior esquerdo da tela. Cada ponto no sistema de coordenadas representa um pixel desenhado na tela. Os valores de `x` aumentam para a direita e os valores de `y` aumentam para baixo. Na maioria dos projetos J2ME, esse sistema pode diminuir a portabilidade de uma aplicação já que os tamanhos de tela e definições de cores entre os diferentes dispositivos não são os mesmos. A classe `Canvas` apresenta métodos que permitem capturar a altura e largura total do dispositivo, deixando assim a responsabilidade com o desenvolvedor no cuidado com os limites da tela.

A Figura 23 apresenta um exemplo do uso do sistema de coordenadas.

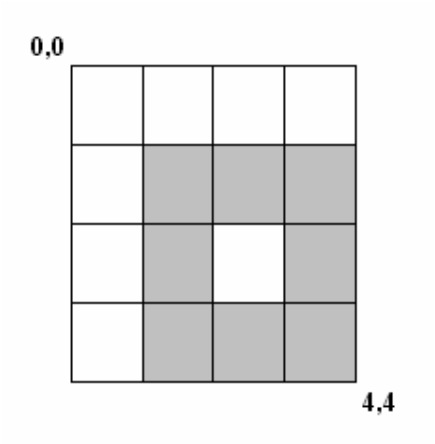

Figura 23: Exemplo de um retângulo desenhado do ponto (1,1) ao ponto (4,4).

3.5.2.2. Tratamento de eventos na classe *Canvas*

O tratamento de eventos desta classe é considerado de mais baixo nível porque utiliza códigos com números inteiros para cada tecla que tenha sido pressionada. Esses valores numéricos podem variar entre dispositivos diferentes e afeta na portabilidade da aplicação. Existem quatro métodos na classe `Canvas` que tratam esses eventos. Em todos esses métodos, a máquina virtual se encarrega de passar como parâmetro o código da tecla envolvida no evento.

- **`void keyPressed(int keyCode)`**: método disparado quando uma tecla é apertada;
- **`void keyReleased(int keyCode)`**: método disparado quando uma tecla é liberada;

- **`void keyRepeated(int keyCode)`**: método disparado quando uma tecla é apertada repetidamente. Este método pode não ser suportado em todos os dispositivos; e
- **`String getKeyName(int keyCode)`**: retorna a String de texto que representa o código da tecla.

3.5.2.3. API *Canvas*

Alguns outros métodos são importantes de serem observados, tais como:

- **`final void repaint()`**: solicita que a tela de desenho seja redesenhada. Este método é chamado quando se deseja redesenhar a tela não havendo necessidade de passar como parâmetro o objeto `Graphics`;
- **`final void repaint(int x, int y, int width, int height)`**: solicita que uma região específica da tela, determinada pelos parâmetros, seja redesenhada;
- **`final void ServiceRepaints()`**: processa todos os pedidos de pintura ainda pendentes;
- **`void showNotify()`**: método disparado quando uma tela é apresentada no vídeo;
- **`void hideNotify()`**: método disparado quando uma tela é removida do vídeo;
- **`int getKeyCode(int gameAction)`**: retorna o código de tecla de uma ação de jogo[1]; e
- **`int getGameAction(int keyCode)`**: retorna a ação de jogo, se houver, de um código de tecla.

[1] Ação de jogo: conjunto de constantes para facilitar o tratamento de eventos relacionados a jogos, tais como, mover para cima, baixo, direita ou esquerda. Essas constantes podem ser normalmente usadas em aplicações comuns que não sejam jogos.

3.6. Resumo

Este capítulo apresentou um panorama geral da tecnologia J2ME e como ela está estruturada em perfis e configurações. Também foi abordado o ambiente de desenvolvimento em J2ME e qual o significado dos arquivos JAR e JAD, além de algumas boas práticas de programação em J2ME. Conceitos básicos do objeto *MIDlet*, do objeto *Display* e tratamento de eventos são explicados, disponibilizando o conhecimento mínimo para o desenvolvimento de uma aplicação simples.

Por fim, foram apresentados os conceitos básicos de montagem de interface gráfica utilizando as duas formas possíveis: com a classe `Screen` e suas subclasses e com a classe `Canvas`, explicando as formas específicas de tratamento de eventos de baixo nível desta classe.

O próximo capítulo apresentará o estudo de caso com uma implementação de componentes para construção de interface gráfica em J2ME, utilizando como base a classe `Canvas` e tendo como método de Desenvolvimento Baseado em Componentes o *UML Components*.

Capítulo 4

Estudo de caso

Este capítulo apresenta em detalhes a aplicação do método *UML Components*, descrito na Seção 2.7, no **desenvolvimento de um *framework*** para criação de interface gráfica com o usuário em dispositivos móveis utilizando J2ME.

O capítulo está organizado do seguinte modo: na Seção 4.1, é elaborado um projeto inicial do estudo de caso e, em seguida, a Seção 4.2 apresenta a aplicação de *UML Components* na criação do *framework*. A Seção 4.5 mostra exemplos de aplicações em J2ME que utilizam o *framework*. Por fim, a Seção 4.6 faz uma análise do estudo de caso, ressaltando os pontos fortes e fracos, as limitações existentes e as lições aprendidas.

4.1. Projeto inicial do estudo de caso

Visando atingir o objetivo deste trabalho de construir um *framework* para desenvolvimento de interface gráfica em J2ME, é necessário levantar quais são os requisitos aplicáveis a este projeto. Analisando-se as interfaces gráficas de aplicações em J2ME, observou-se que o *framework* deve conter componentes que representem:

- Uma **tela padrão adaptável**. Este componente montará uma tela no dispositivo com os seguintes itens: barra de título, com texto e ícone opcional ao lado do texto; espaço no centro da tela para preenchimento com qualquer item a ser apresentado; botões de comando no canto inferior esquerdo e direito; botão de menu de contexto entre os dois botões de comando. Todas as outras entidades do *framework* que utilizam como área de pintura a tela inteira deverão utilizar este padrão de apresentação. A Figura 24 mostra dois exemplos desta tela;

Figura 24: Exemplos de uma tela padrão adaptável.

- Uma **lista de itens**. Este componente monta uma tela que apresenta uma lista de itens com uma barra de rolagem. Esta lista pode ser, por exemplo, um conjunto de arquivos ordenados por ordem alfabética com seus respectivos tamanhos e uma possível imagem em miniatura (*thumbnail*) associada. Ao interagir com a lista, deve ser possível ao usuário visualizar qual item está atualmente selecionado. A Figura 25 mostra dois exemplos de uma lista de itens;

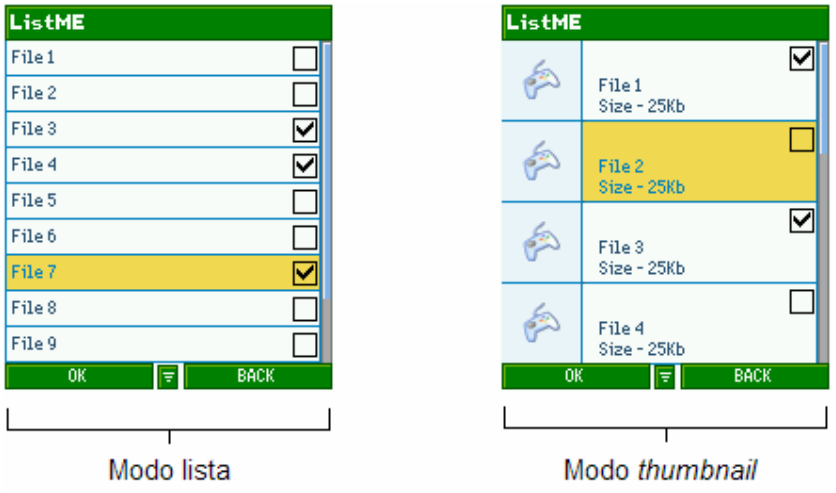

Figura 25: Exemplos de uma lista de itens.

- Um **menu de itens**. Esta entidade é responsável por apresentar um menu de opções ao usuário. As opções podem ser visualizadas de duas formas distintas: uma lista pequena, sem barra de rolagem, com cada uma das opções; um grupo de ícones organizados dois a dois na tela. Nas duas opções, é possível visualizar qual item está selecionado. Esta tela se distingue um pouco da tela padrão adaptável por não apresentar menu de contexto. A Figura 26 apresenta dois exemplos de um menu de itens;

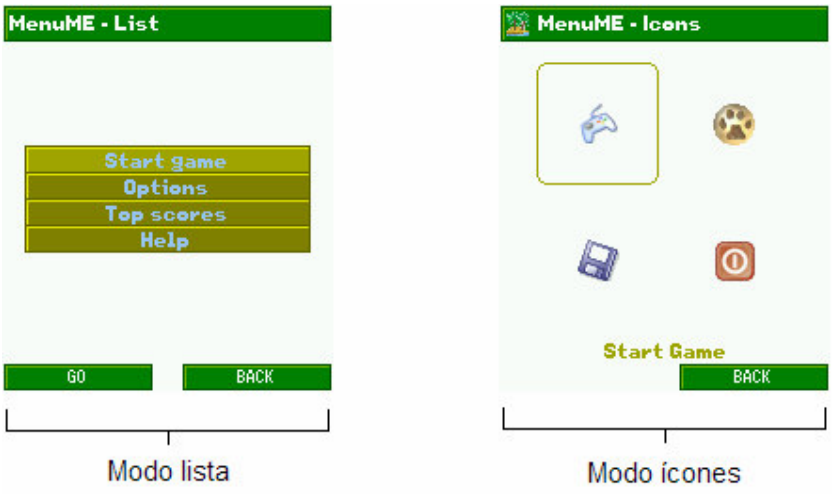

Figura 26: Exemplos de um menu de itens.

- Um **formulário padronizado**. Este componente agrega um conjunto de itens padronizados que podem ser encontrados em formulários digitais, tais como: campo de texto, área de texto, imagens e texto indicador (*label*). A Figura 27 apresenta um exemplo de formulário padronizado;

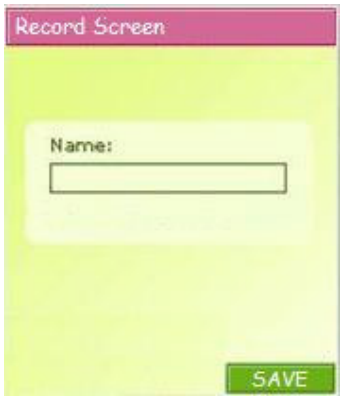

Figura 27: Exemplo de formulário padronizado.

- Uma **mensagem na tela**. Este componente é uma extensão da tela padrão adaptável que escreve no espaço do centro da tela uma mensagem com barra de rolagem e no canto inferior direito um botão de comando. Este botão tem como ação padrão mostrar a tela anteriormente apresentada. Esta entidade poderia ser utilizada para mostrar mensagens ao usuário, como mensagens de erro ou aviso. A Figura 28 mostra um exemplo de uma mensagem na tela; e

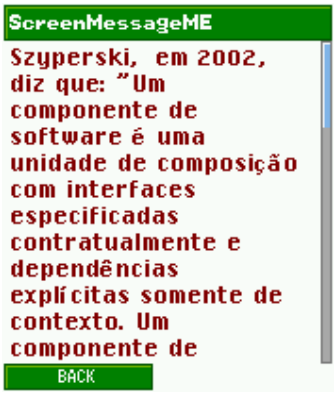

Figura 28: Exemplo de mensagem na tela.

- Um **conjunto de propriedades**, que encapsula as propriedades a serem lidas pelos outros componentes, deixando as telas mais facilmente adaptáveis sem alterações de código. Por exemplo, a mudança de cores do aplicativo pode ser feita somente pelo arquivo de propriedades.

4.1.1. Escopo do projeto

Criar componentes no domínio de construção de interface gráfica em J2ME pode não ser fácil devido ao caráter peculiar que cada interface apresenta. Levando isso em consideração o escopo do projeto terá algumas restrições:

- **O *framework* não será aplicável a jogos** – os componentes desenvolvidos não terão aplicação direta em jogos na plataforma J2ME;
- **O formulário padronizado somente aceitará campo de texto, área de texto, imagens e texto indicador (*label*)** – formulários em J2ME podem apresentar vários componentes personalizados. A API original de Java já provê a classe `javax.microedition.lcdui.CustomItem` para simbolizar um componente adaptado a ser adicionado no formulário. Assim sendo, para não deixar o *framework* com um escopo grande e difícil de ser implementado no tempo disponível, o formulário somente aceitará os componentes citados; e
- **A barra de rolagem terá tamanho fixo** – para telas que precisam apresentar barra de rolagem, ela terá o tamanho fixo baseado no espaço livre disponível no centro da tela. Esse espaço livre considerado será o tamanho total da tela menos o espaço inferior para os botões de comando e o espaço superior para a barra de título. A Figura 28 pode demonstrar o que foi citado.

A partir dos requisitos e do escopo mencionados, a seção seguinte apresenta como o método *UML Components* foi utilizado na construção do *framework*.

4.2. Aplicação do *UML Components* no estudo de caso

Para utilização do método *UML Components* na construção do *framework*, é necessário que algumas adaptações sejam feitas para que os passos a serem seguidos durante o todo o processo tenham significado aplicável dentro do contexto de construção de interface gráfica em J2ME. Algumas adaptações sugeridas são:

i. **Considerar a associação entre o tipo** `<<core>>` **e o estereótipo** `<<interface type>>` **para indicar um contrato de realização**. Durante o desenvolvimento de interface gráfica em J2ME, é indispensável fazer o tratamento de eventos disparados pelo usuário. Desta forma, um componente que vai ser utilizado para construção de diferentes interfaces gráficas não pode prever como será feito o tratamento do evento disparado pelo usuário. Logo, este componente deverá estabelecer um contrato de realização com a outra parte que irá utilizá-lo. Para melhor entender o funcionamento do contrato de realização, pode-se citar como exemplo a classe `Canvas` (ver Seção 3.5.2), que disponibiliza os métodos `keyPressed` e `keyReleased` para tratar eventos de quando um usuário pressiona ou solta uma tecla do dispositivo. O corpo desses métodos deve ser implementado pela classe que herda de `Canvas` ou pela classe da aplicação que herdará dos componentes do *framework*. Portanto, para as entidades do *framework* que apresentarem o estereótipo `<<core>>` associado ao estereótipo `<<interface type>>`, subentende-se que devem manter um contrato de realização por aqueles componentes que a utilizam; e

ii. **Considerar o tipo** `<<core>>` **que não apresenta associação com um estereótipo** `<<interface type>>` **para indicar um contrato de uso**. Isto significa que as entidades que têm o tipo `<<core>>` sem estar associadas a um estereótipo `<<interface type>>` podem ser usadas diretamente por outras partes, as quais não precisam estender o comportamento do componente.

Outras adaptações serão sugeridas dentro de cada passo do processo, descritos nas seções seguintes.

4.2.1. Identificando componentes no *framework*

Seguindo o processo apresentado na Seção 2.8.1, os passos a serem seguidos são: identificar interfaces e operações do sistema; identificar interfaces de negócio; criar especificação inicial de interfaces; especificar arquitetura de componentes. Dentro do contexto de criação do *framework*, esses passos precisam ser submetidos a algumas adaptações.

Considerando o contexto de aplicações em J2ME, o passo de identificação de interfaces e operações do sistema é executado a partir da descrição dos casos de uso, os

quais são analisados passo a passo, facilitando na descoberta das operações. Entretanto, no contexto do estudo de caso, este passo pode ser descartado, já que não existem casos de uso descritos. Logo, o passo de identificação de interfaces de negócio passa a ser o primeiro a ser executado na fase de identificação de componentes do *framework*.

Para identificar interfaces de negócio, é necessário partir de um modelo conceitual do negócio e, submetendo-o a refinamentos, obter o modelo de tipos de negócio. A Figura 29 apresenta o modelo conceitual do *framework* tendo como base os requisitos levantados na Seção 4.1. As entidades receberam o nome com a extensão *ME* simbolizando a sigla *Micro Edition*.

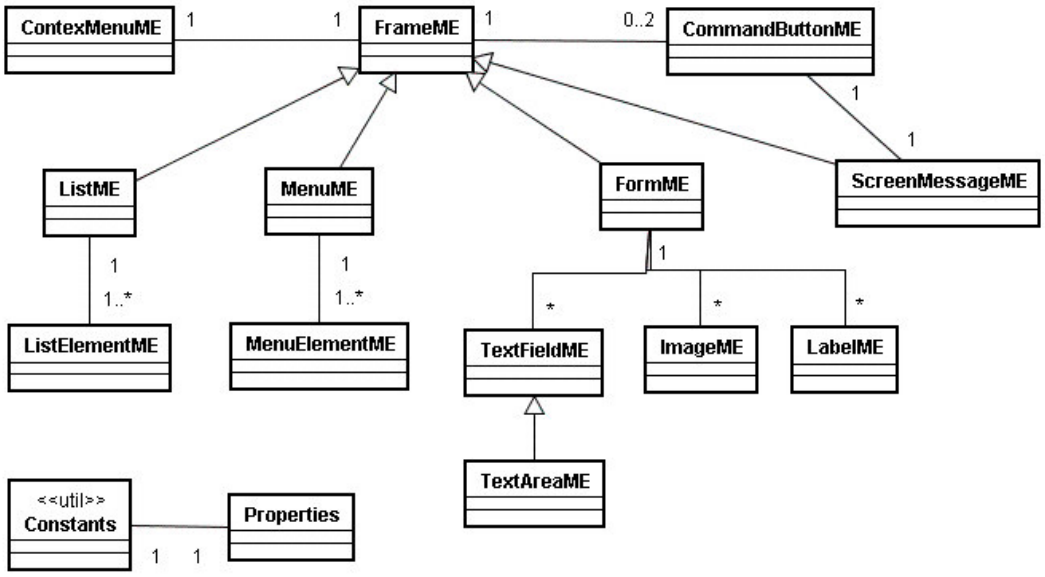

Figura 29: Modelo conceitual do negócio.

Este modelo conceitual atende às necessidades levantadas na definição de requisitos utilizando a seguinte representação:

- **`FrameME`** – tela padrão adaptável com dois botões de comando (**`CommandButtonME`**) e um menu de contexto (**`ContextMenuME`**);
- **`ListME`** – lista de itens que podem ser apresentados na tela. Estes itens são representados pela entidade **`ListElementME`**;
- **`MenuME`** – menu de itens que pode ser apresentados na tela na forma de lista organizados dois a dois. Estes itens são representados pela entidade **`MenuElementME`**;
- **`FormME`** – formulário padronizado que pode receber itens do tipo **`TextFieldME`**, **`TextAreaME`**, **`ImageME`** e **`LabelME`**;

- **`ScreenMessageME`** – mensagem apresentada na tela para o usuário e somente apresenta um botão de comando; e
- **`Constants`** e **`Properties`** – permitem encapsular as propriedades a serem usadas por qualquer outro componente do *framework*.

A partir do modelo conceitual, é possível encontrar o modelo de tipos de negócio, o qual vai servir de guia para as próximas fases. Este modelo, descrito na Figura 30, apresenta alguns refinamentos do modelo conceitual.

Esses refinamentos constituem-se de:

- Criação do tipo **`Scrollable`** para definir um item que pode ser adicionado a um `FormME`. Os tipos `LabelME`, `ImageME` e `TextFieldME` herdariam diretamente de `Scrollable`; e
- Adição das relações entre: `MenuElementME` e `ImageME`, já que cada `MenuElementME` apresentará um ícone com uma imagem a ser rolada na tela; `ContextMenuME` e `CommandButtonME`, pois o menu de contexto também conterá botões de comando.

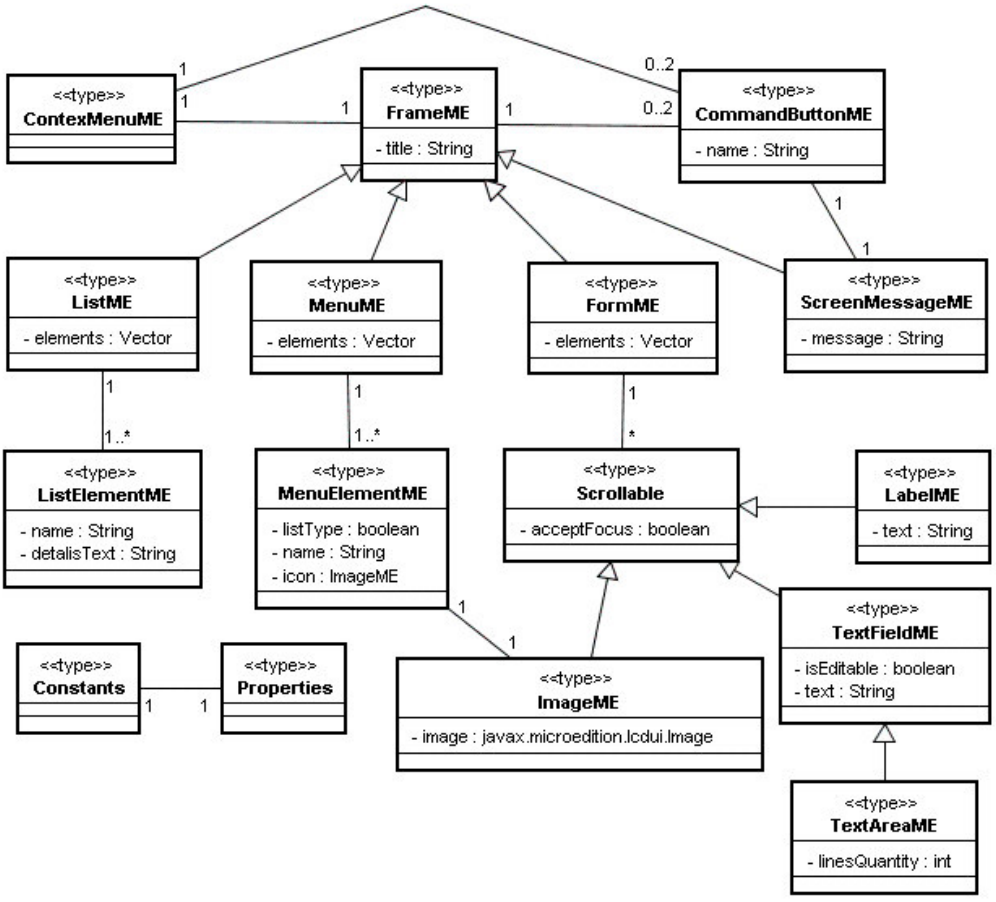

Figura 30: Modelo inicial de tipos de negócio.

A partir do modelo inicial de tipos de negócio, será feita a identificação de tipos `<<core>>` destacados na Figura 31. Os tipos `FrameME`, `ListME`, `MenuME`, `FormME` e `ScreenMessageME` foram considerados tipos `<<core>>` por apresentarem dependências fracas entre os outros elementos do *framework*.

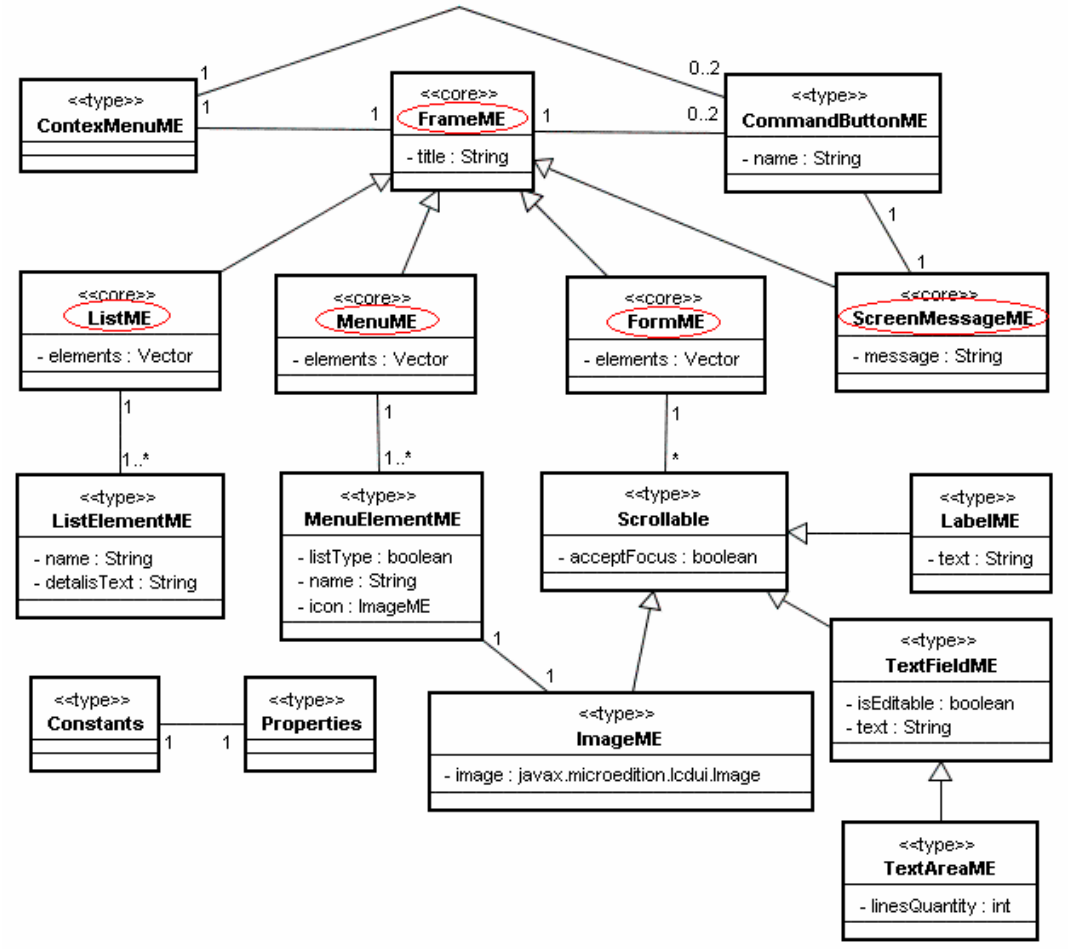

Figura 31: Identificação de tipos <<core>>.

Depois de identificados os tipos `<<core>>`, o próximo passo a ser executado seria a identificação inicial de interfaces do sistema. Entretanto, esse passo precisa ser adaptado para o estudo de caso em questão.

UML Components considera as interfaces do sistema como sendo a descrição dos serviços providos por determinado componente. No caso particular do *framework* construído em J2ME, os componentes não são provedores de serviços, mas sim entidades que apresentam um comportamento a ser reutilizado ou estendido por outros que as utilizam. Como já foi sugerido no início da Seção 4.2, os tipos `<<core>>` que precisam ser estendidos por outras partes têm uma associação com o estereótipo `<<interface type>>`. Assim, as operações presentes no tipo `<<interface type>>` são aquelas que

devem ser estendidas pelas outras partes que o utilizam, estabelecendo um contrato de realização.

Nesta perspectiva, a única entidade que não precisa apresentar associação com um `<<interface type>>` é a `ScreenMessageME`, pois seu comportamento pode ser previsto: apresentação de texto no centro da tela, utilizando somente um botão de comando no canto inferior direito com a ação de voltar para a tela anterior.

Considerando as adaptações sugeridas no processo, a Figura 32 mostra a especificação inicial de interfaces do *framework*. Somente foi necessário a definição de uma interface para a entidade `FrameME`. As outras entidades que herdam dela também herdam a interface, logo não é necessário especificar uma interface para cada uma das entidades filhas.

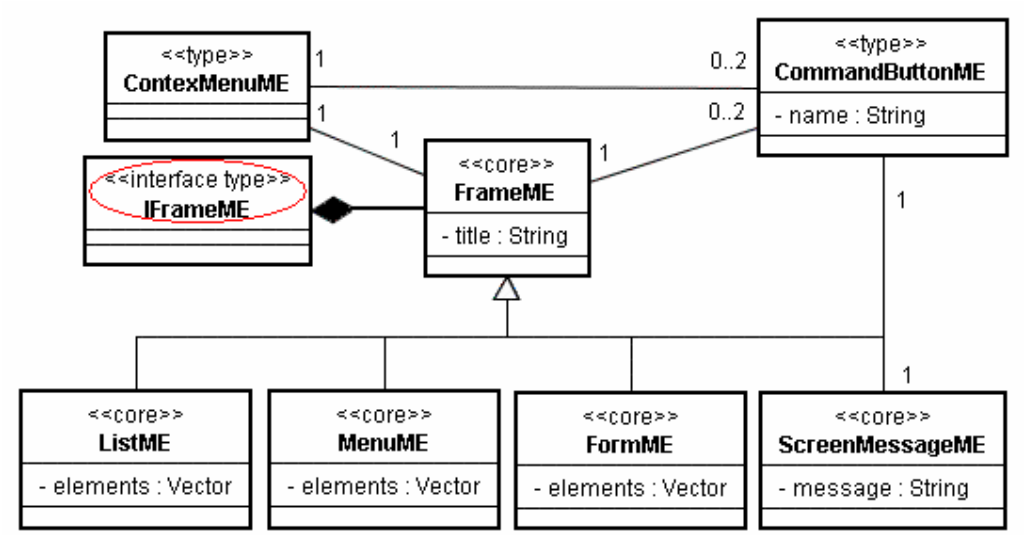

Figura 32: Especificação inicial de interfaces.

O diagrama de arquitetura de componentes não se aplica ao *framework* porque, na prática, qualquer entidade pode ser usada ou estendida sem a necessidade de utilização de uma interface para acessá-la. O diagrama de arquitetura é aplicável a sistemas que se comunicam com outros sistemas e, assim, precisam fornecer interfaces de entrada para utilizar seus componentes.

4.2.2. Interação dos componentes no *framework*

O objetivo principal desta fase é descobrir as operações das interfaces de negócio com suas respectivas assinaturas. Para o caso particular do *framework*, foi considerado que os tipos `<<core>>` subentendem os estereótipos `<<interface type>>` para o caso de contratos de uso. Logo, as operações relativas aos tipos `<<core>>` também serão analisadas.

Algumas adaptações no processo podem ser feitas para se atingir o objetivo desta fase. Neste contexto, um único passo será realizado: analisar detalhadamente o projeto inicial do estudo de caso (ver Seção 4.1) e identificar operações relativas ao tipos `<<core>>` e `<<interface type>>`. O resultado desta análise pode ser observado na Figura 33.

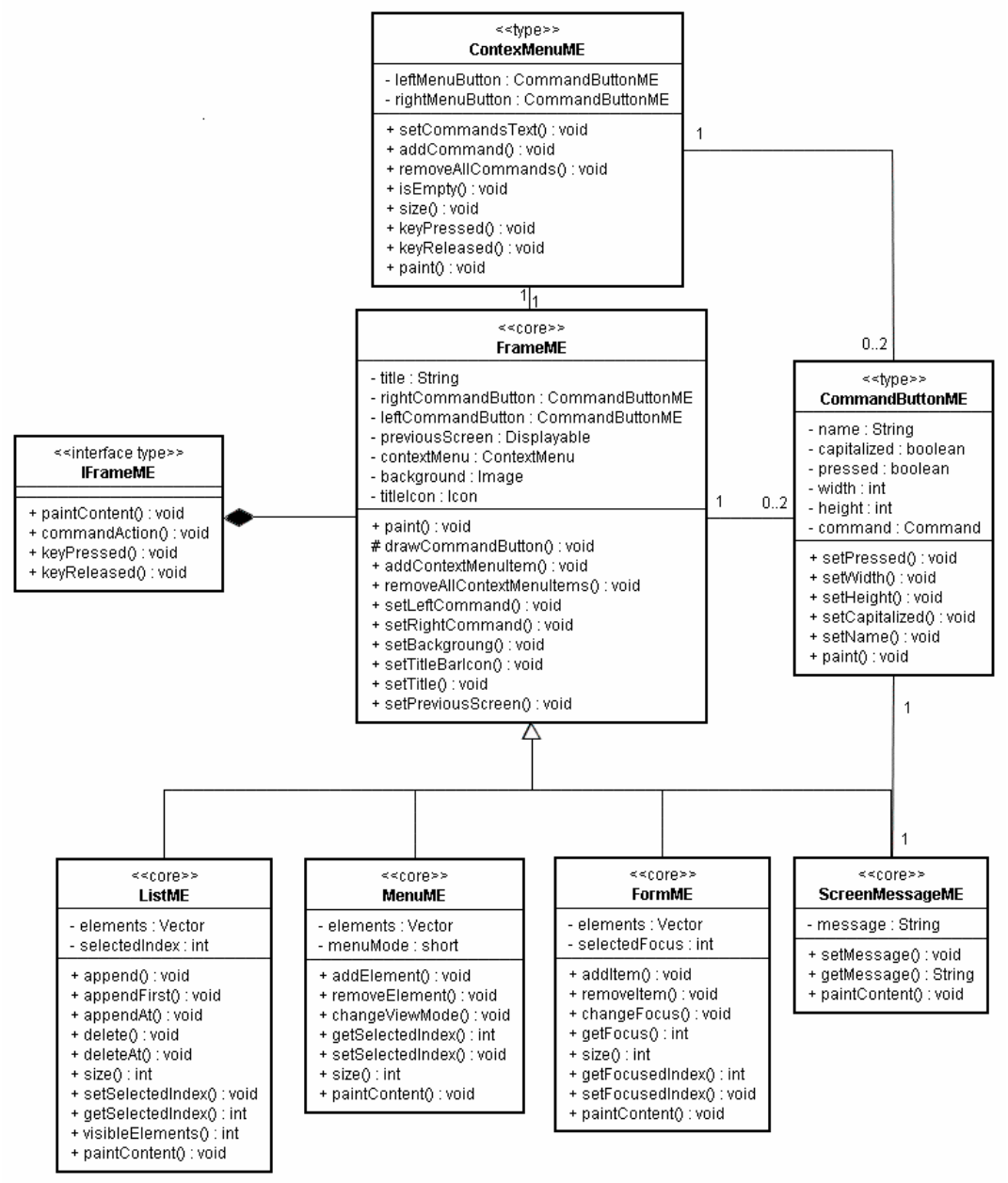

Figura 33: Definição das operações do *framework* com suas respectivas assinaturas.

4.2.3. Especificação de componentes no *framework*

Esta fase tem como objetivo principal produzir o modelo de informação das interfaces e construir o modelo final de especificação do componente. De acordo com a Seção 2.8.3, três passos devem ser seguidos para se atingir esse objetivo: especificar o

modelo de informação da interface, definir pré e pós-condições e montar especificação do componente.

O passo correspondente à especificação do modelo de informação da interface não é aplicável ao estudo de caso em questão porque as interfaces, definidas pelo estereótipo `<<interface type>>`, já estão associadas diretamente a tipos `<<core>>`, os quais já apresentam as informações presentes neste modelo. Ou seja, a Figura 33, que contém a especificação das operações do sistema e suas assinaturas, já especifica também o modelo de informação.

O passo correspondente à definição das pré e pós-condições é aplicável a sistemas que tenham regras de negócios bem definidas, logo este passo não se enquadra no estudo de caso. Geralmente, as pré e pós-condições são levantadas a partir da especificação de casos de uso, o que não é o caso do *framework*. Esse passo seria aplicável num contexto de aplicações em J2ME que tivessem um modelo de casos de uso definido.

Finalmente, o passo relativo à montagem da especificação do componente pode ser executado recebendo como entrada a definição das operações (Figura 33) do *framework*. A Figura 34 apresenta o resultado deste passo.

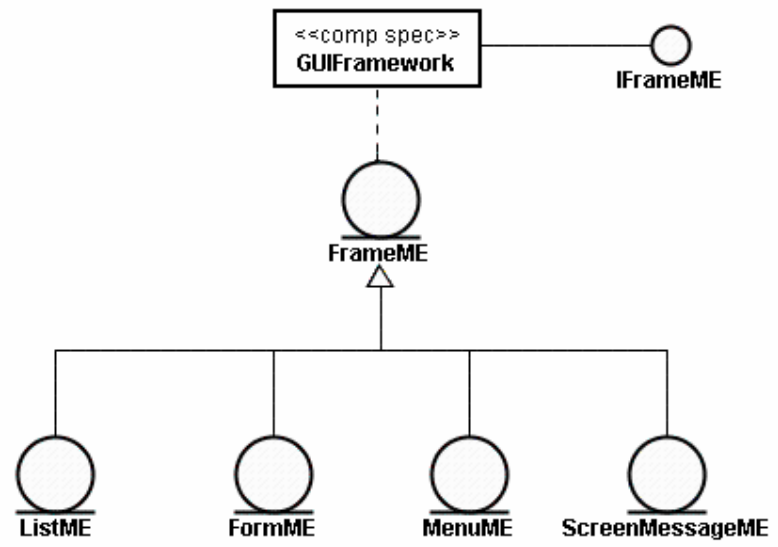

Figura 34: Especificação de componentes do framework.

4.3. Adaptações no método *UML Components* para o estudo de caso

Ao longo da Seção 4.2, foi apresentada a aplicação do método *UML Components* no contexto do estudo de caso. Para que o método se adequasse a este contexto, foram sugeridas algumas adaptações. A Figura 35 resume todas essas adaptações.

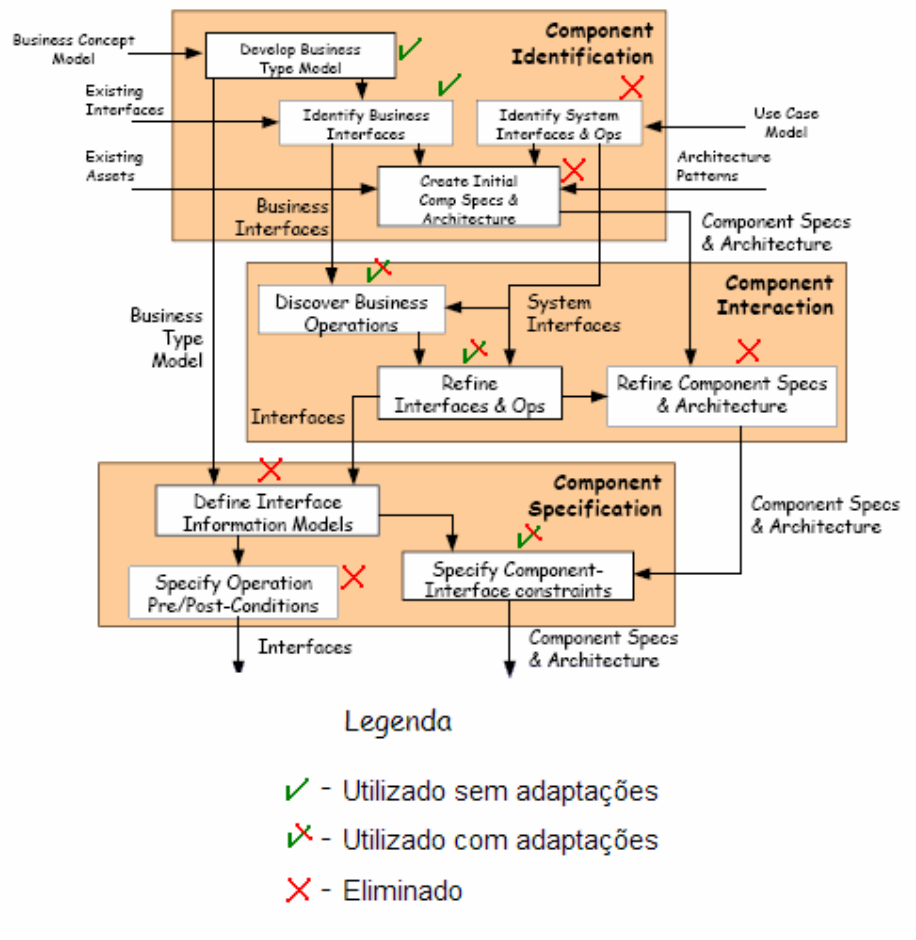

Figura 35: Panorama das adaptações sugeridas no processo de *UML Components*.

4.4. Resultados do estudo de caso

A implementação do *framework* foi realizada com base nas versões CLDC 1.1 e MIDP 2.0 (ver Seção 3.1). As seguintes entidades foram implementadas (as figuras de exemplo de cada tela podem ser visualizadas na Seção 4.1):

- `FrameME` – representa a **tela padrão adaptável** com botões de comando e menu de contexto. Não foi necessário criar uma classe específica para o encapsulamento do menu de contexto. Essa tela pode se moldar em diferentes tamanhos de *displays* de dispositivos. A classe `FrameME` encapsula dois botões de comando, desenhados um de cada lado da tela, e, entre os dois botões, encontra-se o botão de acesso ao menu de contexto. Esta classe é herdade pelas classes `MenuME`, `ListME`, `ScreenMessageME` e `FormME`, as quais também apresentam o mesmo comportamento em diferentes tamanhos de tela;
- `CommandButtonME` – representa um botão de comando inserido na tela padrão adaptável;

- `MenuME` – representa o **menu de itens** que pode ser visualizado de duas formas: em lista ou em ícones. Cada elemento desta classe é representado pela entidade `MenuElementME`;
- `MenuElementME` – representa o elemento adicionado à entidade `MenuME`;
- `ListME` – representa uma **lista de itens** que pode ser exibida de duas formas: em lista ou em miniaturas (*thumbnails*). Cada elemento da lista é representado pela entidade `ListElementME`:
- `ListElementME` – simboliza o objeto que é adicionado à entidade `ListME`;
- `ScreenMessageME` – representa uma **mensagem na tela** que pode ser exibida como um *pop-up*. Se a mensagem de texto for maior que o tamanho da tela, uma barra de rolagem aparece indicando que há mais texto para ser visualizado;
- `FormME` – encapsula o **formulário padronizado** para tratar entrada de dados do usuário e pode receber como elementos as entidade `TextFieldME`, `TextAreaME` e `LabelME`;
- `TextFieldME` – representa um campo de texto para receber entradas do usuário;
- `TextAreaME` – herda da classe `TextFieldME` e representa uma área de texto para receber entradas de texto maiores do que as do que um campo de texto;
- `LabelME` – texto adicionado a um formulário para representar o nome de campos e áreas de texto; e
- Componentes auxiliares – `Constants`, `KeyCodesConstants`, `Properties`, `DisplayHolder`. A classe `Constants` agrupa as constantes utilizadas em todo o *framework* encapsulando cores e espaços entre objetos na tela. Já classe `KeyCodesConstants` guarda todos os atributos referentes aos códigos de teclas do dispositivo. Isso permite instalar a mesma aplicação em diferentes dispositivos obtendo o mesmo comportamento visual. A classe `Properties` permite ler todas as constantes de um arquivo de propriedades, similar a um arquivo de texto, facilitando o desenvolvimento de aplicações para diferentes aparelhos. Por fim, a classe `DisplayHolder` encapsula o objeto `Display` do `MIDlet` facilitando assim para que todos os integrantes do *framework* possam mostrar outras telas a partir da atual, facilitando na navegação entre telas.

As classes que receberam o estereótipo `<<core>>` (`FrameME`, `MenuME`, `ListME`, `ScreenMessageME` e `FormME`) foram codificadas como classes abstratas para simular a implementação de uma interface `IFrameME` (ver Figura 34 – Seção 4.2.3). Assim, a partir

do momento que o desenvolvedor desejar utilizar essas classes, ele deve criar uma nova classe que herde de alguma delas e implementar o método `commandAction` para tratar eventos do usuário e/ou o método `paintContent` para desenhar na tela alguma peculiaridade (ver Figura 33 – Seção 4.2.2).

O *framework* foi codificado com aproximadamente 2386 linhas de código. O Arquivo JAR (*Java Archive*), que o encapsula, teve seu tamanho final de 40 Kb. Levando em consideração que uma aplicação J2ME possa ocupar até 150 Kb, o *framework* ocuparia 27% do espaço total, deixando os outros 110 Kb disponíveis para o desenvolvimento de toda a aplicação.

Todas as dados sobre linhas de código utilizados neste trabalho foram obtidos a partir da ferramenta *Lines of Code Counter* [48]. Para modelagem do estudo de caso foi utilizada a ferramenta *Jude* [40], que suporta adição de estereótipos ao modelo.

A Tabela 3 apresenta os modelos gerados durante todo o desenvolvimento do *framework*[1].

Modelo	***Nome do arquivo***
Modelo conceitual do negócio	ModeloConceitualNegocio.jude
Modelo inicial de tipos do negócio	ModeloTiposNegocio.jude
Identificação dos tipos `<<core>>`	IdentificacaoTiposCore.jude
Especificação inicial de interfaces	EspecificacaoInicialInterfaces.jude
Definição das operações do sistema	OperacoesAssinaturas.jude
Especificação de componentes - arquitetura	EspecificacaoComponentesArquitetura.jude

Tabela 3: Modelos gerados durante o desenvolvimento do *framework*.

A Tabela 4 mostra a relação dos componentes modelados com suas respectivas classes em Java e a quantidade de linhas de código de cada classe.

Componente	***Classes***	***Linhas de código***
FrameME	*FrameME.java* *CommandButtonME.java*	403 152
MenuME	*MenuME.java* *MenuElementME.java*	177 144
ListME	*ListME.java* *ListElementME.java*	361 181
FormME	*FormME.java* *TextFieldME.java* *AlphaTextFieldME.java*	332 146 47

[1] Todos os artefatos podem ser encontrados em http://www.cin.ufpe.br/~lmn2/artefatos_tg.

	TextAreaME.java	69
ScreenMessageME	*ScreenMessageMe.java* *ScrollBarME.java*	102 76
Componentes Auxiliares	*Constants.java* *KeyCodesConstants.java* *Properties.java* *DisplayHolder.java*	87 26 54 29
		2386

Tabela 4: Relação entre os componentes gerados com suas respectivas classes em Java e a quantidade de linhas de código.

4.4.1. Aplicação desenvolvida a partir do *framework*

Uma aplicação simples de exemplo foi desenvolvida reutilizando o *framework* foi e teve o tamanho total de 48 Kb. Esta aplicação foi testada em quatro aparelhos com tamanhos de tela diferentes e apresentou o mesmo comportamento visual em todos eles.

O desenvolvimento da aplicação, que teve no total 375 linhas de código, seguiu os passos citados na Seção 4.5. A Tabela 5 mostra quais foram as classes construídas e quais componentes cada classe reutilizou.

Componentes reutilizados	***Classes construídas***	***Linhas de código***
FrameME e *ScreenMessageME*	*FrameMEExample.java*	88
MenuME e *MenuElementME*	*MenuMEListExample.java* *MenuMEIconsExample.java*	37 54
ListME e *ListElementME*	*ListMEExample.java*	74
FormME, *AlphaTextFieldME* e *TextAreaME*	*FormMEExample.java*	96
----	*MIDletExample.java* (Classe necessária para construção da aplicação)	26
		375

Tabela 5: Classes resultantes da construção de uma aplicação reutilizando o *framework*.

4.5. Construindo aplicações a partir do *framework*

Para se construir aplicações a partir do *framework* desenvolvido é necessário seguir um conjunto de passos:

- **Criar uma nova classe herdando da classe do framework que tenha o comportamento desejado** – neste momento, a nova classe estará herdando toda a forma de desenho da classe mãe, somente precisando adicionar peculiaridades visuais como, por exemplo, cores e textos da tela;
- **Adicionar os botões de comando e comandos do menu de contexto** – os comandos permitem adicionar o tratamento de eventos na interface gráfica;
- **Implementar o método `commandAction` para permitir a manipulação dos comandos** – este método é sempre chamado ocorre um evento disparado pelo usuário e existe um comando associado àquela tecla pressionada. Neste momento, o método `commandAction` é chamado recebendo como parâmetro o comando da tecla pressionada e o objeto `Displayable` da tela em que o comando foi ativado;
- **Implementar, se necessário, o método `paintContent`** – este método é disparado pela classe mãe para permitir à classe filha desenhar qualquer detalhe na tela; e
- **Reescrever, se necessário, os métodos `keyPressed` e `keyReleased`** – deste modo a classe filha pode tratar de uma forma mais personalizada os eventos disparados pelo usuário.

Um exemplo prático pode ser visualizado na Figura 36.

```java
import gui.examples.ExampleMIDlet;
import gui.framework.Constants;
import gui.framework.DisplayHolder;
import gui.framework.FrameME;
import gui.framework.ScreenMessageME;

import java.io.IOException;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;

//Herdando da classe mãe FrameME para capturar seu comportamento na tela
public class FrameMEExample extends FrameME {

    //Definição de comandos para tratamento de eventos disparados pelo usuário
    private Command leftFrameCommand;     //Comando esquerdo do FrameME
    private Command rightFrameCommand;    //Comando direito do FrameME
    private Command leftMenuCommand;      //Comando esquerdo do menu de contexto
    private Command rightMenuCommand;     //Comando direito do menu de contexto
    private Command showMessageCommand;   //Comando para mostrar uma mensagem ao usuário
    private Command contextMenuCommand2;  //Comando adicionado ao menu de contexto

    public FrameMEExample(){
        //Chamando o construtor da classe FrameME, passando o título como parâmetro
        super("FrameME - Example");

        //Definindo imagens de ícone e de fundo de tela
        Image backGround = null;
        Image mainIcon = null;
        try {
            backGround = Image.createImage(Constants.IMAGES_PATH + "background.png");
            mainIcon = Image.createImage(Constants.IMAGES_PATH + "mainIcon.png");
        } catch (IOException e) {}

        //Setando as imagens na classe FrameME
        this.setBackgroundImage(backGround);
        this.setTitleBarIcon(mainIcon);

        //Criando todos os comandos
        leftFrameCommand = new Command("SELECT", Command.OK, 1);
        rightFrameCommand = new Command("EXIT", Command.EXIT, 1);
        leftMenuCommand = new Command("SELECT", Command.OK, 1);
        rightMenuCommand = new Command("BACK", Command.EXIT, 1);
        showMessageCommand = new Command("Show Message", Command.SCREEN, 1);
        contextMenuCommand2 = new Command("Menu Command", Command.SCREEN, 1);

        //Adicionando os comandos da tela principal (FrameME)
        this.setLeftCommand(leftFrameCommand);
        this.setRightCommand(rightFrameCommand);

        //Adicionando os comandos direito e esquerdo do menu de contexto
        this.setContextMenuCommands(leftMenuCommand, rightMenuCommand);

        //Adicionando comandos dentro do menu de contexto
        this.addContextMenuCommand(showMessageCommand);
        this.addContextMenuCommand(contextMenuCommand2);
    }

    //Definindo o metodo commandAction para tratar os comandos adicionados
    public void commandAction(Command c, Displayable d) {
        if (d == this) { //Testando se a tela corrente é esta
            if (c == this.rightCommandButton.getCommand()) {
                //Ação de sair da aplicação, pegando uma instância do MIDlet
                //e chamando o método exitApp
                ExampleMIDlet.getInstance().exitApp();
            } else if (c == this.showMessageCommand) {
                //Apresentando uma mensagem de teste na tela a partir da classe
                //ScreenMessageME
                //Construtor recebe: Título, tela anterior, booleano indicando se é ou nao pop-up
                ScreenMessageME message = new ScreenMessageME("Test", this, true);

                //Configurando a mensage a ser apresentada ao usuário
                message.setText("Testing the message class");

                //Aproveitando o comando do menu para colocar na tela de mensagem
                //um comando do tipo BACK
                message.setLeftCommand(rightMenuCommand);

                //Colocando a tele de mensagem pra ser a tela corrente
                DisplayHolder.show(message);
            }
        }
    }
    //Reescrevendo o método paintContent para desenhar uma imagem no meio da tela
    protected void paintContent(Graphics g) {
        Image flower = null;
        try {
            flower = Image.createImage(Constants.IMAGES_PATH + "flower.png");
        } catch (IOException e) {}
        g.drawImage(flower, DisplayHolder.getScreenWidth()/2,
                    DisplayHolder.getScreenHeight()/2,
                    Graphics.HCENTER | Graphics.VCENTER);
    }
}
```

Declarando comandos

Iniciando comandos

Atribuindo comandos ao *FrameME*

Atribuindo comandos ao menu de contexto

Tratando os comandos no método commandAction

Criando método paintContent para desenhar imagem no centro

Figura 36: Exemplo de utilização do *framework*.

A Figura 37 apresenta o resultado da execução do código de exemplo da Figura 36.

Figura 37: Resultado do exemplo de código da Figura 36.

Todas constantes da aplicação, incluindo cores e distâncias em pixels entre os objetos da tela, podem ser encontradas na classe `gui.framework.Constants`. As constantes com os códigos de tecla, que permitem a instalação da aplicação em diferentes dispositivos mantendo o mesmo comportamento visual, podem ser encontradas na classe `gui.framework.KeyCodesConstants`.

4.6. Análise do estudo de caso

4.6.1. Pontos fortes

Na área de engenharia de *software*, utilizar um processo para guiar todas as fases de desenvolvimento pode melhorar a qualidade do produto final [33][35][36]. Considerando o estudo de caso em questão, aplicar o método *UML Components* ajudou a construir um *framework* com maior qualidade, bem estruturado e modelado.

Dentro das atividades do processo, duas delas podem ser ressaltadas:

- Na fase de identificação dos componentes do *framework* (Seção 4.2.1), o passo correspondente à identificação de tipos `<<core>>`, ajudou bastante a separar quais entidades do sistema são independentes de outras e a montar os outros modelos em seguida a partir destas entidades; e
- Na fase de interação dos componentes do *framework* (Seção 4.2.2), identificar as operações dos componentes a partir das interações entre eles auxilia bastante durante toda a implementação dos mesmos.

4.6.2. Pontos fracos

O método *UML Components* é muito voltado para a definição de interfaces dos componentes para que eles possam ser usados como provedores de serviços. Isso é aplicável para sistemas voltados para *web*, já que estes necessitam fazer interações entre outras partes do sistema ou até mesmo outros sistemas e, por isso, a definição de interfaces torna-se primordial. No estudo de caso em questão, a definição de interfaces

não se faz necessária, levando-se em consideração que as entidades do *framework* podem ser usadas diretamente, atendendo às necessidades de construção de interfaces gráficas com o usuário em J2ME.

Além disso, o próprio contexto de construção de interfaces gráficas na área computacional implica no uso de entidades que apresentam comportamento pré-definido e não realizam ou implementam interfaces. Um exemplo prático desta afirmação é o *Swing* [44], *framework* desenvolvido pelo time da Sun para construção de interfaces gráficas em Java para *desktops*.

Outro ponto fraco a ser considerado é que os exemplos encontrados com estudos de caso reais de aplicação do método *UML Components* utilizam como tecnologia alvo EJB [45] (*Enterprise JavaBeans*) ou COM+ [46] (*Component Object Model*), ambos voltados para sistemas web.

4.6.3. Limitações

Uma limitação apresentada pelo método *UML Components* no estudo de caso é sobre o estereótipo `<<interface type>>`. Este estereótipo representa o estabelecimento de um contrato de uso entre o componente e a outra parte que vai utilizá-lo. No caso particular do *framework*, algumas entidades precisam ser estendidas pela parte que vai utilizá-la para adicionar o tratamento de eventos disparados pelo usuário durante sua interação com a interface gráfica. Ou seja, não somente um contrato de uso vai ser estabelecido, mas também um contrato de realização. Para este caso em particular, foi sugerida uma adaptação no método, anteriormente descrita na Seção 4.2 – pontos i e ii.

4.6.4. Lições aprendidas

Diante das adaptações que precisaram ser feitas no processo apresentado neste capítulo, pode-se observar que *UML Components* apresenta um conjunto de atividades genéricas que podem ser utilizadas como referência e modificadas de acordo com o contexto em que se está aplicando o método. Em analogia ao que foi dito, pode-se citar o RUP [36], que apresenta um grande conjunto de atividades com um *workflow* complexo e, quando utilizado na prática, sofre modificações de acordo com o contexto de desenvolvimento de software que está sendo aplicado.

4.7. Resumo

Neste capítulo, foi apresentada a aplicação do método *UML Components* na construção de um *framework* para desenvolvimento de interface gráfica em J2ME.

O projeto inicial do estudo de caso é discutido na Seção 4.1. Nesta seção, é feita uma definição inicial dos requisitos do escopo do *framework*.

A Seção 4.2 mostra, em detalhes, quais foram todos os passos seguidos para a construção do *framework* utilizando o método *UML Components* com suas respectivas sugestões de adaptações para o contexto em questão. Na Seção 4.3, é apresentado um panorama geral de todas as adaptações no método em relação às fases do processo original. A Seção 4.4 mostra quais foram os resultados obtidos após a finalização do estudo de caso. Em seguida, a Seção 4.5 apresenta passo a passo como desenvolver uma aplicação em J2ME a partir do *framework* e descreve um exemplo prático disso.

Por fim, a Seção 4.6 faz uma análise do estudo de caso apresentando os pontos fortes e fracos e as limitações da aplicação do método no *framework*, além de descrever as lições aprendidas com este trabalho.

O capítulo seguinte apresenta as considerações finais sobre todo o trabalho.

Capítulo 5

Conclusões

5.1. Principais Contribuições

Este trabalho mostrou-se pioneiro ao propor a aplicação de um método de Desenvolvimento Baseado em Componentes no contexto de aplicações em J2ME. Os métodos estudados (ver Seção 2.6) não apresentaram adequações para este contexto. Diante disso, o método que melhor se enquadrasse ainda deveria passar por algumas modificações. Após análise realizada na Seção 2.6.6, o *UML Components* foi escolhido para ser adaptado e utilizado na construção do estudo de caso. Neste ponto, o trabalho apresenta uma das principais contribuições sugerindo modificações no processo de *UML Components* proposto por Daniels e Cheeseman [11].

Outra contribuição importante diz respeito ao alinhamento dos propósitos deste trabalho com a próxima versão do MIDP. A Sun pretende seguir a linha de utilização de componentes em aplicações J2ME a partir do MIDP 3.0, atualmente em desenvolvimento, como foi publicado no evento *JavaOne 2005* [47]. A idéia é poder instalar um conjunto de componentes no dispositivo que irá rodar as aplicações e, depois disso, qualquer outra aplicação instalada naquele dispositivo poderia utilizar os componentes. Ou seja, as aplicações teriam um tamanho final menor por reutilizarem os componentes já anteriormente instalados e, além disso, também poderiam obter uma qualidade do produto final mais satisfatória, já que os componentes teriam sido testados anteriormente.

Além das contribuições citadas, o resultado deste trabalho é um *framework* para criação de interface gráfica em J2ME que pode ser utilizado sem restrições de aparelhos onde as aplicações serão instaladas. Empresas como a Motorola [49], Nokia [50], Siemens [51] e Sony Ericsson [52], dentre outras, se esforçam para construir *frameworks* com o mesmo propósito do construído neste trabalho. Entretanto, esses *frameworks* são proprietários e somente funcionam adequadamente na plataforma sobre a qual é desenvolvido.

5.2. Trabalhos futuros

Alguns trabalhos futuros podem ser considerados:

- Desenvolver aplicações em J2ME reutilizando o *framework* e também aplicando o método *UML Components*. Desta forma seria possível avaliar a completa aplicação do método dentro do contexto da tecnologia, tanto para a criação do *framework*, quanto para criação de aplicações que o utilizem. Este trabalho apresentou a fase de implementação de aplicações, deixando para um trabalho futuro as fases de análise e projeto, utilizando o método *UML Components*;
- Adaptar o *framework* para usar *Generative Programming* [53]. Desta maneira, seria possível dar como entrada ao *framework* modelos e receber como saída implementações de código prontas para serem adaptadas. Uma possível forma para futura implementação desta proposta seria montar modelos a partir de arquivos XML (*Extensible Markup Language*). Os arquivos descreveriam todas as propriedades que a interface deveria apresentar, como, por exemplo, cores e tamanhos de espaços entre objetos na tela. Atualmente, o *framework* lê essas propriedades de um arquivo de constantes; e
- Adicionar novos componentes ao *framework*. Um componente que poderia ser desenvolvido em um trabalho futuro poderia ser responsável por adaptar caracteres não latinos, como, por exemplo, hebraico e coreano, às telas que precisem exibir texto. Além disso, alguns outros componentes poderiam ser construídos para serem adicionados ao formulário padronizado, tais como, caixas de seleção, por exemplo.

5.3. Considerações finais

Este trabalho teve como objetivo criar um *framework* para desenvolvimento de interface gráfica em J2ME utilizando um método de Desenvolvimento Baseado em Componentes, que neste caso foi o *UML Components*.

A relevância deste trabalho existe quando se observa que os métodos de DBC atualmente disponíveis não se enquadram no contexto da plataforma J2ME. Desta forma, foram sugeridas algumas adaptações no método, permitindo, assim, que o *framework* pudesse ser normalmente modelado.

Além disso, a estrutura do *framework* demonstrou estar dentro da realidade de projetos em J2ME montados a partir de reutilização de código.

Outra consideração relevante a ser observada é a importância da utilização de métodos e processos de Engenharia de Software para a completude com sucesso deste trabalho.

Referências

[1]. **Java™ 2 Platform, Micro Edition (J2ME)**. Disponível em http://java.sun.com/j2me/, Maio 2005.

[2]. **Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification**. Disponível em http://java.sun.com/j2se/1.4.2/docs/api/, Maio 2005.

[3]. Muchow, J., W. **Core J2ME – Technology & MIDP**, Pearson Makron Books, 2004.

[4]. Keogh, J. **J2ME – The Complete Reference**, McGraw-Hill/Osborne, 2003.

[5]. **Java Specification Request 118: Mobile Information Device Profile 2.0**. Disponível em http://jcp.org/jsr/detail/118.jsp, Maio 2005.

[6]. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. **Design Patterns**, Addison-Wesley Professional, 1995.

[7]. **J2ME MicroDevNet**. Disponível em http://www.microjava.com/, Maio 2005.

[8]. **Sun Microsystems**. Disponível em http://www.sun.com/, Maio 2005.

[9]. **3G Americas**. Disponível em http://www.3gamericas.org, Maio 2005.

[10]. **Unified Modeling Language™ - UML**. Disponível em http://www-306.ibm.com/software/rational/uml/, Maio 2005.

[11]. Cheeseman, J.; Daniels, J. **UML Components: A Simple Process for Specifying Component-Based Software**, Addison-Wesley, 2001.

[12]. Schneider, J.; Han, J. **Components – the Past, the Present, and the Future**, Swinburne University of Technology, School of Information of Technology, Junho 2004.

[13]. Bachmann, F.; Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; Wallnau, K. **Technical Concepts of Component-Based Software Engineering 2nd Edition**, Carnegie Mellon Software Engineering Institute, Maio 2000.

[14]. Almeida, E., S. **Uma Abordagem para o Desenvolvimento de Software Baseado em Componentes Distribuídos**, Dissertação de mestrado, Universidade Federal de São Carlos, 2003.

[15]. **Software Engineering Institute (SEI)**. Disponível em http://www.sei.cmu.edu/, Maio 2005.

[16]. Sametinger, J. **Software Engineering with Reusable Components**, Springer-Verlag, 1997.

[17]. **Webopedia**. Disponível em http://www.webopedia.com/, Junho 2005.

[18]. Möller, A.; Åkerholm, M.; Fredriksson, J.; Nolin, M. **Software Component Technologies for Real-Time Systems - An Industrial Perspective**, Mälardalen University, 2003.

[19]. **Catalysis.org site**. Disponível em http://www.catalysis.org/, Junho 2005.

[20]. **PECOS Project**. Disponível em http://www.pecos-project.org/, Junho 2005.

[21]. D'Souza, D.; Wills, A., C. **Objects, Components, and Frameworks with UML - The Catalysis Approach**, Addison-Wesley, 2001.

[22]. McIlroy, M., D. **Mass Produced Software Components**, Nato Software Engineering Conference. 1968.

[23]. Szyperski, C. **Component Software – Beyond Object-Oriented Programming**, Addison-Wesley, 2002.

[24]. Heineman, G., T.; Councill, W., T. **Component-based Software Engineering: Putting the Pieces Together**, Addison-Wesley, 2001.

[25]. Gold, N.; Mohan, A. **A Framework for Understanding Conceptual Changes in Evolving Source Code**, Proceedings of the International Conference on Software Maintenance. IEEE, 2003.

[26]. Pressman, R., S. **Software Engineering: A Practitioner's Approach**, McGraw-Hill, 2001.

[27]. Tanenbaum, A., S. **Distributed Systems: Principles and Paradigms**, Prentice Hall, 2002.

[28]. Alvaro, A.; Meira, S., R., L. **Component Certification: A Component Quality Model**, IV Simpósio Brasileiro de Qualidade de Software, III Workshop de Teses e Dissertações em Qualidade de Software, Porto Alegre, Brasil, 2005.

[29]. Alvaro, A.; Almeida, E., S.; Meira, S., R., L. **Software Component Certification: A Survey**, 31st IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Component-Based Software Engineering Track, Portugal, IEEE Press, 2005.

[30]. **WebServices.org**. Disponível em http://www.webservices.org/, Junho 2005.

[31]. Boertien, N.; Steen, M., W., A.; Jonkers, H. **Evaluation of Component-Based Development Methods**, Telematica Instituut.

[32]. Stojanovic, Z.; Dahanayake, A.; Sol, H. **A Methodology Framework for Component-Based System Development Support**, The 6th CaiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design EMMSAD'01, Interlaken, Switzerland, Junho 2001.

[33]. Kruchten, P. **Rational Unified Process: An Introduction**, Addison-Wesley, 1999.

[34]. Rumbaugh, J., R. **Object Oriented Modeling and Design**, Prentice Hall, 1991.

[35]. Jacobson, I., et. al. **Object-Oriented Software Engineering - A Use Case-Driven Approach**, Reading, MA: Addison-Wesley, 1992.

[36]. Jacobson, I., et. al. **The Unified Software Development Process**, Addison-Wesley. USA 4nd edition, 2001.

[37]. **Catalyst Methodology** (Internal Document), Computer Sciences Corporation, 1995.

[38]. **OMG - Object Management Group: Unified Modeling Language 1.3 specification**. Disponível em http://www.omg.org/uml/, 2005.

[39]. **Gentleware Poseidon – Ferramenta para modelagem UML**. Disponível em http://www.gentleware.com/, Julho 2005.

[40]. **Jude – UML Modeling Tool**. Disponível em http://www.esm.jp/jude-web/en/index.html, Julho 2005.

[41]. **Component-Based Design: A Complete Worked Example**. Disponível em http://oopsla.acm.org/oopsla2k/fp/tutorials/tut71.html, Julho 2005.

[42]. **J2ME Wireless Toolkit**. Disponível em http://java.sun.com/products/j2mewtoolkit, Julho 2005.

[43]. **The Object Constraint Language**. Disponível em http://www.csci.csusb.edu/dick/samples/ocl.html, Agosto 2005.

[44]. **Creating a GUI with JFC/Swing**. Disponível em http://java.sun.com/docs/books/tutorial/uiswing/, Agosto 2005.

[45]. **Enterprise JavaBeans Techonology**. Disponível em http://java.sun.com/products/ejb/, Agosto 2005.

[46]. **Component Object Models Technologies**. Disponível em http://www.microsoft.com/com/, Agosto 2005.

[47]. **The Mobility and Devices Track at JavaOne 2005**. http://developers.sun.com/techtopics/mobility/javaone2005/j2metrack.html, Agosto 2005.

[48]. **Lines of Code Counter**. Disponível em http://www.csc.calpoly.edu/~jdalbey/SWE/PSP/LOChelp.html, Agosto 2005.

[49]. **Motorola™**. Disponível em http://www.motorola.com/, Agosto 2005.

[50]. **Nokia™**. Disponível em http://www.nokia.com/, Agosto 2005.

[51]. **Siemens™ Mobile**. Disponível em http://www.siemensmobile.com/, Agosto 2005.

[52]. **Sony Ericsson™**. Disponível em http://www.sonyericsson.com, Agosto 2005.

[53]. Czarnecki, K.; Eisenecker, U., W. **Generative Programming: Methods, Tools, and Applications**. Addison-Wesley, 2000.

[54]. **Centro de Estudos e Sistemas Avançados do Recife – C.E.S.A.R**. Disponível em http://www.cesar.org.br, Agosto 2005.

[55]. **Centro de Informática – UFPE**. Disponível em http://www.cin.ufpe.br, Agosto 2005.

Apêndice A – Questionário da pesquisa sobre projetos em J2ME

Seção 1 – Identificação e Perfil

Nome: _____ Data: ___/___/_____

Curso: ( ) Ciência da Computação ( ) Engenharia da Computação
( ) Sistemas de Informação ( ) Processamento de Dados
( ) Outros _____

Nível: ( ) Curso Técnico ( ) Graduação
( ) Mestrado ( ) Doutorado ( ) Especialização

Em qual das categorias abaixo você melhor se encaixa como desenvolvedor utilizando a tecnologia J2ME:
( ) Não tenho experiência em desenvolvimento de aplicações nessa área
( ) Desenvolvi alguns projetos nessa área como parte de disciplina de graduação ou pós-graduação.
( ) Desenvolvi profissionalmente alguns projetos nessa área (até 5)
( ) Desenvolvi profissionalmente muitos projetos nessa área (acima de 5)
( ) Outros, especifique: _____

Você já participou do desenvolvimento de aplicações nas quais foi reutilizado código de outras aplicações para construção de interface com o usuário:
( ) Não
( ) Sim, somente classes (código-fonte) foram reutilizadas
( ) Sim, classes testadas foram reutilizadas
( ) Sim, classes documentadas foram reutilizadas
( ) Sim, classes documentadas e testadas (componentes de software) foram reutilizadas

Dos projetos que você participou, quantos por cento do tempo total de codificação do projeto você acha que foi gasto somente com o desenvolvimento de interface gráfica com o usuário:
( ) Menos de 10% ( ) Entre 10% e 20% ( ) Entre 20% e 40%
( ) Entre 40% e 60% ( ) Entre 60% e 80% ( ) Mais de 80%

Seção 2 – Obtendo dados sobre projetos em J2ME (específico para programadores J2ME)

Considerando aplicações desenvolvidas em camadas, enumere os itens listados abaixo em ordem crescente de dificuldade encontrada durante a codificação (1 – para menor dificuldade; 6 – para maior dificuldade):
( ) Interface com o usuário (GUI)
( ) Classes de comunicação
( ) Classes de negócio
( ) Classes de persistência
( ) Tratamento de eventos
( ) Tratamento de exceções

Qual o nível de dificuldade que você encontra quando precisa desenvolver interface com o usuário em J2ME utilizando a classe Screen e suas subclasses:
( ) Nunca trabalhei com classes do tipo Screen
( ) Muito fácil ( ) Fácil( ) Médio
( ) Difícil ( ) Muito difícil

Qual o nível de dificuldade que você encontra quando precisa desenvolver interface com o usuário em J2ME utilizando a classe Canvas:
( ) Nunca trabalhei com classes do tipo Canvas
( ) Muito fácil ( ) Fácil( ) Médio
( ) Difícil ( ) Muito difícil

Qual tipo interface com o usuário você considera mais amigável quando se utiliza a tecnologia J2ME:
( ) Nunca tive experiência no uso de interface com o usuário em J2ME
( ) Aquela construída a partir da classe Screen e suas subclasses
( ) Aquela construída a partir da classe Canvas
( ) Aquela construída a partir de uma conjunção da classe Canvas com a classe Screen e suas subclasses

Considerando os projetos que você participou que reutilizaram componentes prontos para criação de interface com o usuário, quantos por cento de ganho de produtividade você acha que foram adicionados ao tempo total de desenvolvimento do projeto:
( ) Menos de 10% ( ) Entre 10% e 20% ( ) Entre 20% e 40%
( ) Entre 40% e 60% ( ) Entre 60% e 80% ( ) Mais de 80%

Levando em consideração as características peculiares de cada interface com o usuário construída em J2ME, você acha viável utilizar componentes de software num novo projeto sem alterar o código fonte desses componentes?
( ) Não, de forma alguma
( ) Neutro
( ) Sim, em algumas partes do projeto
( ) Sim, em todo o projeto