UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

MoSOA: Um *Framework* para Desenvolvimento de Aplicações Móveis Orientadas a Serviços

TRABALHO DE GRADUAÇÃO

Aluno: Vítor Torres Braga - vtb@cin.ufpe.br
Orientador: Silvio Meira - srml@cin.ufpe.br

Janeiro de 2008

ASSINATURAS

Este Trabalho de Graduação é resultado dos esforços do aluno Vítor Torres Braga, sob a orientação do professor Silvio Meira, sob o título: "*MoSOA: Um Framework para Desenvolvimento de Aplicações Móveis Orientadas a Serviços*". Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

_____

Vitor Torres Braga

_____

Silvio Meira

"A tecnologia, é de fato, o fruto, ou a expressão, de uma maneira particular de entender o mundo. A ciência é a base para essas expressões"
(Dalai Lama, O Universo Em Um Átomo)

Dedicado à minha família, em especial a minha mãe, meu pai e minha irmã.

AGRADECIMENTOS

A minha família que me ajudou e motivou durante toda a minha vida. Em especial a meu pai, minha mãe e minha irmã, por serem exemplos de dignidade, honestidade e perseverança que modelaram o meu caráter nesses 24 anos de minha vida. Meus verdadeiros heróis.

Ao professor Sergio Cavalcante por ter me iniciado na pesquisa acadêmica e aberto as portas para o mercado de trabalho. Apesar de está um pouco afastado da universidade, precisamos de mais professores como ele no curso de engenharia da computação para termos como referencia.

Ao meu co-orientador Vinicius Garcia pelas sugestões apresentadas, paciência e excelente orientação ao longo do desenvolvimento do trabalho. Muito obrigado pelo tempo gasto com leitura e correções de cada capitulo deste trabalho.

Ao professor Silvio Meira pelas idéias, aulas e palestras que tive a oportunidade de assistir e pela oportunidade dada para desenvolvimento do trabalho.

A os meus amigos de Maceió: Chita, Cabeça, Kiko, Negão, BT, Tiago, Tonhão, Zana, Xiba, Lopes, Jambo, Sergio, Bruno, Rick pelo companheirismo e amizade. Em especial a meu amigo Otávio Lessa pelo convívio, conversas, conselhos, angustias e descontrações vividas durante os seis anos que moramos juntos em Recife (ainda escrevo um livro contanto as historias dessa época).

A meus amigos de Ciência da Computação(Leo, Zé, Dudu, Joabe, Allan,Tiago, Geraldo) e aos guerreiros do curso de Engenharia da Computação(Pedim, Paulo, Marcelo) .

Aos meus parceiros empreendedores da MobilIT Farley e Firma por terem sido pessoas com as quais sempre pude contar e compartilhar os sucessos e fracassos.

RESUMO

A cada dia cresce o numero de serviços disponíveis na web (buscas, mapas, *feeds*, vídeos e outros). Segundo pesquisa do *Gartner Group*, é uma tendência mundial as empresas adotarem Arquitetura Orientada a Serviços, do inglês *Service Oriented Architecture* (SOA), proporcionando mais agilidade e flexibilidade a seus negócios.

Diante desse contexto, a proposta desse trabalho é construir um *framework* para desenvolvimento de aplicações móveis orientadas a serviços aplicando um método de Desenvolvimento Baseado em Componentes (DBC). O objetivo é elaborar uma arquitetura padrão para as aplicações, que facilite a inclusão de novos serviços (seguindo os princípios de SOA), e fornecer componentes que auxiliem o desenvolvimento de software nesse domínio de aplicações.

Palavras chave: Engenharia de Software, Arquitetura Orientada a Serviços, *Mobile* SOA, Reuso de Software, *Framework*, Desenvolvimento Software Baseado em Componentes, *Java Micro Edition*

ABSTRACT

Every day increases the number of services available on the web (search, maps, feeds, videos and other). According Gartner Group, is a global tendency companies adopt a Service Oriented Architecture (SOA) to provide more agility and flexibility to its business.

In this context, the proposal of this work is to build a framework to develop mobile SOA applications applying a Component Based Development (CBD) process. The goal is to develop a standard design for applications, to facilitate the inclusion of new services (following the SOA principles), and provide components that help software implementation in these application domain.

Key words: Software Engineering, Service Oriented Architecture, Mobile SOA, Software Reuse, Framework, Component Based Development, Java Micro Edition.

Índice

Índice de figuras

Índice de Tabelas

1. Capítulo 1 – INTRODUÇÃO

1.1. Contexto

Segundo estudo da consultoria *In Satt*, o número total de celulares no mundo será de 3,1 bilhões até 2011 [2]. O dado representa um crescimento de 58% nos próximos quatro anos, contra 1,8 bilhão de aparelhos registrados em 2006. Hoje, estima-se que esse número esteja em torno de 2,8 bilhões [1], ou seja, celulares são de longe os dispositivos eletrônicos mais populares do planeta, seguidos pelos computadores pessoais e televisores [4].

Acompanhando esses números, o mercado de aplicações para dispositivos móveis vem crescendo em todo o mundo e, de acordo com estudos no setor [65], deverá crescer em média 102% ao ano nos próximos cinco anos. Nos Estados Unidos deve movimentar cerca de U$ 9 bilhões [5], e no Brasil, só o mercado corporativo deve movimentar U$ 40 milhões em 2007 [7].

A cada dia cresce o numero de serviços disponíveis na *web* (buscas, mapas, *feeds*, vídeos, redes sociais e outros). As operadoras estão começando a migrar para as tecnologias 3G[1] a fim de finalmente tornar realidade a banda larga móvel, permitindo o desenvolvimento de aplicações móveis que utilizem esses serviços.

[1] 3G é um termo usado para nova geração de redes de telefonia sem fio de alta velocidade.

1.2. Problema

Apesar desse cenário promissor, ainda existe uma escassez de ferramentas, trabalhos e artigos que abordem diretamente o desenvolvimento de aplicações móveis com Arquitetura Orientada a Serviços [6], do inglês *Service Oriented Architecture* (SOA), nos principais institutos de pesquisa de engenharia de software como o *Software Engineering Institute* (SEI) [10], *Association for Computing Machinery* (ACM) [11], IBM [12] e Centro de Estudos e Sistemas Avançados do Recife (C.E.S.A. R)[13].

Diante desse contexto, a proposta desse trabalho é construir um *framework* [14] para desenvolvimento de aplicações móveis orientadas a serviços aplicando um método de Desenvolvimento de Software Baseado em Componentes (DBC) [52]. O objetivo é elaborar uma arquitetura padrão para as aplicações, que facilite a inclusão de novos serviços (seguindo os princípios de SOA), e fornecer componentes que auxiliem o desenvolvimento desses tipos de aplicações tais como:

- **Componente para comunicação com *web services*** [21], sem a necessidade da *Web Services* API [22] estar implementada no dispositivo, deixando essa tarefa simples e transparente para o desenvolvedor;

- **Componente para internacionalização**, fornecendo um mecanismo para que as aplicações suportem várias línguas sem a necessidade de mudanças no código fonte; e,

- **Componente para leitura de propriedades da aplicação** como atributos de telas (altura, largura, espaçamentos), localização e parâmetros de serviços, facilitando o porting das aplicações para vários aparelhos.

1.3. Organização do trabalho

O objetivo desse trabalho é especificar, projetar e implementar um *framework* para desenvolvimento de aplicações móveis aplicando os princípios de **Arquitetura Orientada a Serviços** para a plataforma ***Java Micro Edtion*** (Java ME) utilizando um **método de Desenvolvimento Baseado em Componentes.** Para atingir esse objetivo, o trabalho está divido como se segue:

- **Capítulo 2 – Desenvolvimento Baseado em Componentes**

  Neste capitulo é apresentada uma breve introdução sobre reutilização de software, detalhando seus principais conceitos e definições. No final, é feita uma análise dos principais métodos de desenvolvimento baseado em componentes a fim de escolher o mais adequado ao trabalho.

- **Capítulo 3 – Arquitetura Orientada a Serviço**

  Apresenta os conceitos e tecnologias relativos à Arquitetura Orientada a Serviços, do inglês *Service Oriented Architecture* (SOA), e as principais vantagens em adotá-la.

- **Capítulo 4 – *Java Micro Edition* (Java ME)**

  Este capítulo apresentada a tecnologia Java ME que servirá de base para implementação do *framework* proposto, mostrando uma visão geral da plataforma e o processo para construção de aplicações.

- **Capítulo 5 – MoSOA: Um *Framework* para Desenvolvimento de Aplicações Móveis Orientadas a Serviços**

  O capítulo 5 apresenta todos os detalhes de desenvolvimento do *framework* usando o método *UML Components*. No final é apresentada a aplicação construída para validação do *framework*.

- **Capítulo 6 – Conclusão**

Neste capítulo é apresentado um resumo de todo o trabalho, mostrando o relacionamento com outros trabalhos feitos. Por fim são mencionadas as oportunidades e limitações a serem desenvolvidas em trabalhos futuros.

A figura 1 mostra o *roadmap* para este trabalho.

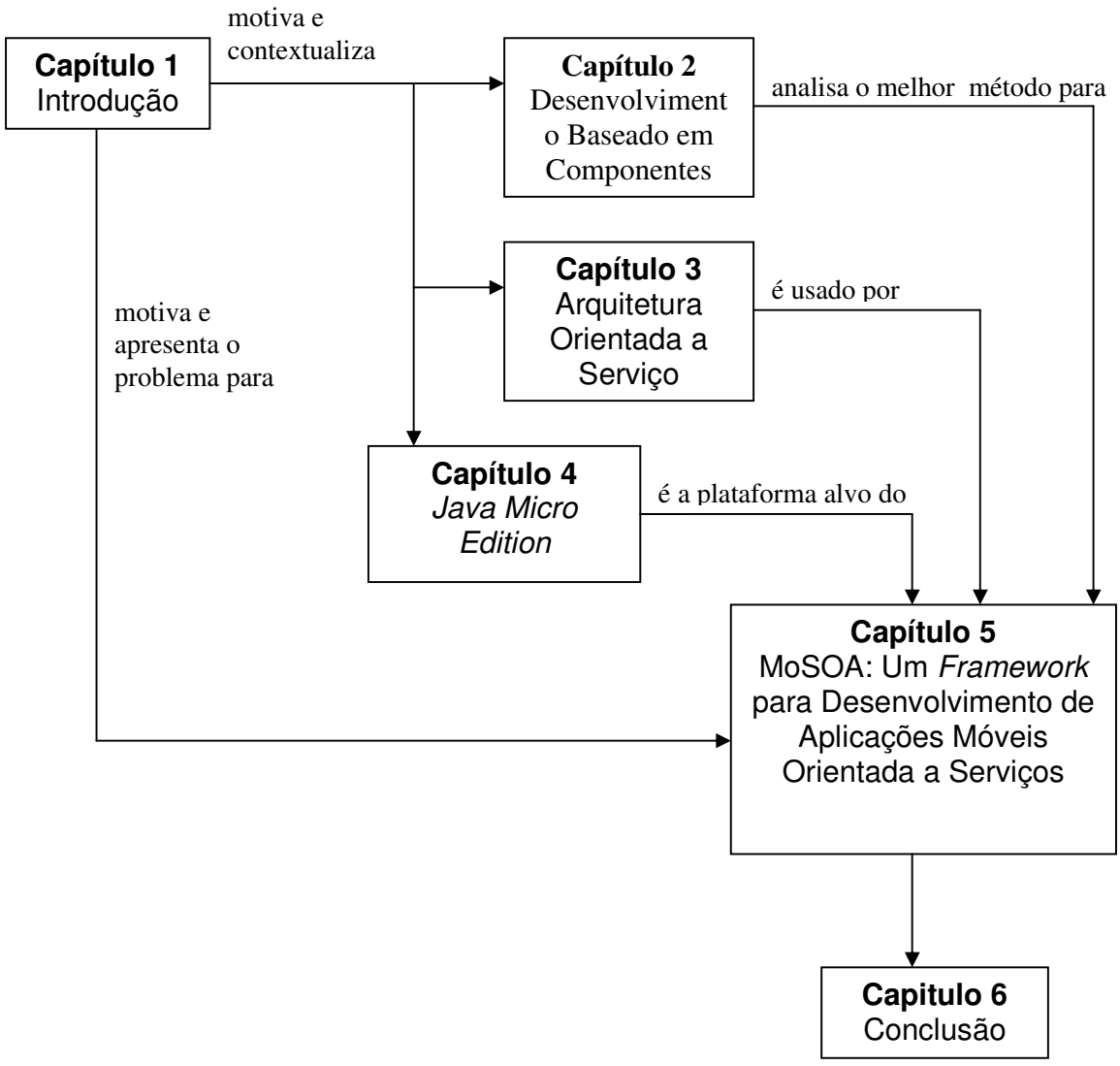


Figura 1 - *Roadmap* do trabalho

2. Capítulo 2 – Desenvolvimento Baseado em Componentes

A maioria dos softwares existentes no mercado foi desenvolvida baseada em blocos monolíticos, formados por partes relacionadas com alto grau de acoplamento [36]. Com isso, o Desenvolvimento Baseado em Componentes (DBC) surgiu como um novo modelo de desenvolvimento de software para transformar esses blocos monolíticos em componentes, diminuindo a complexidade e o custo de desenvolvimento, onde os componentes podem ser usados em outras aplicações [38].

O termo desenvolvimento baseado em componentes vem sendo usado com bastante freqüência em engenharia de software nos dias de hoje. Tanto a industria de software como a academia aceitam a idéia de que construir grandes sistemas a partir de partes menores (componentes), que já foram implementadas, diminui o tempo de desenvolvimento e aumenta a qualidade do produto final [15]. Essa idéia segue um dos princípios básicos da ciência da computação: dividir para conquistar.

Diante desse contexto, este capítulo apresenta uma breve introdução em reuso de software na seção 3.1. Na seção 3.2, será apresentado o conceito de componentes de software. No final, a seção 3.3 mostra os principais métodos de desenvolvimento baseado em componentes, fazendo uma comparação entre eles para justificar o mais adequado para utilização neste trabalho.

2.1. Reuso de software

Desde o inicio da história do desenvolvimento de software, no final da década de 40 [39], tem ocorrido um fluxo contínuo de inovações para melhorar os processos de desenvolvimento de software. Alguns avanços na área de

engenharia de software tem contribuído para aumentar a produtividade na construção de sistemas, tais como Programação Orientada a Objetos (POO), Desenvolvimento de Software Baseado em Componentes (DSBC), Engenharia de Domínio (ED), Linhas de Produtos de Software (LPS), entre outros [37]. Esses avanços são conhecidas formas de obter reutilização de software. Essa evolução ocorreu devido ao aumento da complexidade dos projetos de softwares, juntamente com problemas envolvendo cronogramas, custos e fracassos em atender os requisitos definidos pelos clientes [40]. Diante deste cenário, reuso de software oferece a oportunidade de alcançar melhorias radicais nos métodos de produção de software existentes.

Neste contexto, as próximas seções mostram as motivações, definições e benefícios na reutilização de software.

2.1.1. Motivação

Na literatura existem trabalhos publicados com diferentes taxas sobre reutilização de software [42], no entanto, alguns estudos mostram que 40% a 60% do código é reutilizável de uma aplicação para outra, 60% do design e código são reutilizáveis em aplicações de negócios, 75% das funções de um programa são comuns a mais de uma aplicação, e apenas 15% do código encontrado na maioria dos sistemas é único e novo para uma aplicação específica [39]. Com o aumento da reutilização de artefatos testados, certificados e organizados, as organizações podem obter melhorias no custo, tempo e qualidade dos sistemas desenvolvidos [37].

2.1.2. Definição

Existem várias definições de reutilização de software na literatura. Segundo Peter Freman, reuso de software é a utilização de qualquer informação que o desenvolvedor precisa no processo de criação de software [39]. Frakes & Isoda definem reutilização de software como o uso de artefatos e conhecimento de um sistema existente para construir outros [44]. Basili e Rombach definem como

sendo a utilização de qualquer coisa associada a um projeto de software, incluindo o conhecimento [43]. De acordo com Ezran, reuso de software é uma pratica sistemática para desenvolver software a partir de blocos, com isso as semelhanças nos requisitos e arquitetura entre aplicações podem ser exploradas para aumentar a produtividade, qualidade e performance nos negócios [39].

Neste trabalho, será adotada a definição de Krueger [45]:

"Reutilização de software é um processo de criação de sistemas de software a partir de softwares existentes ao invés de construí-lo do zero."

2.1.3. Benefícios

Como discutido anteriormente, a principal motivação para reusar artefatos de softwares é reduzir o tempo e esforço requeridos na construção de um sistema. A qualidade do sistema é melhorada reusando artefatos previamente testados e validados, reduzindo também o tempo e esforço gasto na manutenção [45]. Alguns dos principais benefícios obtidos com a reutilização são listados a seguir [38] [46] [47] [48]:

- **Aumento da produtividade:** Ganho na produtividade é obtido com a diminuição do código desenvolvido, gastando menos tempo com teste, analise e projeto e evitando redundância de esforço de desenvolvimento dos engenheiros;
- **Redução do custo de manutenção:** Reutilizando artefatos com qualidade alta, os defeitos são reduzidos. Com isso, o esforço com manutenção são centralizados e correções e atualizações de um artefato podem ser propagadas facilmente para os clientes;
- **Aumento da qualidade:** correções de erros são acumuladas de reutilização para reutilização. Isso resulta em uma melhoria na

qualidade do artefato reutilizado se o componente fosse desenvolvido e utilizado apenas uma única vez;

- **Confiabilidade:** utilizando artefatos bem testados aumenta a confiabilidade de um sistema, pois o uso de um artefato em diversos sistemas aumenta a chance da detecção de erros, reforçando a confiança nesse artefato;

- **Redução do *time-to-market***: o sucesso ou fracasso de um produto de software é muitas vezes determinado por seu *time-to-market*. Usando artefatos reutilizáveis pode resultar em uma redução desse tempo;

- **Documentação:** reutilizando artefatos de software reduz a quantidade de documentos que devem ser escritos, mas aumenta a importância do que é escrito. Assim, apenas a estrutura geral do sistema e novos artefatos desenvolvidos têm que ser documentados;

- **Redução da equipe de desenvolvimento**: algumas equipes grandes sofrem com problemas de comunicação. Dobrar a equipe de desenvolvimento não significa dobrar a produtividade. Se muitos artefatos de software podem ser reutilizados, então sistemas podem ser desenvolvidos com equipes menores, melhorando a comunicação e aumentando a produtividade.

Diante dos benefícios apresentados, a conclusão é que a reutilização de software proporciona uma vantagem competitiva para a empresa que adotá-la. Algumas experiências relevantes de reutilização de software podem ser encontradas na literatura [39] [49] [50] [51]. Um interessante estudo que relaciona vantagens, sucessos, fracasso, mitos e inibidores é apresentado no livro C.R.U.I.S.E [15].

2.2. Componentes de Software

Devido a área de desenvolvimento de software baseado em componente (DSBC) ser relativamente recente, o conceito exato de componente de software não é um consenso na academia. Cada grupo de pesquisa o caracteriza, de acordo com o seu próprio contexto [35]. Desde o primeiro workshop de DSBC, em 1998, em Kyoto, varias definições tem sido apresentadas. No livro de Heineman [52], um grupo formado por especialistas em DSBC relata que depois de dezoito meses foi atingindo um consenso sobre a definição de um componente de software.

De acordo com esses especialistas, um componente de software é um elemento de software (uma seqüência de instruções que descrevem computações a serem executadas por uma máquina) que está de acordo com um modelo de componente (definição específica de padrões de interação e composição utilizados pelos componentes) e pode ser implantado independentemente sem alterações.

Atualmente, a definição de componente mais próxima de um consenso dentro da academia, e que será adotada neste trabalho, é a de Szyperski [53]. Segundo ele: “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.”

Segundo Szyperski, os termos componente e objeto são muitas vezes usados com o mesmo intuito. Assim, é necessário deixar claro a diferença entre eles. Segundo Szyperski, um componente [53]:

- **Pode ser implantado independentemente**: um componente pode ser implantado separadamente do seu ambiente e de outros componentes, porem não pode ser parcialmente implantado;

- **Composição com terceiros**: para que um componente faça parte da composição de terceiros é necessário que ele seja autocontido e deve conter uma documentação que especifique claramente o que ele precisa e o que ele provê. Ou seja, um componente encapsula a sua implementação e interage com o ambiente onde está inserido através de interfaces bem definidas;

- **Não apresenta estado observável externamente**: um componente não pode ser distinguido por cópias dele mesmo. Ou seja, componentes não são instanciados como objetos o são.

Os conceitos de instancia, encapsulamento e identidade se referem a objetos. Em contrapartida aos conceitos de componentes, um objeto possui [53]:

- **Unidade de instanciação com um identificador único:** o identificador aponta unicamente para o objeto independente de mudanças em seu estado durante todo o seu ciclo de vida.

- **Possui estado observável externamente:** o objeto permite a observação de seu estado interno através de sua classe;

- **Encapsula estado e comportamento**: a classe do objeto permite o encapsulamento de seu estado e seu comportamento.

2.3. Métodos de Desenvolvimento Baseados em Componentes

Todo processo de desenvolvimento de software necessita de uma abordagem sistemática, no desenvolvimento de componentes não é diferente. Por isso, usar métodos ou processos facilita a criação de sistemas complexos. Existem vários métodos de DSBD na literatura, entre eles se destacam: *Catalysis* [17], PECOS (Pervasive Component Systems) [18] e *UML Components* [16]. Uma visão geral desses métodos será mostrada nas próximas seções.

2.3.1. Catalysis

O método *Catalysis* foi desenvolvido na universidade de Brighton, Inglaterra, por D'Souza e Wills [17], É um método que integrada varias técnicas para desenvolver Sistemas Distribuídos Orientados a Objetos. *Catalysis* é um método de DSBC que cobre todas as fases de desenvolvimento de um componente, desde a especificação até a implementação. Este método é baseado num conjunto de princípios, entre eles pode-se destacar [17]:

- **Abstração:** esse princípio guia o desenvolvedor na busca de aspectos essenciais do sistema, dispensando detalhes que não são relevantes para o contexto do mesmo;

- **Precisão**: o principio da precisão tem como objetivo descobrir erros e inconsistências na modelagem. Sucessivos refinamentos entre as transições de uma fase para outra auxiliam na obtenção de artefatos mais precisos e propensos à reutilização;

- **Componentes *"plug-in"***: reutilizar componentes existentes para criar novos.

- **Lei de reutilização**: a principal lei de reutilização do método *Catalysis* é não reutilizar código sem reutilizar os modelos de especificações desse código.

O processo de desenvolvimento de software *Catalysis* se divide em três níveis [54]: **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.

2.3.2. PECOS (Pervasive Component Systems)

PECOS é um método de DSBC voltado para sistemas embarcados, no qual existe uma dificuldade em atender os requisitos não-funcionais do sistema (potência consumida, tamanho do código, performance entre outros) [18].

O principal objetivo desse método 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 pontos-chave para respeitar as características dos sistemas embarcados [18]:

- **Modelo de Componente para sistemas embarcados** levando em consideração a especificação de seu comportamento, propriedades não-funcionais e limitações;

- **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;

- **Ambiente de Composição para montar aplicações embarcadas a partir de componentes**, validar limitações de requisitos funcionais e não-funcionais, gerar um executável para o dispositivo e monitorar sua execução. Esse ambiente deve ser leve para instalar, rodar e testar aplicações de sistemas com recursos limitados, permitindo o seu gerenciamento.

2.3.3. UML Components

O *UML Components* é um processo para a especificação de software baseado em componentes onde os sistemas são estruturados em quatro camadas distintas: (i) interface com o usuário, (ii) comunicação com o usuário, (iii) serviços de sistemas e (iii) serviços de negócios [16]. Esta estrutura representa a arquitetura do sistema. Além disso*, UML Components* propõe a utilização de UML para modelar todas as fases do desenvolvimento do sistema.

A linguagem UML permite que qualquer elemento tenha um estereótipo anexado, facilitando a extensão da linguagem, que é encapsulado pelo símbolo "<< >>" permitindo ao usuário criar novos elementos de modelagem. UML Components estende a linguagem UML adicionando seis novos estereótipos que ajudam na modelagem [16]:

- **Estereótipo <<type>>:** usado para representar uma entidade 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.

- **Estereótipo <<datatype>>:** possui a mesma representação de Type, mas não contem relacionamento, geralmente é usado para representar atributos.

- **Estereótipo <<interface type>>:** representa uma interface em nível de especificação. Por convenção, o nome da entidade 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.

- **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.

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

- **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

Para melhor compreender o processo, o fluxo geral de atividades é mostrado na figura a seguir.

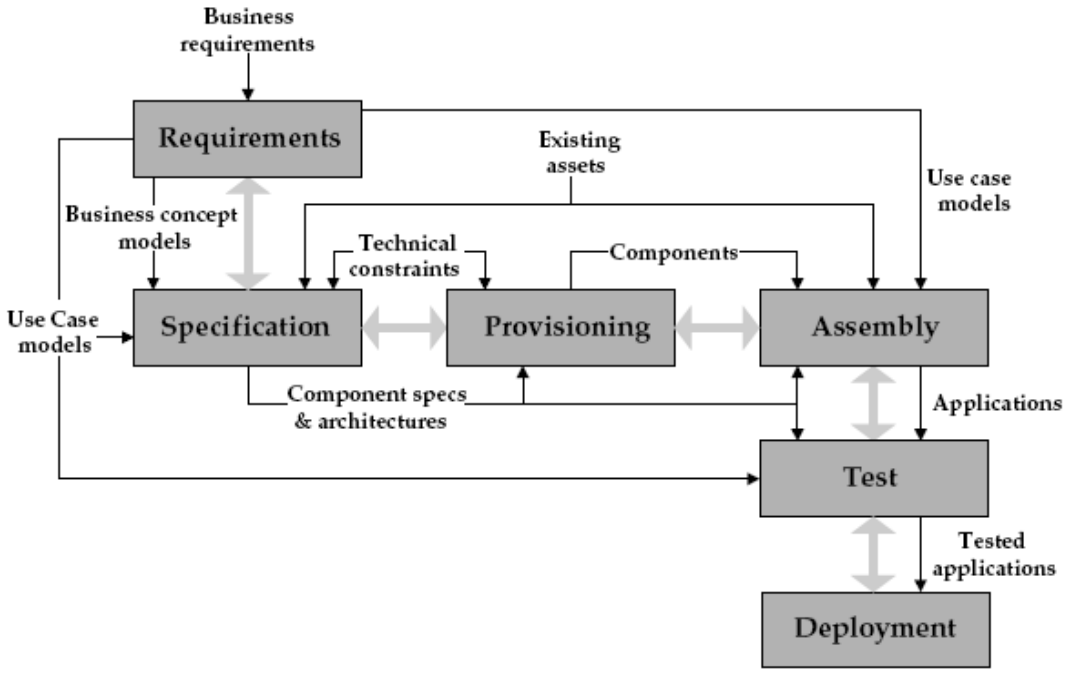


Figura 2 - Fluxo de atividades de UML *Components* [16].

Comparando a figura 2 com o fluxo de atividades do RUP [19], Requisitos(*Requirements),* Teste(*Test) e* Implantação(*Deployment)* correspondem, com pequenas adaptações, às atividades com os mesmos nomes no RUP [16]. Especificação(Specification), Adequação (*Provisioning), e* Codificação*(Assembly)* substituem as atividades de Análise e Projeto e Implementação do RUP, mais detalhes sobre essas atividades serão mostrados na seção 3.3.5.

2.3.4. Escolha do método de DSBC

Analisando os métodos de desenvolvimento baseado em componentes apresentados anteriormente, o mais adequado para o desenvolvimento do *framework* proposto é o *UML Components* pelos seguintes motivos:

i. **Material disponível**: Uml Components possui um livro especificando detalhadamente todas as atividades do processo com exemplos explicativos que facilita o aprendizado;

ii. **Integração com UML**: Todo a especificação do processo é baseada em UML;

iii. **Flexibilidade:** ao contrário dos processos PECOS, elaborado para o domínio de sistemas embarcados, e *Catalysis*, voltado para o desenvolvimento de sistemas distribuídos, Uml *Components* pode ser usado sem dificuldades em qualquer domínio de aplicação;

iv. **Aceitação**: o método é bem aceito e vem sendo usado na industria e na academia [54].

Por esses motivos, o *UML components* se mostrou o mais adequado para este trabalho e será explicado em mais detalhes na seção 2.3.5.

2.3.5. Detalhamento do processo UML *Components*

A figura 3 mostra em detalhes a principal parte do fluxo de atividades de UML *Components*.

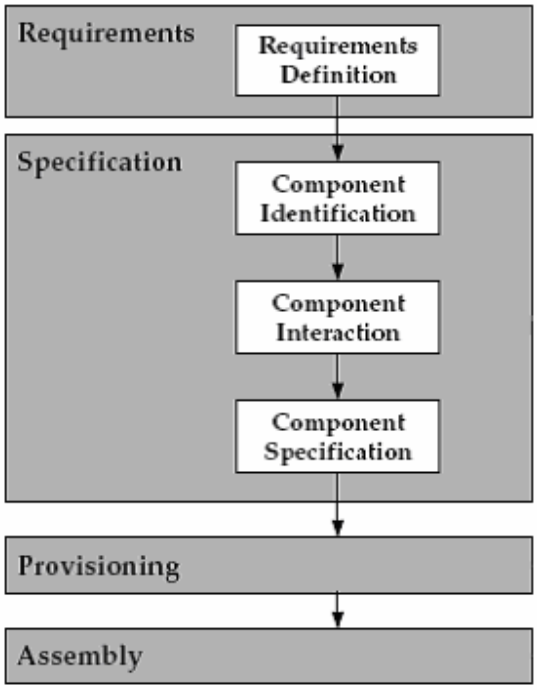


Figura 3 - Detalhamento do fluxo de atividades de *UML Components* [16].

A principal atividade é Especificação(*Specification*), ela recebe como entrada o modelo de casos de uso e o modelo conceitual do negócio [16]. O modelo conceitual de negocio não é um modelo de software, é um modelo de informação que existe no domínio do problema. Seu principal objetivo é identificar capturar conceitos e identificar relacionamentos entre eles. Também podem ser usadas informações previamente existentes, como, sistemas legados, pacotes de código, banco de dados e restrições técnicas. Essa atividade tem como saída um conjunto de especificações e arquiteturas dos componentes. A especificação do componente contém as interfaces que os componentes suportam e dependem. A arquitetura do componente mostra como os componentes interagem uns com os

outros. Todos os detalhes e artefatos gerados nesta fase serão apresentados no capítulo 5.

Estas saídas serão usadas pela atividade de Adequação (*Provisioning*). Esta atividade garante que os componentes necessários estarão disponíveis, sejam eles comprados, desenvolvidos do zero, integrados com outros componentes ou reutilizados outros sistemas. A atividade também inclui testes unitários dos componentes antes da Codificação (*Assembly*) [16].

Por fim, a atividade Codificação (*Assembly*) integra todos os componentes com os possíveis artefatos de softwares existentes, utilizando uma interface com o usuário apropriada, gerando uma aplicação que atenda aos requisitos levantados [16].

3. Capítulo 3 – Service Oriented Architecture

A grande maioria das empresas possui soluções rodando em plataformas heterogêneas que foram desenvolvidas usando diferentes tecnologias. Como os recursos (pessoas, tempo, maquinas, investimentos) são escassos, as empresas não podem simplesmente jogar fora as aplicações existentes. Por isso, Arquitetura Orientada a Serviços, do inglês *Service Oriented Architecture* (SOA) está se tornando cada vez mais popular porque permite reutilizar aplicações e prover a interoperabilidade entre diferentes tecnologias.

Neste contexto, este capítulo apresenta uma introdução sobre Arquitetura Orientada a Serviços. A seção 3.1 mostra os conceitos básicos relativos a SOA. Na seção 3.2 serão apresentados os principais objetivos e vantagens em desenvolver aplicações com arquitetura orientada a serviços. Por fim, a seção 3.3 mostra as tecnologias usadas para desenvolvimento de aplicações SOA e detalhes sobre *Web Services*.

3.1. Fundamentos

O termo *"service oriented architecture"* é muitas vezes confundido com *"service oriented computing"*, entretanto é muito importante deixar clara a distinção entre os mesmos.

Segundo Thomas Erl, *"service oriented computing"* é um conceito bastante amplo que representa uma nova geração de plataforma de computação distribuída. Engloba vários elementos como princípios de *design*, padrões de projeto, linguagens de programação, hardware, *framework*s, processos e estratégias organizacionais. SOA é, basicamente, um modelo de arquitetura de software que beneficia a eficiência, agilidade e produtividade no desenvolvimento de aplicações e está alinhado com os objetivos de *"service oriented computing"* [20].

Uma arquitetura de software é um conceito abstrato que dá margem a uma série de definições. A definição usada pelo IEEE afirma que uma arquitetura de software trata basicamente de como os componentes fundamentais de um sistema se relacionam intrinsecamente e extrinsecamente [34]. Uma arquitetura orientada a serviços tem como componente fundamental o conceito de serviços.

Serviços, assim como componentes [53], são considerados blocos de construção independentes, os quais coletivamente representam um ambiente de aplicação [20]. Cada serviço é completamente autônomo em relação a outros serviços. Isto significa que cada serviço é responsável por seu próprio domínio, o que tipicamente significa limitar seu alcance para uma função de negócio específica (ou um grupo de funções relacionadas) [20].

Este enfoque de projeto resulta na criação de unidades isoladas de funcionalidades de negócio ligadas fracamente entre si. Isto é possível por causa da definição de uma estrutura padrão de comunicação. Devido à independência que esses serviços desfrutam dentro desta estrutura, a lógica de programação que encapsulam não tem necessidade de obedecer a nenhuma outra plataforma ou conjunto de tecnologias.

Devido ao não esclarecimento dos termos *"service oriented architecture"*, *"service oriented architecture"* e serviços, SOA tem sido utilizado em diferentes contextos e com diferentes propósitos. Por isso, dependendo do interlocutor, arquitetura orientada a serviços pode ter diferentes interpretações, conforme mostra a figura 4:

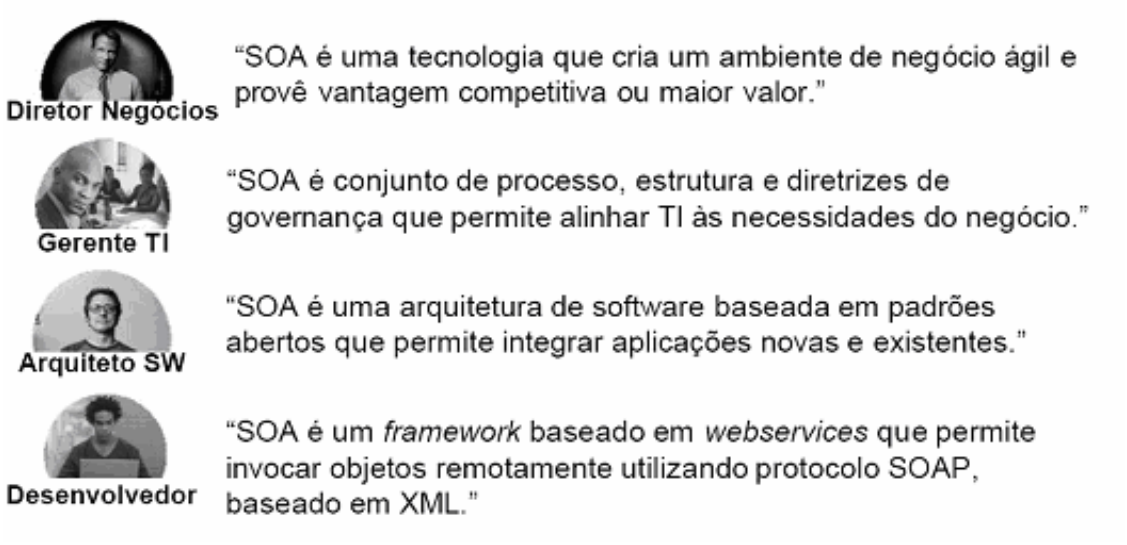


Figura 4 - Diferentes interpretações de SOA [32].

No contexto desse trabalho, a interpretação mais adequada é a do arquiteto de software, porem não é a mais correta. Para melhor entendimento da arquitetura do *framework* MoSOA e dos conceitos que serão apresentados nas próximas seções a melhor interpretação seria: Arquitetura Orientada a Serviços é uma arquitetura de software baseado em padrões abertos que permite desenvolver aplicações utilizando serviços existentes.

3.1.1. Objetivo e Vantagens

A arquitetura orientada a serviços tem como principal objetivo o reuso intenso dos seus componentes (serviços), para que em médio prazo, a tarefa do desenvolvimento de uma aplicação seja primordialmente a composição e coordenação dos serviços já implementados, aumentando o reuso e diminuindo o dispêndio de recursos. Segundo alguns autores, SOA pode ser vista como uma evolução do desenvolvimento baseado em componentes [33].

Segue alguns benefícios em utilizar uma arquitetura orientada a serviços [6] [20] [33]:

- **Reuso "Caixa-preta"**

O reuso “caixa-preta” visa eliminar a necessidade do programador ter conhecimento da implementação de um determinado componente de software que fará parte do processo de reuso. O reuso caixa-preta se dá através da descrição de interfaces ou contratos bem definidos que devem ser respeitados durante o desenvolvimento. O esforço é sempre usado na nova implementação e não no entendimento da implementação do componente.

- **Interoperabilidade**

O fenômeno da Web 2.0 apresentou uma nova realidade para os desenvolvedores de sistemas. Por ser uma rede de escala gigantesca existe uma infinidade de serviços que podem ser acessados através da Internet. Usando SOA é possível acessar serviços em diferentes plataformas de hardware, implementados em diferentes linguagens de programação, usando protocolos bem definido.

- **Baixo acoplamento**

Cada atividade (função de negocio) é implementada como um serviço, ou seja, um componente independente que poderá ser utilizado quantas vezes forem necessárias em diversas partes do sistema. Assim, SOA é uma arquitetura baseada em componentes, onde cada componente preserva a sua independência.

3.2. Tecnologias

Segundo Tomas Erl “Você não precisa de *Web services* para construir aplicativos SOA! Essas palavras você escutará muitas vezes quando se tenta explicar os princípios da arquitetura orientada a serviços. Mas utilizar *Web services* para implementar SOA é uma excelente idéia.”[6]. Muitas pessoas pensam que *Web service* e SOA são sinônimos e que é impossível desenvolver aplicações com arquitetura orientada a serviços sem *Web services*. Isso não é

verdade, pois é possível implementar aplicações SOA usando tecnologias como Java RMI, CORBA ou DCOM [20], mas essas tecnologias apresentam algumas dificuldades. CORBA tem o problema de possuir protocolos complexos. DCOM é uma tecnologia totalmente proprietária e dependente da plataforma. Java RMI é muito ligada a Java e dificilmente se comunica com outras linguagens de programação.

Por esses motivos Web Services se tornou popular, pois tenta resolver esses problemas criando protocolos de comunicação abertos que possam ser facilmente integrados a diferentes plataformas e possui dois requisitos fundamentais:

- comunica-se via protocolos Internet;
- envio e recebimento de dados em formatados de documentos XML.

3.2.1. Web Services

Em 2000, a World Wide Web Consortium (W3C) [21] aceitou a submissão do Simple Object Access Protocol (SOAP) e estabeleceu um formato de mensagem baseado em XML para comunicação entre aplicações via protocolo HTTP. Por ser uma tecnologia aberta, SOAP é uma alternativa aos protocolos proprietários, tais como CORBA e DCOM [20].

Em 2001, o W3C publicou a especificação WSDL, este padrão forneceu uma linguagem para descrever a interface dos *web services*. Posteriormente foi complementada pela especificação Universal Description, Discovery and Integration (UDDI), que proporcionou um mecanismo padrão para a descoberta dinâmica de descrições de serviço. A figura 5 mostra o relacionamento entre os padrões UDDI, WSDL, SOAP:

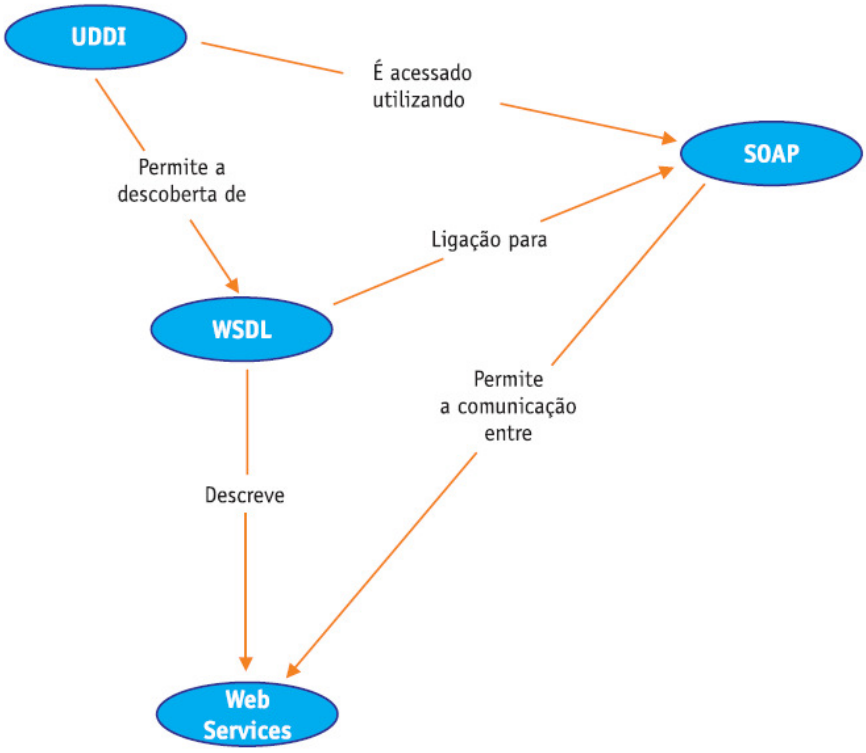


Figura 5 - Relacionamento entre padrões [35].

A estrutura W3C para *web services* está fundamentada em três especificações XML fundamentais [21]:

- SOAP – Simple Object Access Protocol;
- WSDL – Web Services Definition Language;
- UDDI – Universal Description, Discovery and Integration.

3.2.1.1. Web Services Definition Language (WSDL)

Web services devem ser definidos de forma consistente para que possam ser descobertos e acessados através de outros serviços e aplicações. A WSDL é uma especificação que fornece a linguagem para a descrição e definições de serviços:

- Abstrato + Concreto = Definição de Serviço

- Definição de Serviço + Definições Complementares = Descrição de Serviço.

WSDL pode conter os seguintes construtores primários:

- Interface;
- Message;
- Service;
- Binding;

A figura 6 mostra o relacionamento entre a definição e descrição de serviços e os principais construtores de um de um documento WSDL.

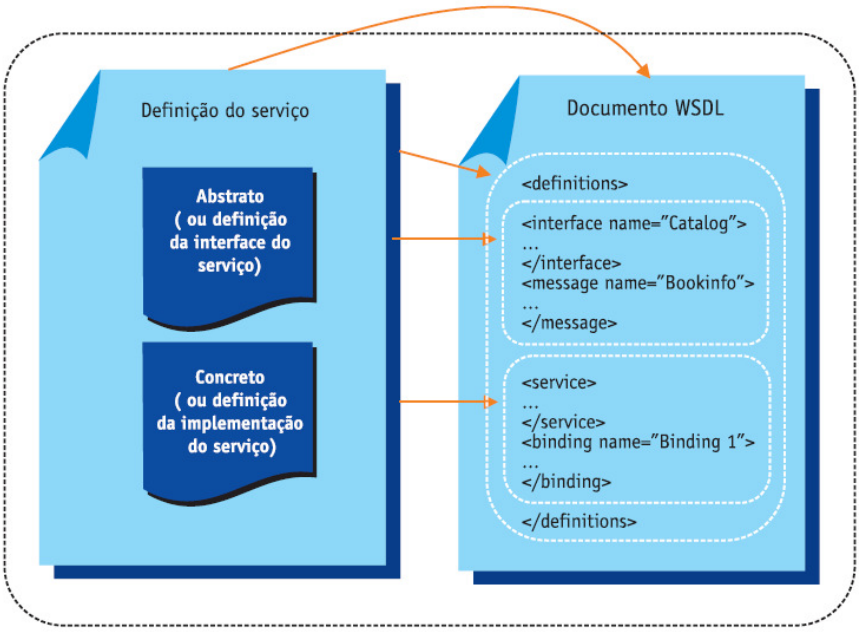


Figura 6 - Definição do serviço em um web service [35].

Os construtores *Interface* e *Message* representam a definição da interface do serviço e os outros dois fornecem os detalhes de implementação. Na figura 4.3 o elemento *definitions* age como um *conteiner* para a definição do serviço.

Interfaces de *web services* são representadas por elementos *Interface*. Estes construtores contêm um conjunto de operações lógicas relacionadas. Comparando com uma arquitetura baseada em componentes, o elemento *Interface* equivale à interface de um componente e o elemento *operation* a um método. O quadro abaixo mostra um exemplo de como os construtores são usados em um WSDL.

```
<definitions>
            <message name="InfoLivro">

                  ...

            </message>
            <interface name="Catalogo">
                  <operation name="getLivro">

                        <input name="Msg1" message="InfoLivro" />

                  </operation>
            </interface>
      </definitions>
```

O elemento *operation* consiste de um grupo de mensagens de entrada e saída relacionadas. A execução de um *operation* requer a troca destas mensagens entre o solicitante e o provedor do serviço. Mensagens *operation* são representadas por construtores *message* que são declarados dentro dos elementos *definitions*. Os nomes das mensagens são referenciados nos elementos filho de *operation*. O quadro abaixo mostra como um elemtento message pode ser descrito em um documento WSDL.

```
<definitions>
            <message name="InfoLivro">
                  <part name="Titulo" type="xs:string">
                        MoSOA Framework
                  </part>
                  <part name="Autor" type="xs:string">
                        Vitor Braga
                  </part>
            </message>
      </definitions>
```

Um elemento *message* pode conter um ou mais parâmetros de entrada (input) ou saída (output) que pertencem a um *operation*. O elemento *part* define esses parâmetros como um conjunto nome/valor (name/value) e o tipo de dado associado. Em uma arquitetura baseada em componentes, um *part* equivale a um parâmetro ou retorno de um método.

Um documento WSDL pode estabelecer detalhes de implementação. Dentro do WSDL, o elemento *service* contem um ou mais pontos de término. Estes pontos de término, elementos *endpoint*, fornecem informações como localização do serviço e protocolos utilizados. O quadro abaixo mostra os elementos endpoints em um WSDL.

```
<definitions>
      <service name="Servico1">
            <endpoint name="Endpoint1" binding="Binding1">
                  Detalhes da implementação ...
            </endpoint>
      </service>
</definitions>

<definitions>
      <message name="InfoLivro">
            <part name="Titulo" type="xs:string">
                  Framework MoSOA
            </part>
            <part name="autor" type="xs:string">
                  Vitor Braga
            </part>
      </message>
</definitions>
```

Depois de descrever como um *web service* pode ser acessado, é preciso definir os requisitos de chamada para cada *operations*. O quadro abaixo mostra como isso é feito.

```
<definitions>
      <service>
            <binding name="Binding1">
                  <operation>
                        <input name="Msg1" message="InfoLivro" />
                  </operation>
            </binding>
      </service>
```

Os elementos *binding* associam aos *operations* informações de formato de mensagem e protocolo.

Os elementos do WSDL vistos podem ser resumidos assim:

- **Interfaces**: representam interfaces de serviço e podem conter diversos *operations*;
- **Operations**: representam uma função web service e podem referenciar várias *messages*;
- **Messages**: representam um conjunto de parâmetros input ou output e podem conter vários *parts*;
- **Parts**: representam parâmetros de dados *operation*;
- **Service:** elementos *services* são coleções de *endpoint*
- **Endpoint:** elementos *endpoint* possuem dados como localização, informação de protocolo e mensagem;
- **Binding*:*** elementos *binding* associam aos *operations* informações de formato de mensagem e protocolo.

3.2.1.2. Simple Object Access Protocol (SOAP)

A especificação SOAP estabelece um formato padrão de mensagem que consiste em um documento XML capaz de hospedar dados RPC e centrados em documentos (*document-centric*). Com o WSDL estabelecendo um formato de descrição padrão para aplicações, o uso do formato de mensagem *document-centric* é mais comum.

SOAP estabelece um método de comunicação baseado em um modelo de processamento que está relativamente alinhado com a arquitetura de web services descrita anteriormente. Para entender como as mensagens são manipuladas é

importante compreender alguns termos como nós SOAP, tipos de nós e estrutura das mensagens.

Um nó SOAP representa o processamento lógico responsável pela transmissão, recebimento e realização de uma série de tarefas sobre mensagens SOAP. Uma implementação de um nó SOAP é tipicamente específica à plataforma e é normalmente rotulada como um SOAP server (servidor) ou SOAP listener (ouvinte) [6]. Existem também variações especializadas, tais como SOAP routers. A figura 7 mostra os tipos de nós SOAP ao longo de uma rota de mensagem.

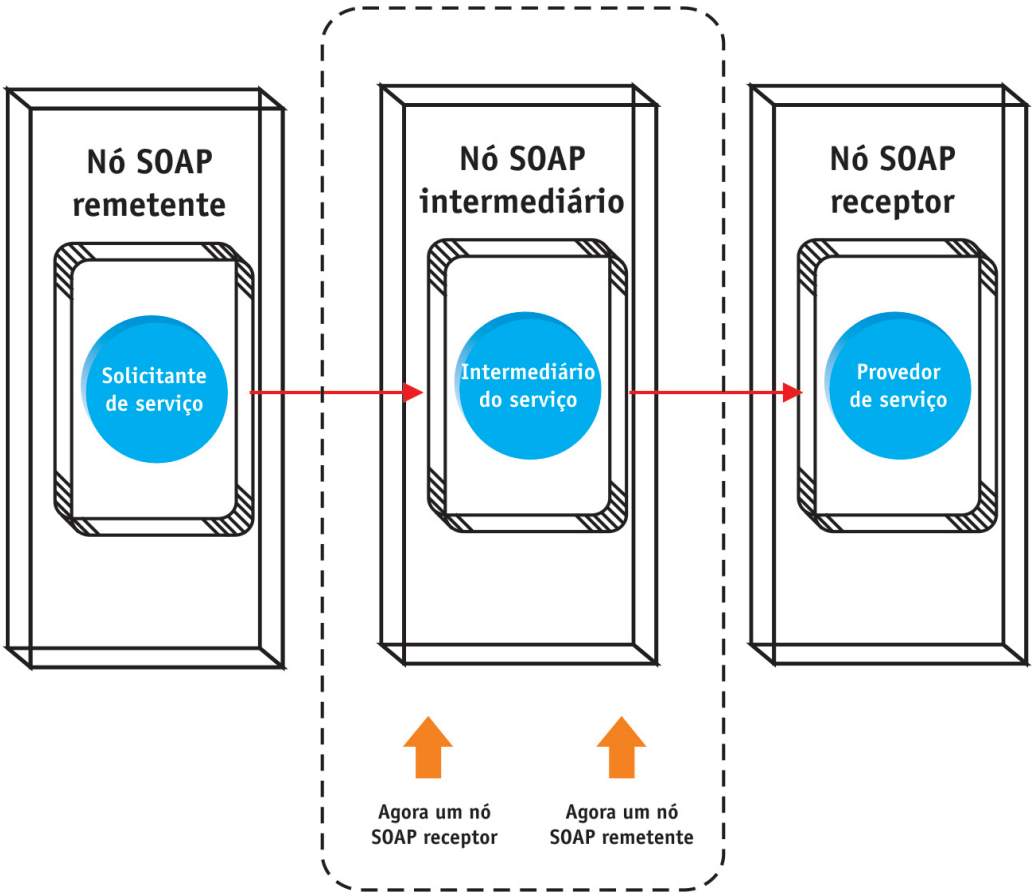


Figura 7 - Nós SOAP ao longo de uma rota de mensagem [35].

Nós SOAP podem existir como emissores iniciais, intermediários e receptores finais. Nós SOAP realizando tarefas de envio e recebimento são chamados de

SOAP emissores e SOAP receptores respectivamente. Um nó SOAP intermediário age como receptor e emissor durante o processamento de uma mensagem SOAP.

Assim como nos papéis atribuídos aos web services, o mesmo nó SOAP pode assumir comportamentos diferentes dependendo da sua posição dentro da rota da mensagem e do estado da atividade de negócio atual. Por exemplo, um nó SOAP como um emissor inicial ao transmitir uma mensagem pode, em outro momento, assumir o papel de receptor final e também receber uma resposta enviada por um outro nó.

O recipiente da informação de uma mensagem SOAP é chamado de envelope SOAP. O quadro abaixo mostra uma estrutura típica de uma mensagem SOAP.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soapenvelope">
      <env:Header>
            ...
      </env:Header>

      <env:Body>
            ...
      </env:Body>

</env:Envelope>
```

O elemento raiz Envelope, que delimita o documento da mensagem, é composto por uma seção *Body* e uma área *Header* que é opcional. O cabeçalho SOAP é representado através do uso de um construtor *Header*, o qual pode conter um ou mais blocos ou seções. O uso mais comum de blocos *Header* ocorre em:

- Implementação de extensões SOAP;
- Identificação de alvos SOAP intermediários;
- Informação adicional sobre mensagem SOAP.

Enquanto uma mensagem SOAP trafega até o seu destino, intermediários podem acrescentar, remover ou processar as informações contidas no *Header*. Mesmo sendo parte opcional de uma mensagem SOAP, o uso da seção *Header*

para carregar informações é muito comum quando se trabalha com especificações *web services*.

A única parte de uma mensagem SOAP que é obrigatória é o *Body*. Esta seção age como um recipiente para os dados que são entregues pela mensagem. Esses dados são freqüentemente chamados de carga útil ou dados de carga útil. O quadro seguir mostra um exemplo do *Body* em uma mensagem SOAP.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
      <env:Header>
      ...
      </env:Header>

      <env:Body>
            <x:Book xmlns:x="http://www.examples.ws/">
                  <x:Title>
                        MoSOA: Um Frameowrk para desenvolvimento
                        de aplicações móveis orientada a Serviços.
                  </x:Title>
            </x:Book>
      </env:Body>
</env:Envelope>
```

3.2.1.3. Universal Description, Discovery,and Integration (UDDI)

Um componente fundamental da arquitetura orientada a serviço é o mecanismo pelo qual descrições *web services* podem ser descobertas por requisitantes potenciais. Para estabelecer esta parte de um *framework* baseado em *web service*, é necessário um diretório centralizado para hospedar as descrições dos serviços utilizados. Tal diretório pode se tornar uma parte integrada de uma organização ou de uma comunidade disponibilizada na Internet [6].

Universal Description, Discovery,and Integration (UDDI) é a padronização da descrição que é armazenada em tais diretórios, também conhecidas como registro. Um registro público de negócio é um diretório global de descrições de

serviços internacionais de negócios. Instâncias deste registro são hospedadas em um grupo de servidores UDDI dedicados. Esses registros UDDI são replicados automaticamente entre instâncias de repositórios. Algumas empresas também agem como registradoras de UDDI, permitindo que outros adicionem e editem seus perfis de descrições web service.

Registros privados são repositórios de descrições de serviços hospedados dentro de uma organização. Aqueles autorizados a acessar este diretório geralmente são parceiros de negócios externos. Um registro restrito a usuários internos só pode ser referenciado como registro interno. A figura 8 mostra a descrição de serviço centralizado em um UDDI privado.

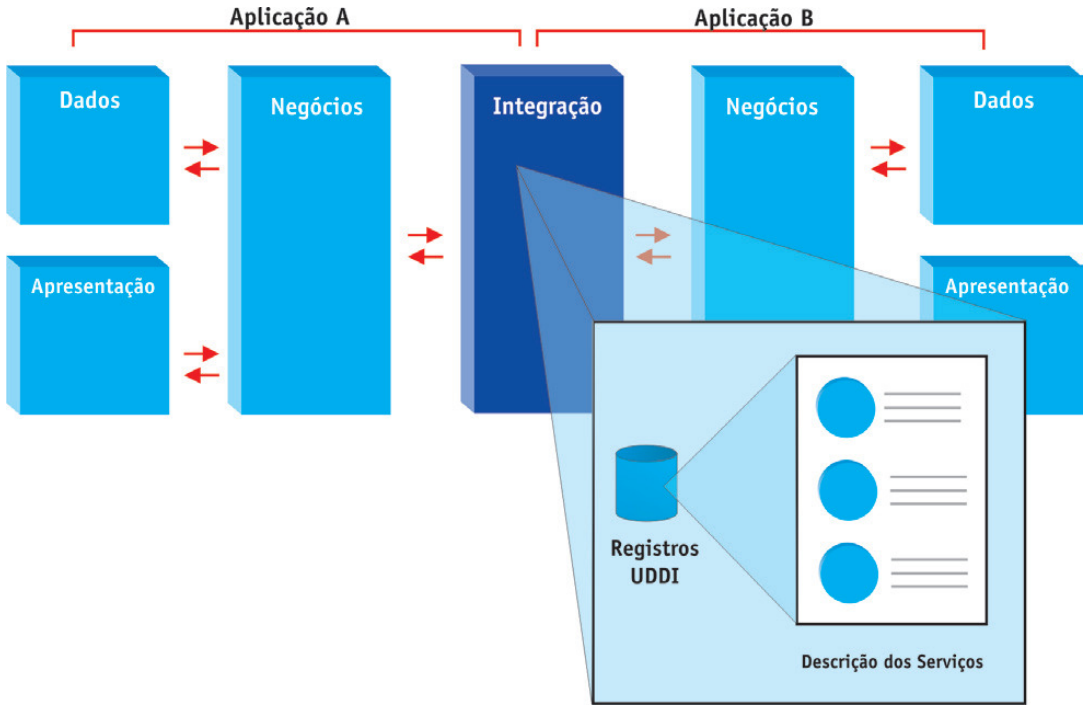


Figura 8 - Descrições de serviço centralizados em um registro UDDI privado [35].

O processo de descoberta pode ocorrer em várias situações dependendo da necessidade da informação de serviço:

- uma organização desejando estabelecer novas relações de negócio para transações *on-line* pode procurar por (e comparar) parceiros apropriados utilizando um registro público de negócio;

- um arquiteto que esteja projetando uma nova aplicação *e-Bussiness* pode antes querer pesquisar a disponibilidade de lógica de programação genérica dentro da organização. Pela leitura de descrições de serviço existentes, podem ser descobertas oportunidades de reuso;

- o mesmo arquiteto pode também querer vender *web service* para terceiros, fornecendo lógica de aplicação pré-construída que pode ser incorporada (localmente ou remotamente) dentro de outra aplicação;

- um desenvolvedor que está construindo novos serviços precisará acessar as definições de interface para serviços existentes. O registro interno poupa ao desenvolvedor a preocupação de saber se a interface que está sendo incorporada é ou não a mais atual.

É importante salientar que não existe uma relação de formato entre UDDI e WSDL. Um registro UDDI fornece apenas meios para apontar para as definições da interface de serviços.

4. Capítulo 4 - Java Micro Edition

Java Micro Edition (Java ME) é um conjunto de tecnologias e especificações para criar uma plataforma de desenvolvimento de software para dispositivos com recursos limitados tais como celulares, assistentes digitais (PDA) e set-up boxes. É uma versão reduzida da máquina virtual e da API de Java projetada para funcionar nesses dispositivos.

Atualmente, Java ME é a tecnologia mais usada no desenvolvimento de aplicações para dispositivos móveis, por este motivo é a tecnologia alvo deste trabalho [3].

Neste contexto, este capítulo apresentada a tecnologia Java ME que servirá de base para implementação do *framework* proposto. A seção 4.1 apresenta os detalhes da plataforma. Na seção 4.2 apresenta o processo para construção de uma aplicação simples na plataforma Java ME.

4.1. Plataforma Java ME

Java é uma linguagem de programação que permite o desenvolvimento de aplicações para diversas plataformas, desde dispositivos pequenos, como telefones celulares, até computadores de grande porte. Devido a essa característica, a linguagem é estruturada nos seguintes ambientes de desenvolvimento:

- **Java Plataform, Standard Edition (Java SE) [23]**: voltado para execução em computadores pessoas e servidores. Pode-se dizer que essa é a plataforma principal, pois serve como base para as outras.

- **Java Plataform, Enterprise Edition (Java EE) [24]:** como o nome já diz, é voltada para o desenvolvimento de aplicações corporativas. contém bibliotecas especialmente desenvolvidas para o acesso a servidores, a

sistemas de e-mail e a banco de dados. Foi desenvolvido para suportar uma grande quantidade de usuários simultâneos.

- **Java Plataform, Micro Edition (Java ME):** é o ambiente de desenvolvimento para dispositivos móveis ou portáteis, como telefones celulares e assistentes digitais (palmtops, smartphones).

Existe também o Java Card [25], voltada para dispositivos embarcados com limitações de processamento e armazenamento, como smart cards, e o Java FX [26], lançada oficialmente em 2007 para desenvolvimento de aplicações multimídia. A figura 9 mostra a estrutura geral da plataforma Java.

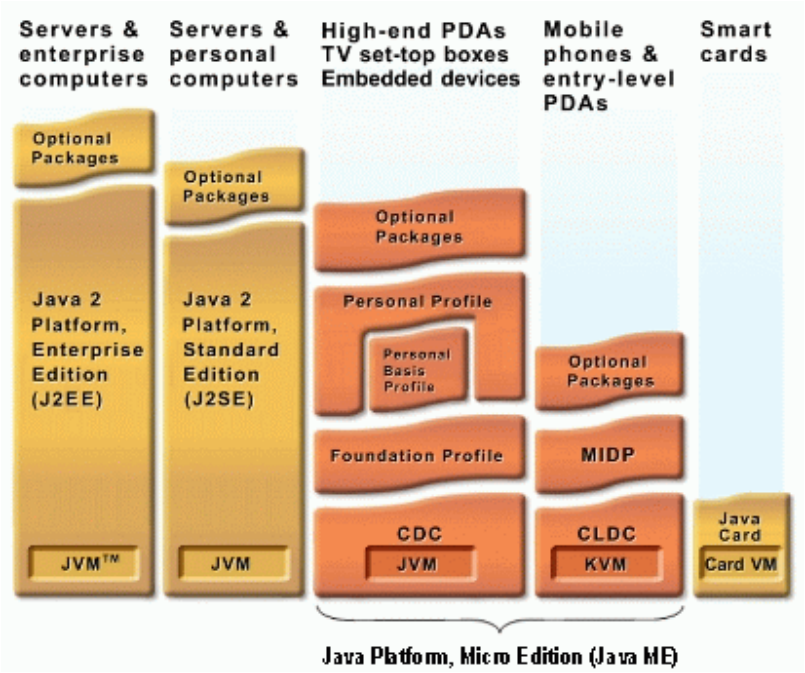


Figura 9 - Visão geral da plataforma Java [3].

Java ME é dividido em configurações, perfis e API's opcionais [27]. Existem dois tipos de configurações, o CLDC (Connected, Limited Device Configuration), que rege as configurações para aparelhos pequenos como celulares e o CDC (Connected Device Configuration) o que rege as configurações para aparelhos com maior capacidade computacional. Um perfil é uma extensão de uma configuração. No topo, existem as API´s opcionais, não especificadas nos perfis,

que oferecem funcionalidades especificas como bluetooh API(JSR 082) [28], Web Services API(JSR 172) [22], Mobile Media API (JSR 135) [29] e 3D API (JSR 184)[30].

As seções a seguir mostram mais detalhes dos conceitos de perfil e configuração.

4.1.1. Configuração de Dispositivo Conectado (CDC)

Os dispositivos que fazem parte do CDC possuem uma arquitetura de 32 bits, têm, no mínimo, dois megabytes de memória disponível. Alguns exemplos destes dispositivos são: set-top boxes, sistema de navegação e posicionamento e pontos de venda (PDVs). Suas principais características são:

- Mínimo de 512 kilobytes de memória para executar a maquina virtual Java;
- Mínimo de 256 kilobytes para alocação de memória em tempo de execução (memória heap);
- Conectividade de rede com largura de banda persistente e alta (WLAN, WiFi).

4.1.2. Configuração de Dispositivo Conectado Limitado (CLDC)

Os dispositivos fazem parte do CLDC possuem uma arquitetura de 16 ou 32 bits com memória reduzida, entre 128KB e 512KB, e são alimentados por bateria. Eles também utilizam uma conexão de rede sem fio de banda estreita, podem não apresentar interface com o usuário e implementam uma versão reduzida da máquina virtual Java. Alguns exemplos destes dispositivos são: pagers, PDAs e telefones celulares. As principais características do CLDC são:

- 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;
- Conectividade de rede, normalmente dispositivos sem fio com largura de banda baixa e acesso intermitente (CDMA, GSM).

4.1.3. Perfis

Perfil é uma extensão de uma configuração. Ele oferece bibliotecas para o desenvolvedor escrever aplicativos para um tipo em particular de dispositivo [core j2me]. Um exemplo de perfil é o MIDP (Mobile Information Device Profile). Ele é usado em conjunto com a CLDC e define um conjunto de APIs padrões para entrada e tratamento de eventos de interface com o usuário, persistência de dados, comunicação e cronômetros(timer), levando em consideração as limitações de tamanho de tela e memória dos dispositivos.

Existem outros perfis como 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) [27]. Atualmente, a maioria dos aparelhos celulares no mercado que suporta Java implementa o perfil MIDP, por isso, ele é alvo desse trabalho.

4.1.3.1. Mobile Information Device Profile (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 10, 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.

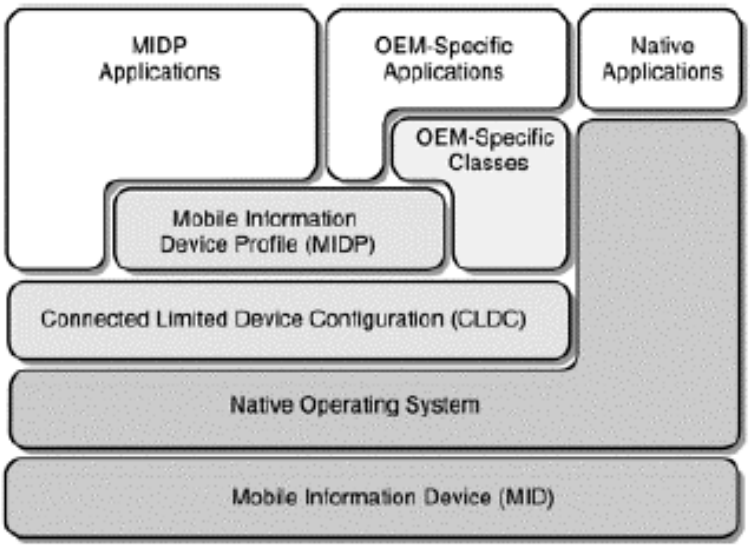


Figura 10 - Arquitetura MIDP [27].

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.

4.2. Criando aplicações em Java ME

Para entender corretamente a modelagem do *framework* é importante saber como implementar um MIDlet, compreender seus principais métodos, tratamento de eventos do usuário e como é mostrados informações na tela dos dispositivos.

4.2.1. MiDlets

Um MIDlet é um aplicativo Java projetado para ser executado em um dispositivo móvel [27]. Mais especificamente, um MIDlet tem como base a CLDC e o MIDP. Um Suíte consiste em um ou mais MIDlets empacotados em um arquivo JAR (Java Archive). A figura 11 apresenta o ciclo de vida de um MIDlet.

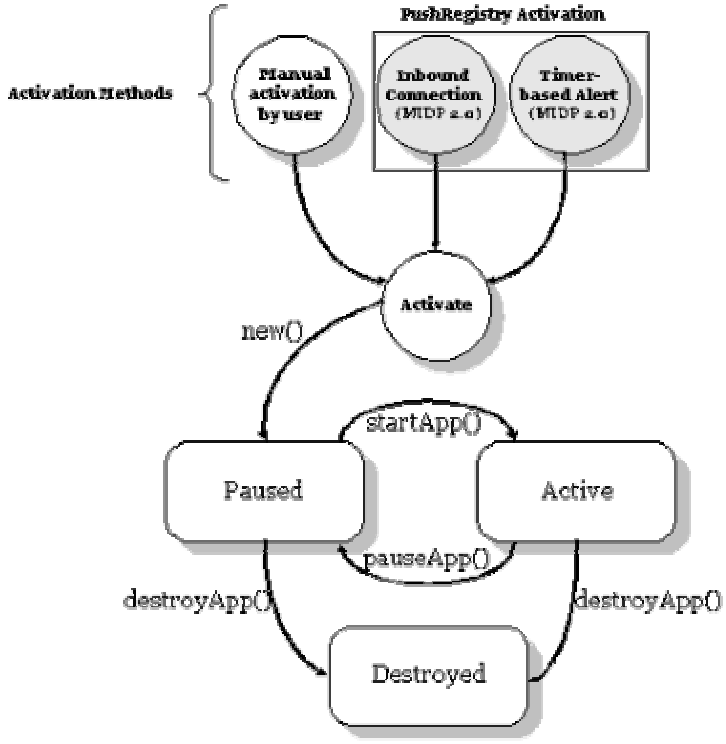


Figura 11 - Cliclo de vida de um MIDlet [31].

O Aplication Manager (AM) de cada dispositivo controla os aplicativos a serem instalados, onde serão armazenados e como serão executados. Assim que o MIDlet é invocado, o AM chama o método startApp(), o qual muda o midlet para o estado Active. Enquanto ela estiver executando o AM pode pausá-lo invocando o método pauseApp() no caso de uma chamada sendo recebida, ou SMS recebido, por exemplo, ou finalizá-lo, invocando o método notifyDestroyed().

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

- **abstract void destroyApp(boolean unconditional**): o AM solicita o desligamento do MIDlet;
- **abstract void pauseApp():** o AM solicita a pausa do MIDlet;
- **abstract void startApp():** o AM 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;

Os métodos abstratos (abstract) precisam ser implementados quando se está construindo um novo MIDlet. Esses métodos são a base da comunicação entre o AM e a aplicação Java ME.

A tela do dispositivo é representada por uma instância da classe Display, a qual é obtida pelo método getDisplay(), geralmente contida no método startApp(). Este objeto é usado para controlar o que está sendo mostrado na tela do dispositivo. 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;
- **void callSerially(Runnable r):** pede para que um objeto executável seja chamado após a repintura

O algoritmo básico para implementação de um MIDlet pode ser definido como:

1. Mostrar um Displayable;
2. Esperar por Ação do Usuário;
3. Decidir qual Displyable mostrar em seguida;
4. Repetir passo1;

As ações do usuário são gerenciadas por comandos (commands), os quais são adicionados a componentes visuais, as classes Screen ou Canvas.

Basicamente, duas classes são levadas em consideração quando se deseja fazer o tratamento de eventos do usuário: Command e CommandListener. O

objeto Command contém informações sobre um evento. O CommandListener é uma interface que serve como receptor para os Commands.

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.

4.2.2. Hello World

Para compreender os conceitos apresentados na seção anterior segue o código de um exemplo simples de um MIDlet que apresenta uma mensagem na tela e espera uma ação do usuário para finalizá-lo.

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

public class MyMidlet extends MIDlet implements CommandListener{
    Alert myAlert;//Tela de alerta
    Form myForm;// Tela principal
    Command okCommand;//comando ok
    Command exitCommand;//comando sair
    Display myDisplay;// display da aplicação
   public MyMidlet(){
        this.myAlert = new Alert(null, "OK Command", null, null);;
        this.myForm = new Form("Tela inicial");
        this.okCommand = new Command("OK",Command.OK,1);
        this.exitCommand = new Command("sair",Command.EXIT,1);
        this.myForm.addCommand(this.okCommand);
        this.myForm.addCommand(this.exitCommand);
        this.myForm.setCommandListener(this);
        this.myDisplay = Display.getDisplay(this);
    }
      //Método chamado quando aplicação é iniciada pelo usuário
    public void startApp() {
        this.myDisplay.setCurrent(this.myForm);
    }
    //Método chamado quando aplicação é pausada
    public void pauseApp() {}
      //Método chamado para destruir a aplicação
    public void destroyApp(boolean unconditional) {}
      //Método chamado para sair da aplicação corretamente
    public void exitMIDlet() {
        this.myDisplay = null;
        destroyApp(true);
        notifyDestroyed();
    }
      // método responsável por tratar os comandos do usuário
    public void commandAction(Command command, Displayable displayable) {
        if(command == this.okCommand){
```

```
            this.myDisplay.setCurrent(this.myAlert);
        }else if(command == this.exitCommand){
            exitMIDlet();
        }
    }

}
```

5. Capítulo 5 – MoSOA: Um *Framework* para Desenvolvimento de Aplicações Móveis Orientadas a Serviços

Como mencionado no capítulo 1, a proposta desse trabalho é desenvolver um *framework* para desenvolvimento de aplicações móveis orientadas a serviços aplicando um método de DSBC. O objetivo é elaborar uma arquitetura padrão para as aplicações, que facilite a inclusão de novos serviços, e fornecer componentes que auxiliem o desenvolvimento desse tipo de aplicações.

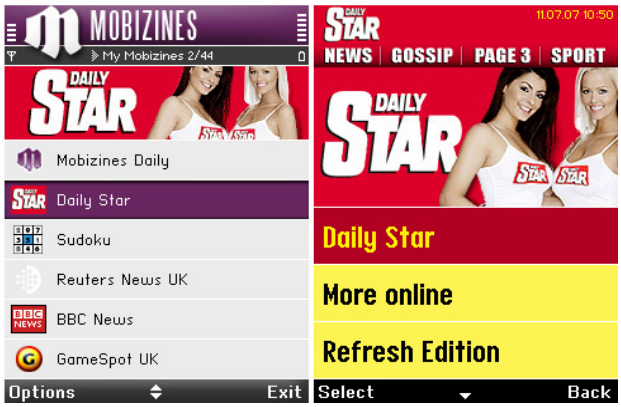
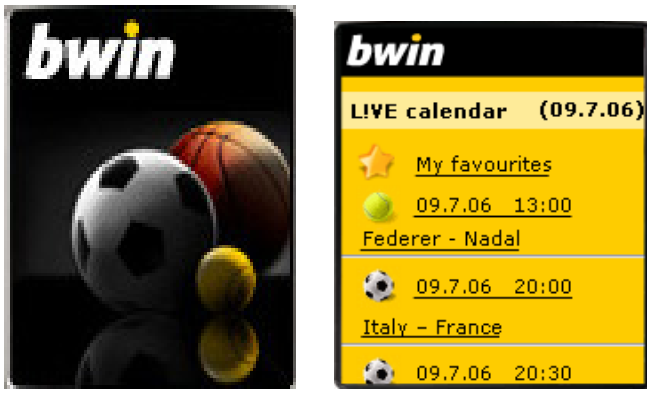
Neste capítulo será apresentado o MoSOA, um *framework* para desenvolvimento de aplicações móveis orientadas a serviços utilizando a tecnologia Java ME. Para construção do *framework* foi utilizado o método UML Components [16].

O MoSOA é um projeto open-source idealizado e implementado pelo autor deste trabalho. Todos os diagramas apresentados neste capítulo, código fonte e os documentos de requisitos e casos de uso estão disponíveis no site do projeto [59].

A seção 5.1 apresenta uma breve introdução do domínio das aplicações móveis orientadas a serviços. A seção 5.2 apresenta o escopo negativo do projeto. A seção 5.3 apresenta os detalhes de desenvolvimento do *framework* usando o método UML Components. Por fim, a seção 5.4 apresenta a aplicação construída para validação do *framework*.

5.1. Introdução

Para o contexto desse projeto, aplicações móveis orientadas a serviços são, resumidamente, aplicações para dispositivos móveis que utilizam serviços existentes, tais como: google maps[2], Amazon API[3], fickr API[4]. São aplicações com interface gráfica com usuário simples, onde, basicamente, possuem um menu inicial com uma lista dos serviços, telas para preencher parâmetros de acesso aos serviços e telas para mostrar o resultado das requisições. A tabela 1 mostra alguns exemplos de aplicações nesse domínio.

	**Mobizines** [55] software em Java ME para leitura de conteúdo móvel (blogs, ress). Utiliza serviços da BBC, Planet F1, Sporting Life, entre outros.
	**Bwin mobile** [56] é uma aplicação para fazer apostas esportivas on-line, consultar resultados de jogos, entre outras funcionalidades.

[2] http://maps.google.com.br/, servidor de mapas do google.
[3] http://www.amazon.com/gp/browse.html?node=3435361, API para utilizar serviços da Amazon
[4] http://www.flickr.com/services/api/, API para utilizar serviços do Flickr.

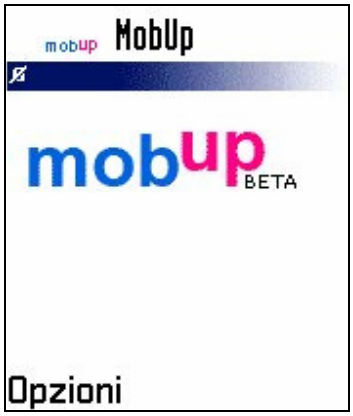
 	**Mobup** [57] é uma aplicação que controla *uploads* de fotos no Flickr a partir de um celular. Tem a possibilidade de tira fotos, adicionar título, tags e descrição nas fotos.

Tabela 1 - Exemplo de aplicações orientadas a serviços.

5.2. Escopo negativo

Construir um *framework* para o domínio de aplicações móveis orientadas a serviços não é uma tarefa fácil, pois existem diversas plataformas de desenvolvimento como Brew [62], Symbian [63], Windows Mobile[64], Java Me[3] e outras. Alem disso, existem vários protocolos e padrões de comunicação para acessar serviços (socket, tcp/ip, http, REST, *Web Services*). Por isso, o MoSOA possui as seguintes restrições:

- **Java ME como plataforma de desenvolvimento**: todos os componentes, mecanismo de tratamento de eventos do usuário e arquitetura foram desenvolvidos levando-se em conta as características da plataforma Java ME, exigindo configuração mínima do perfil MIDP 2.0 e configuração 1.1;

- **Restrito ao domínio de aplicações móveis orientadas a serviços**: o *framework* não é aplicado para o desenvolvimento no domínio de jogos. A arquitetura proposta foi modelada para o domínio de aplicações moveis orientadas a serviços;

- **Tecnologia de comunicação**: o modulo de comunicação foi desenvolvido e testado apenas para o acesso a *Web Services*;

- **Não possui componentes para construção de interface gráfica com o usuário:** existem diversos *frameworks* e API's que auxiliam a construção de interface gráfica em Java ME [58], portanto no MoSOA não foram implementados componentes para tal.

5.3. Desenvolvimento do *framework* usando *UML Components*

Com o escopo negativo do projeto bem definido, as próximas seções mostram em detalhes cada atividade do UML *Components* (ver seção 2.3.3 e 2.3.5) para construção do *framework*.

5.3.1. Requisitos

A primeira atividade do *workflow* de UML *Components* é a de Requisitos. Segundo Jack e Daniels, autores do método, essa fase é bem flexível, não fornece nenhum detalhamento de levantamento de requisitos, podendo ser adaptada [16]. O método apenas exige um conjunto mínimo de artefatos de saída a ser usado como entrada na atividade de Especificação. Esses artefatos são: modelo de caso de uso, que será mostrado na seção 5.3.1.2, e modelo conceitual do negocio, na seção 5.3.1.2):

Portanto, analisando o domínio das aplicações do *framework* foram identificados os seguintes requisitos funcionais (RF) e não funcionais (NF):

- **[RF 01] Acesso a serviços:** O *framework* deverá prover um módulo para acesso a *web services* de forma transparente e simples para o usuário, sem que seja necessário que o dispositivo tenha a Web Service API [22] implementada.

- **[RF 02] Tratamento de eventos do usuário**: O *framework* deverá prover um mecanismo simples e eficiente para tratamento de eventos do usuário. Deve ser implementado de tal forma que seja possível ser usado em qualquer aplicação Java ME.

- **[RF 03] Gerenciamento de execução de serviços:** O *framework* deverá implementa um mecanismo simples e eficiente para adição, remoção e modificação de serviços.

- **[RF 04] Suporte a internacionalização:** O *framework* deverá fornecer um mecanismo de suporte a internacionalização das aplicações. Os textos devem estar em arquivos de propriedades específicos para cada idioma suportado pela aplicação. Com isso as mudanças nos textos não terão impacto no código da aplicação.

- **[NF 01] Extensibilidade**: O *framework* deverá especificar uma arquitetura padrão no domínio de aplicações orientadas a serviços.

- **[NF 02] Portabilidade**: O *framework* deverá ser usado no desenvolvimento de aplicações Java ME que sejam executados em aparelhos celulares que implementem a configuração CLDC 1.1 (ou superior) e perfil MIDP 2.0 (ou superior) [3], sem a necessidade de nenhuma API adicional.

- **[NF 03] Sistema *open-source*:** O *framework* deverá ser open-source (código aberto, sob a licença GPL) e seu código fonte deverá ser disponibilizado em sites como SourceFourge.net para que qualquer individuo ou instituição possa utilizá-lo, integrá-lo com outros sistemas ou customizá-lo como desejar.

- **[NF 04] Documentação:** O *framework* deverá ser bem documentado facilitando o entendimento do código fonte, requisitos e casos de uso.

5.3.1.1. Modelo de casos de uso

O modelo de caso de uso é composto pelo diagrama de casos de uso a serem desenvolvidos e suas respectivas descrições [16]. A figura 12 mostra o diagrama de casos de uso do *framework* e na próxima seção é mostrada a descrição de cada um deles.

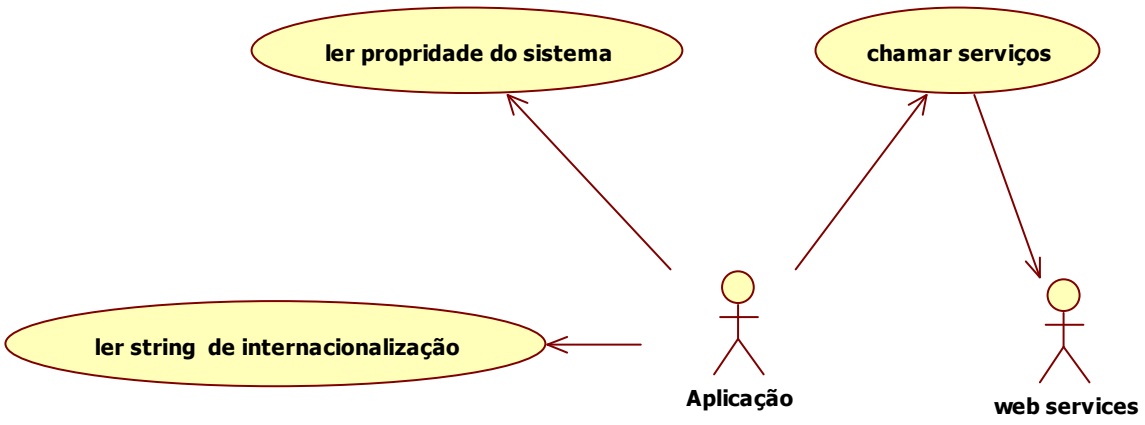


Figura 12 - Diagrama de casos de uso do *framework*.

5.3.1.2. Descrição dos casos de uso

UC01 – Chamar Serviço

Este caso de uso é responsável pela chamada de um serviço em um determinado *web services*. Isso será feito de forma transparente para o desenvolvedor, ou seja, sem a necessidade de implementar o mecanismo de mensagens trocadas com o *web service*.

Pré-condição: nenhuma.

Pós-condição: resposta do *web services* encapsulada em um objeto.

Fluxo de eventos principal:

1. A aplicação informa o tempo máximo de retorno da comunicação (timeout), o *name space*, a versão do *wsdl*, o tipo de *encode*, o nome do método a ser chamado, os parâmetros e a *url* do *web service*.([FS 01])

2. O *framework* monta a mensagem corretamente e envia ao *web service*.([FS 02])

3. *Web service* retorna a mensagem de resposta.

4. O *framework* decodifica a mensagem e retorna um objeto contento os elementos da resposta esperada.

Fluxos secundários (alternativos e de exceção)

[FS 01] Parâmetros inválidos

1. Se os parâmetros para comunicação forem inválidos o *framework* informa a aplicação e encerra o caso de uso.

[FS 02] *Timeout*

1. Se o tempo de resposta for superior ao *timeout* informado, o *framework* informa a aplicação e encerra o caso de uso.

UC03 – Ler string do arquivo de Internacionalização

Este caso de uso é responsável pela captura de um determinado termo em um arquivo de internacionalização. Ele recebe uma chave, que é única para cada entrada do arquivo, e retorna o valor (a String) associado.

Pré-condição: o arquivo de internacionalização está carregado.

Pós-condição: A String associada à chave informada.

Fluxo de eventos principal:

1. A aplicação informa a chave do termo desejado.

2. O *framework* procura a chave no arquivo.([FS 01])

3. O *framework* retorna o valor da String.

Fluxos secundários (alternativos e de exceção)

[FS 01] Chave inválida

1. Se a chave informada não existe no arquivo, o *framework* informa a aplicação e encerra o caso de uso.

UC03 – Ler propriedades do sistema

Este caso de uso é responsável pela leitura de propriedades do sistema em um arquivo de propriedades. Essas propriedades são, por exemplo, localização e parâmetros de um serviço, propriedades de telas (altura, largura, espaçamentos).

Pré-condição: o arquivo de propriedades está carregado.

Pós-condição: O valor associado à propriedade informada.

Fluxo de eventos principal:

1. A aplicação informa a propriedade desejada.

2. O *framework* procura a propriedade no arquivo.([FS 01])

3. O *framework* retorna o valor da propriedade.

Fluxos secundários (alternativos e de exceção)

[FS 01] Propriedade inválida

1. Se a propriedade informada não existe no arquivo, o *framework* informa a aplicação e encerra o caso de uso.

5.3.1.3. Modelo conceitual do negócio

O modelo conceitual do negocio não é um modelo de software, mas uma informação que existe para o domínio do problema. É, basicamente, um diagrama de classes que captura os principais conceitos e identifica o relacionamento entre eles.

Os principais conceitos identificados são: termo(strings que será usada na aplicação), aplicação, tela, comando(eventos do usuário), parâmetro(dos serviços), resposta(dos serviços) e propriedades(da aplicação). Um possível modelo conceitual do negocio para o domínio de aplicações moveis orientadas a serviço é mostrado na figura 12.

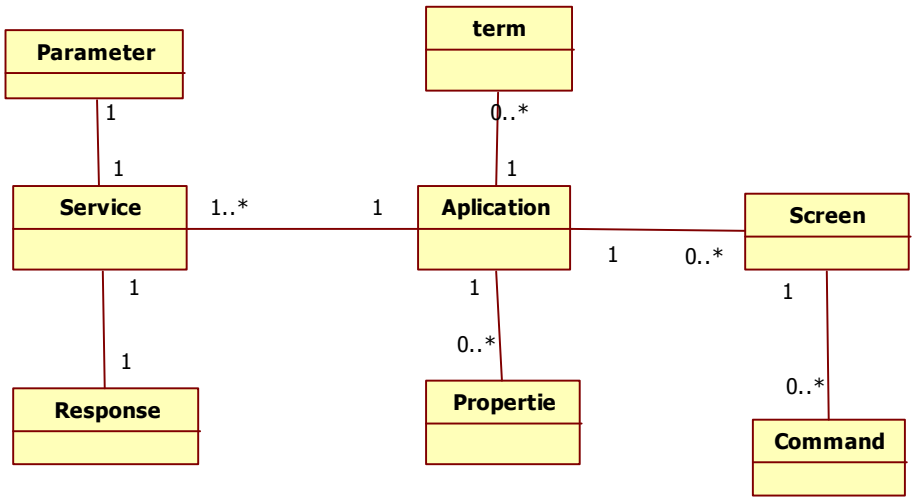


Figura 13 - Modelo conceitual de negócio.

5.3.2. Especificação

Esta seção mostra em detalhes o fluxo de Especificação para desenvolvimento do *framework* proposto. A figura 14 mostra em detalhes as atividades e os artefatos gerados em cada fase.

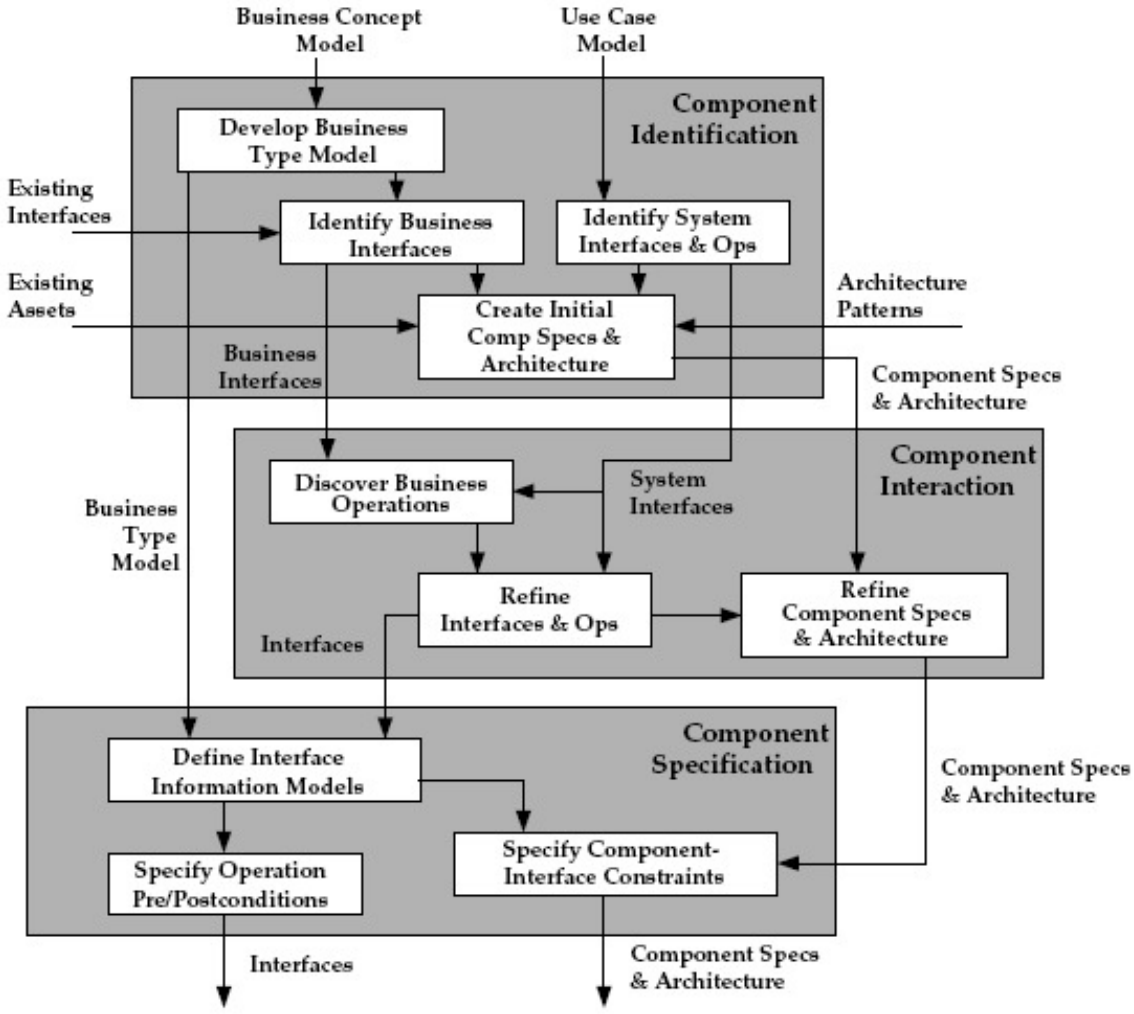


Figura 14 - Detalhamento das três fases do *workflow* de Especificação [16].

5.3.2.1. Identificação dos componentes

Identificação de Componentes (*Component Identification*), ilustrado na figura 14, é a primeira etapa da Especificação (*Specification*). Ele recebe como entrada o modelo conceitual de negocio e o modelo de casos de uso para identificar o conjunto inicial de interfaces e criar as primeiras especificações dos componentes, que juntos fornecem o primeiro modelo da arquitetura dos componentes.

Esta fase gera um importante artefato interno para o fluxo de especificação: o **modelo de tipos de negócio** *(Business Concept Model)*, que é usado para desenvolver os **modelo de informação de interface**.

A ênfase nesta fase é sobre a descoberta de quais informações precisam ser gerenciadas, quais interfaces são necessárias para controlá-las, que componentes são necessários para fornecer as funcionalidades do sistema, e como eles se relacionam. Além da descoberta de componentes, também é preciso levar em consideração componentes e API's existentes, que precisarão ser adaptados, e padrões que serão utilizados na arquitetura do sistema.

5.3.2.1.1. Modelo de tipos do negocio

O modelo de tipos de negocio é um artefato interno do fluxo de especificação. É um diagrama de classe com o refinamento do modelo conceitual do negocio onde são identificados tipos type, core, data types e são eliminados os conceitos não utilizados. A figura 15 mostra o modelo de tipos do negócio do *framework* (Note que term mudou de nome para i18n, uma abreviação para internacionalização).

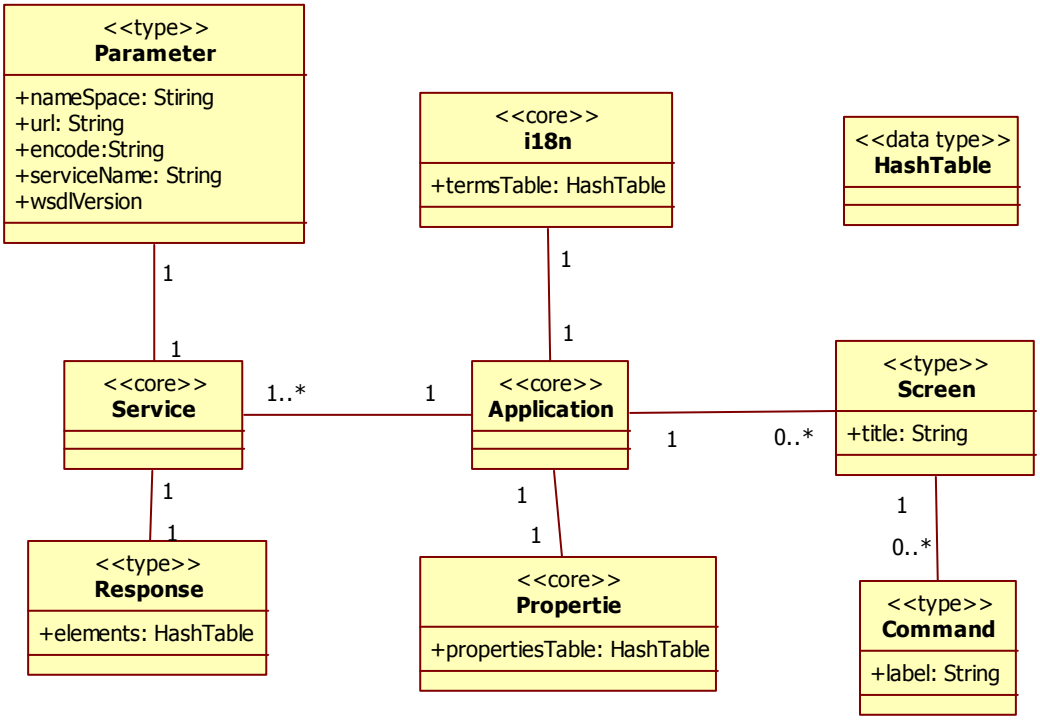


Figura 15 - Modelo de tipos de negocio do *framework* MoSOA.

A partir da especificação dos tipos *core*, é possível criar a **especificação inicial de interfaces de negócio** (figura 16). Normalmente é associado (usando agregação) um tipo *interface type* para cada tipo *core*. Para facilitar a distinção entre as interfaces de negocio e interfaces de sistemas, os nomes das interfaces de sistemas seguirá o padrão IXXMgt, onde XX é o nome da interface.

Com isso, até o momento foram identificadas as seguintes interfaces de negócio: Ii18nMgt, IServiceMgt, IPropertieMgt, IApplicationMgt.

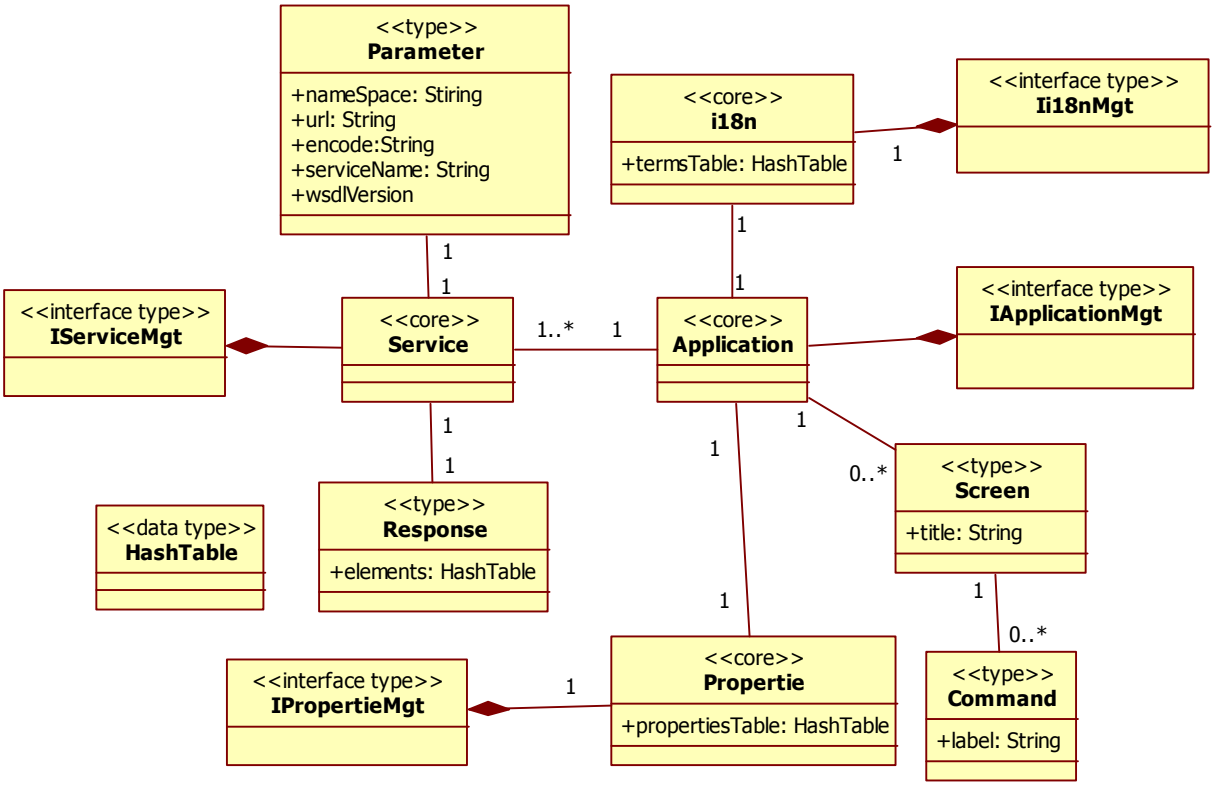


Figura 16 - Especificação inicial de interfaces de negócio.

5.3.2.1.2. Identificação de interfaces

O método UML *Components* propõe dois tipos de interfaces: **Interfaces de sistema** e **Interfaces de negócio** [16]. Interfaces de sistema são geradas a partir do modelo de casos de uso. Suas operações iniciais são encontradas a partir da análise dos casos de uso, ou seja, são as operações que o *framework* terá que fornecer para que os requisitos propostos sejam atendidos. Já as interfaces de negócio são as interfaces geradas a partir do modelo de tipos de negócio apresentadas anteriormente.

Analisando os casos de uso do *framework* descritos na seção 5.3.1.2, as interfaces de sistema identificadas são apresentadas na figura 17.

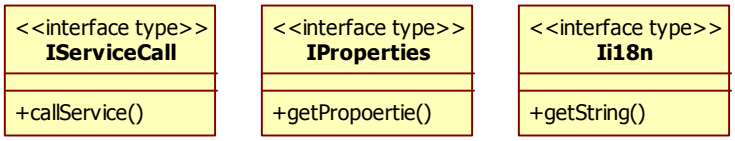


Figura 17 - Interfaces de sistema do *framework*.

5.3.2.1.3. Especificação inicial da arquitetura dos componentes

O objetivo desta fase é identificar, a partir do conjunto de interfaces anteriormente definidas e de componentes disponíveis, a especificação inicial dos componentes do *framework* e como eles se relacionam. É importante levar em conta que cada componente é uma unidade de desenvolvimento independente e que pode ser substituído posteriormente sem impactar os outros componentes. A figura 18 mostra a especificação inicial da arquitetura dos componentes do *framework*.

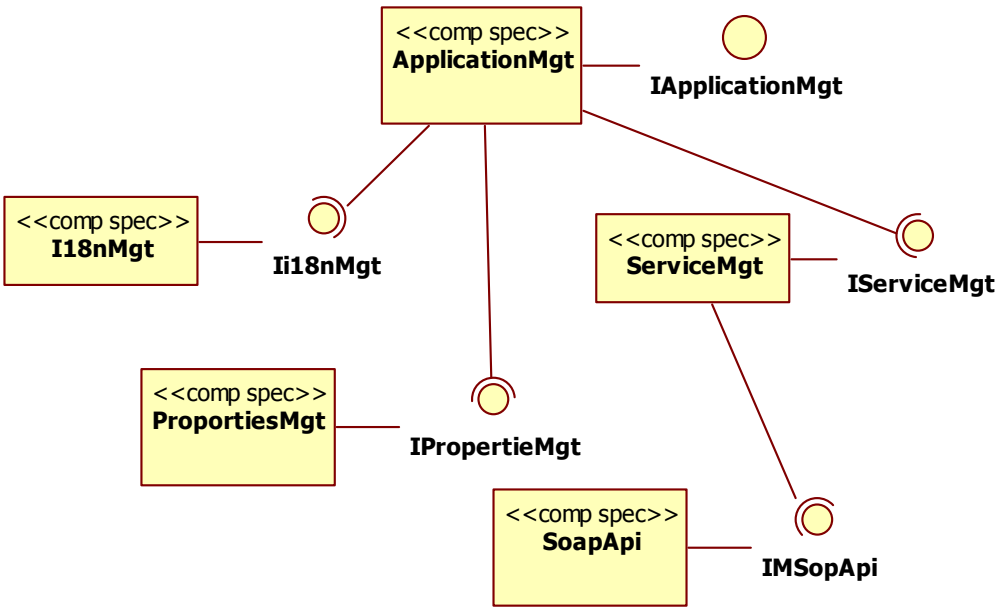


Figura 18 - Especificação inicial da arquitetura dos componentes.

O novo componente SoapAPi surgiu da necessidade de utilizar uma API que implemente o protocolo SOAP [21] em Java ME. Portanto, será necessária a criação de um componente que encapsule essa API. Com isso, a manutenção, atualização e mudanças da API não tem impacto em nenhum outro componente do *framework*, apenas no SoapAPI.

5.3.2.2. Interação dos componentes

O principal objetivo desta fase é examinar como as operações das **interfaces de sistema**, identificadas na seção anterior (figura 17), serão realizadas usando a arquitetura dos componentes (figura 18). Para alcançar esse objetivo, são usados digramas de interações (digramas de colaboração ou diagramas de seqüência) para descobrir as operações das interfaces de negócio

Depois de identificadas as operações das interfaces de negócio, é feito um refinamento das operações de todas as interfaces e da arquitetura dos componentes.

Os autores do método sugerem o uso de diagramas de colaboração [16], mas para o melhor entendimento da modelagem do *framework*, neste trabalho será usado diagramas de seqüência. A próxima seção mostra os digramas de seqüência do *framework*.

5.3.2.2.1. Diagramas de seqüência

A figura 19 mostra o diagrama de seqüência para a operação callService da Interface de sistema IServiceCall. Foram identificadas as seguintes operações com as respectivas assinaturas:

- **IServiceMgt:**

    1. callService(Parameter, ResponseListener)

- **ISoapAPI:**

  1. SoapEnvelop: createSoapEnvelop(String url, int wsdlVersion, int wsdlType, HashMap parameter, String methodName).

  2. SoapObject:requestService(SoapEnvelop)

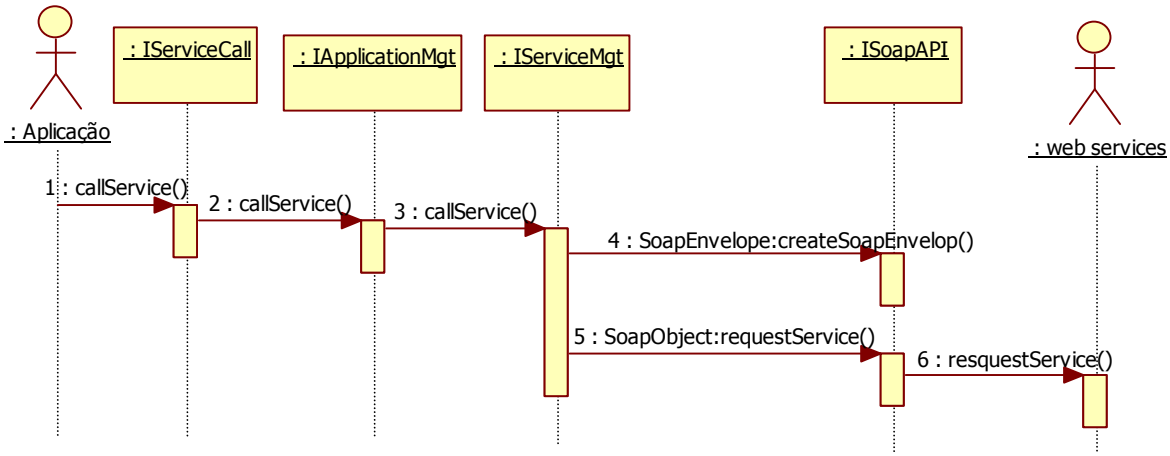


Figura 19 - Diagrama de seqüência da operação serviceCall.

A figura 20 mostra os diagramas de seqüência para os operações da Interface de sistema IProperties. Foram identificadas as seguintes operações com as respectivas assinaturas:

- **IPropertieMgt:**

  1. int: getInt(String key)

  2. String: getStringt(String key)

  3. loadProperties(String fileName)

- **IAplicationMgt:**

  1. int: getIntPropertie(String key)

2. String: getStringPropertie(String key)

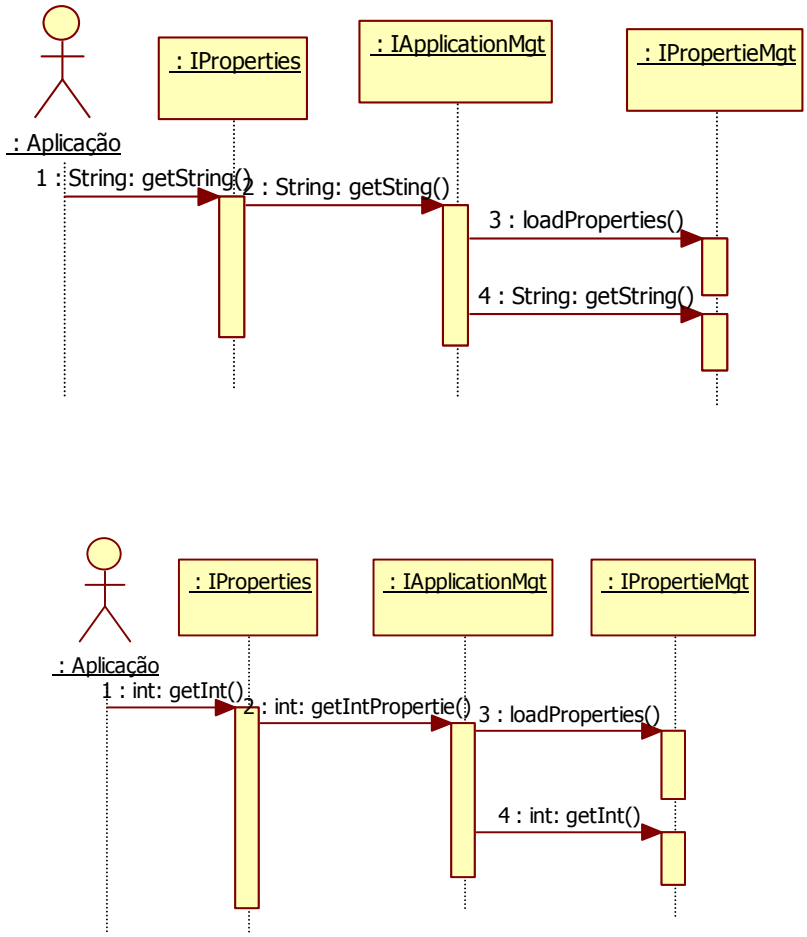


Figura 20 - Diagramas de seqüência das operações da interface de sistema Iproperties.

A figura 21 mostra os diagramas de seqüência da operação getString da Interface de sistema Ii18n.

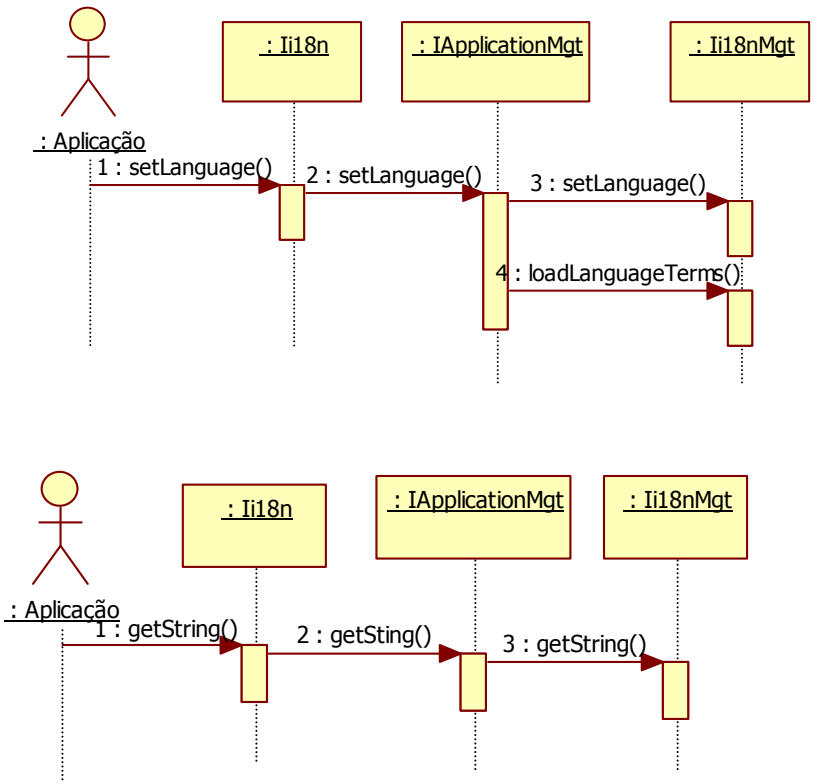


Figura 21 - Diagramas de seqüência das operações da interface de sistema Ii18n.

Foram identificadas as seguintes operações com as respectivas assinaturas:

- **Ii18n Mgt:**

  1. loadLanguageTerms(String fileName)
  2. String: getStringt(String key)
  3. setLanguage(String key)

- **IAplicationMgt:**

  1. String: getString(String key)
  2. setLanguage(String language)

5.3.2.2.2. Refinamentos das interfaces e do diagrama de arquitetura dos componentes

Com a elaboração dos diagramas de seqüências apresentados na seção anterior foram identificadas novas operações em todas as interfaces do *framework*. Com isso, o método UML *Components* gera como artefato de saída as interfaces de sistema, interfaces de negócio e arquitetura de componentes refinadas com as novas operações. A figura 22 mostra as interfaces de negócio com as assinaturas de suas operações. A figura 23 apresenta as interfaces de sistema com as novas operações identificadas. Por fim, a figura 24 mostra a arquitetura dos componentes refinada.

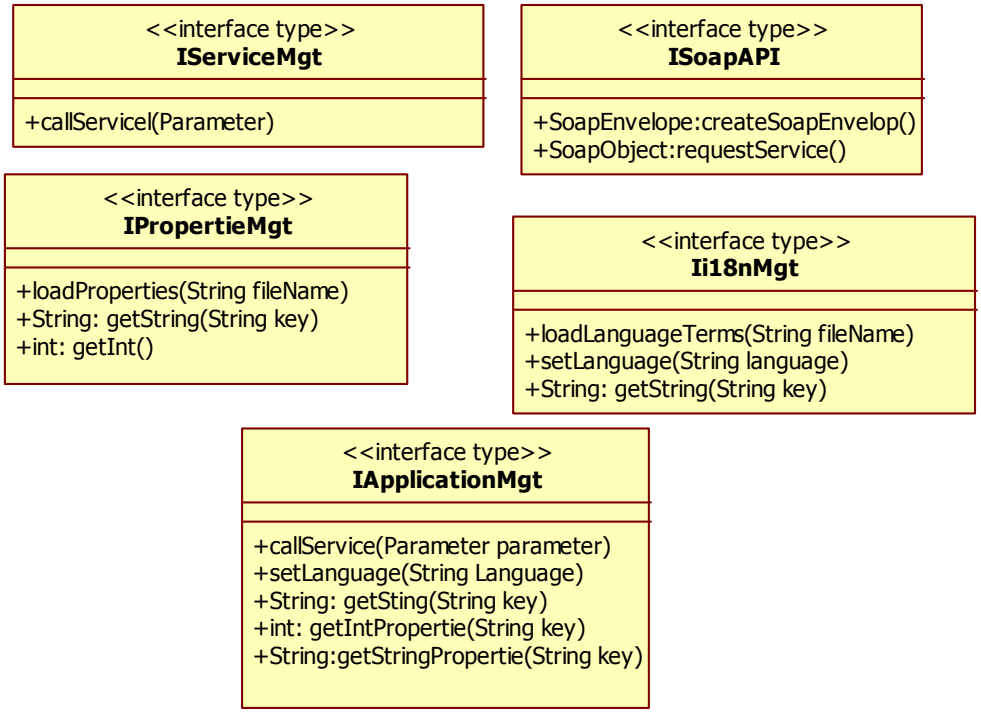


Figura 22 - Interfaces de negócio refinadas.

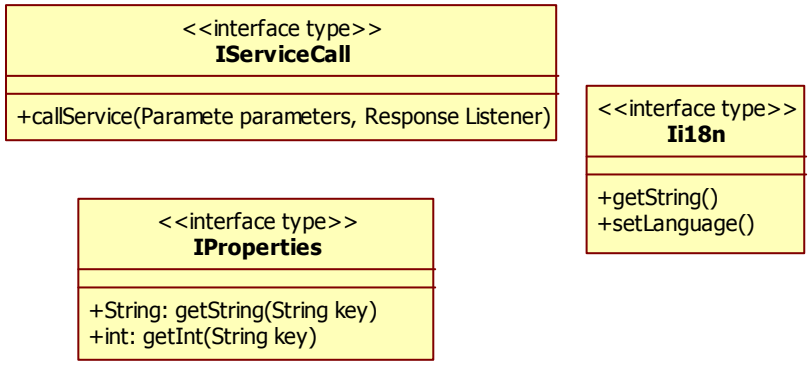


Figura 23 - Interfaces de sistema refinadas.

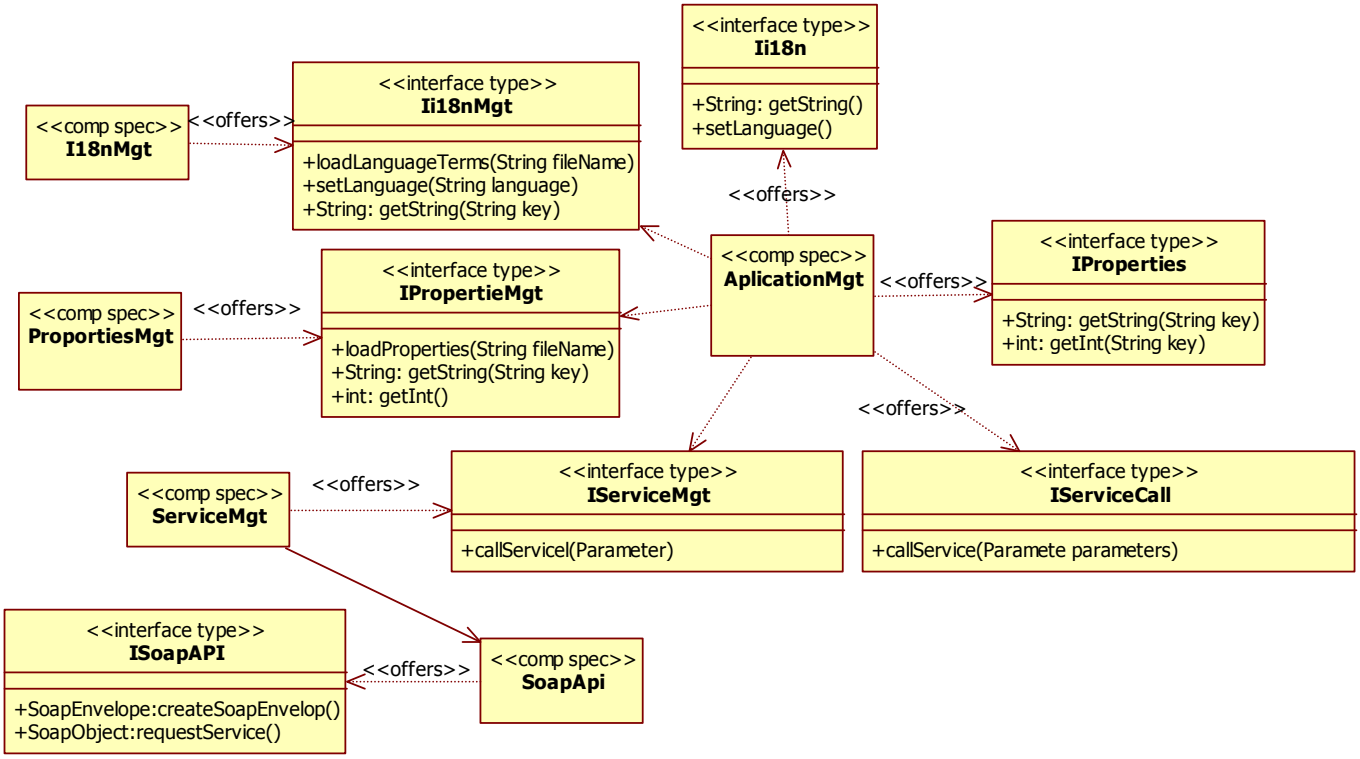


Figura 24 - Modelo de arquitetura dos componentes refinado.

5.3.2.3. Especificação dos componentes

O principal objetivo desta fase é construir **o modelo de informações das interfaces** e o **modelo final de especificação dos componentes**. O método *UML Components* define três passos a serem seguidos: especificar o modelo de informação da interface, definir pré e pós-condições e montar especificações dos componentes.

Os autores do método sugerem que na fase definir pré e pós-condições sejam escritas restrições contratuais usando OCL [61] para cada operação das interfaces. Porém, para o contexto desse trabalho não é necessário detalhar as operações até esse nível e optou-se por não aplicar essa fase no processo de desenvolvimento do *framework*.

5.3.2.3.1. Modelo de informação de interface

O passo especificar modelo de informação da interface gera como saída o modelo de informação das interfaces que é um diagrama de classes com o mapeamento dos relacionamentos de cada interface do *framework* com os tipos definidos no modelo de tipos de negócio e outras interfaces. As figuras a seguir mostram os diagramas de classe das interfaces do *framework*.

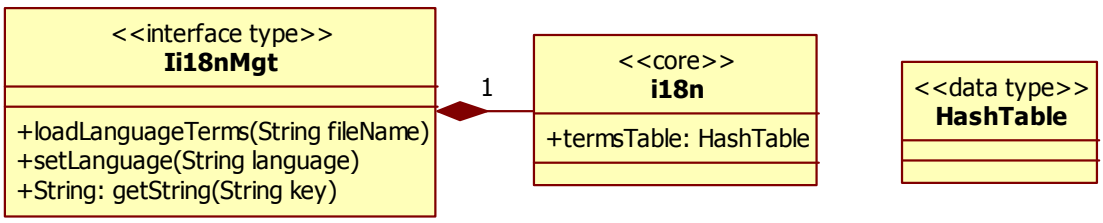


Figura 25 - Modelo de Informação da Interface Ii18nMgt.

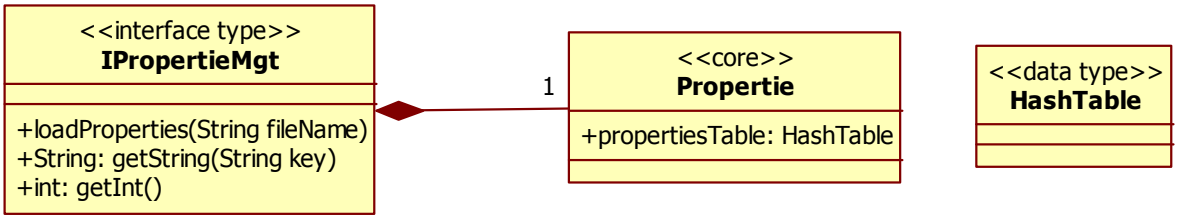


Figura 26 - Modelo de Informação da Interface IPropertieMgt.

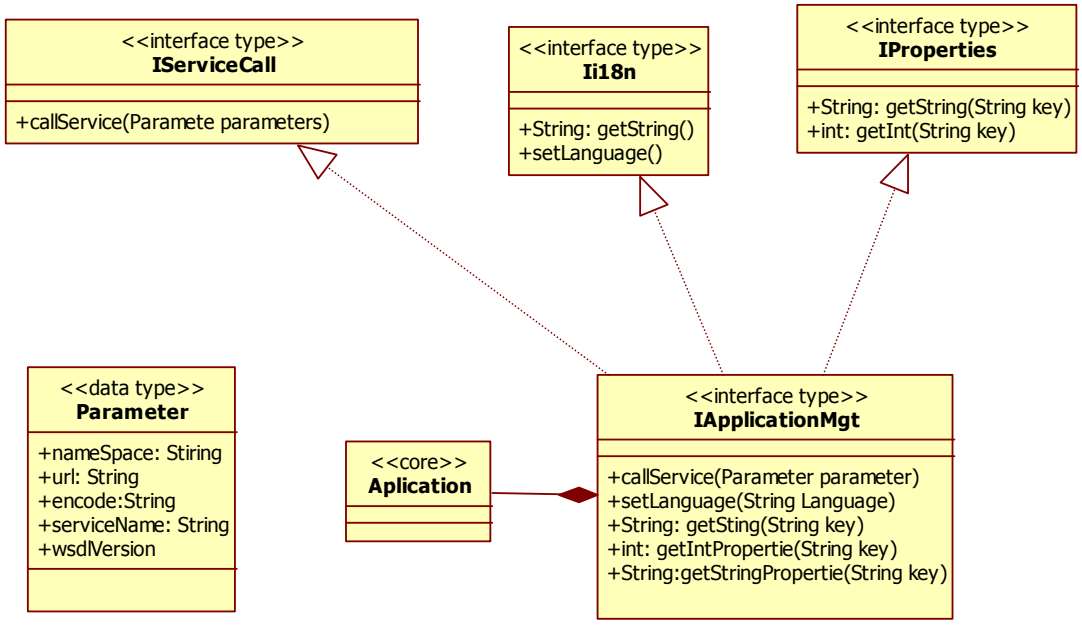


Figura 27 - Modelo de Informação das Interfaces IApplicationMgt, IServiceCall, I18n e IPropertie.

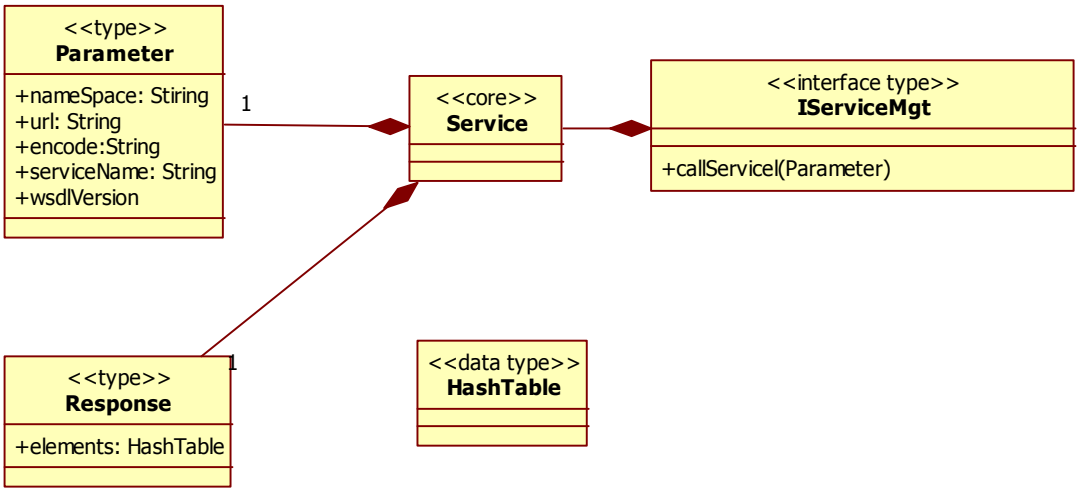


Figura 28 - Modelo de Informação da Interface IServiceMgt.

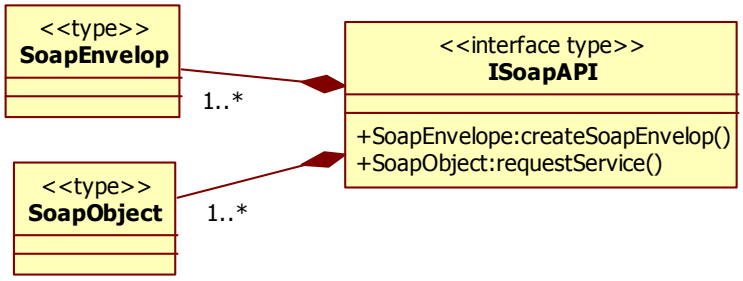


Figura 29 - Modelo de Informação da Interface ISoapAPI.

5.3.2.3.2. Modelo final de arquitetura dos componentes

A partir do **modelo de informações das interfaces** é necessário refinar novamente as especificações dos componentes com as mudanças feitas. A figura 30 mostra o modelo final da arquitetura dos componentes que servirá como entrada para a atividade de adequação. Com isso, a atividade de Especificação gera como saída um conjunto de diagramas de classes totalmente independente de tecnologia que será usada na implementação do *framework* MoSOA.

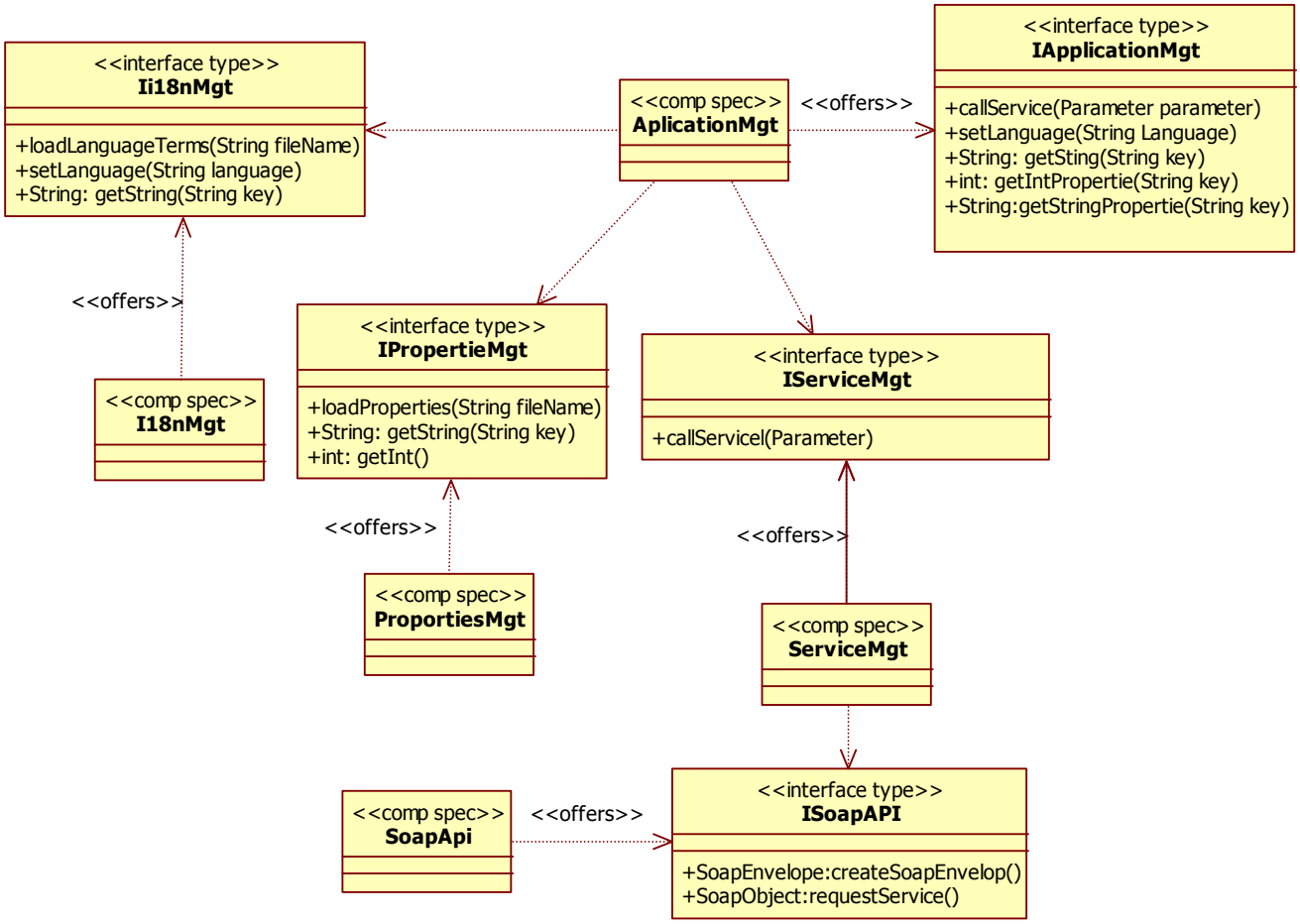


Figura 30 - Modelo final da arquitetura dos componentes.

5.3.3. Adequação

Até agora foi feita a modelagem dos componentes do *framework* MoSOA de forma que eles sejam completamente independentes e possam ser implementados separadamente. Nesta fase é necessário definir o diagrama de classes final do *framework*, levando-se em consideração características da plataforma Java ME (ver capitulo 4), que atenda os requisitos definidos. Para isso, será implementado um conjunto de classes, seguindo o padrão State MVC [60], que serão herdadas pelas aplicações desenvolvidas a partir do *framework* para

que elas possuam uma arquitetura padrão robusta e extensível, facilitando a manutenção e adição de novos serviços. A figura 31, mostra a arquitetura dos componentes com as novas classes adicionadas.

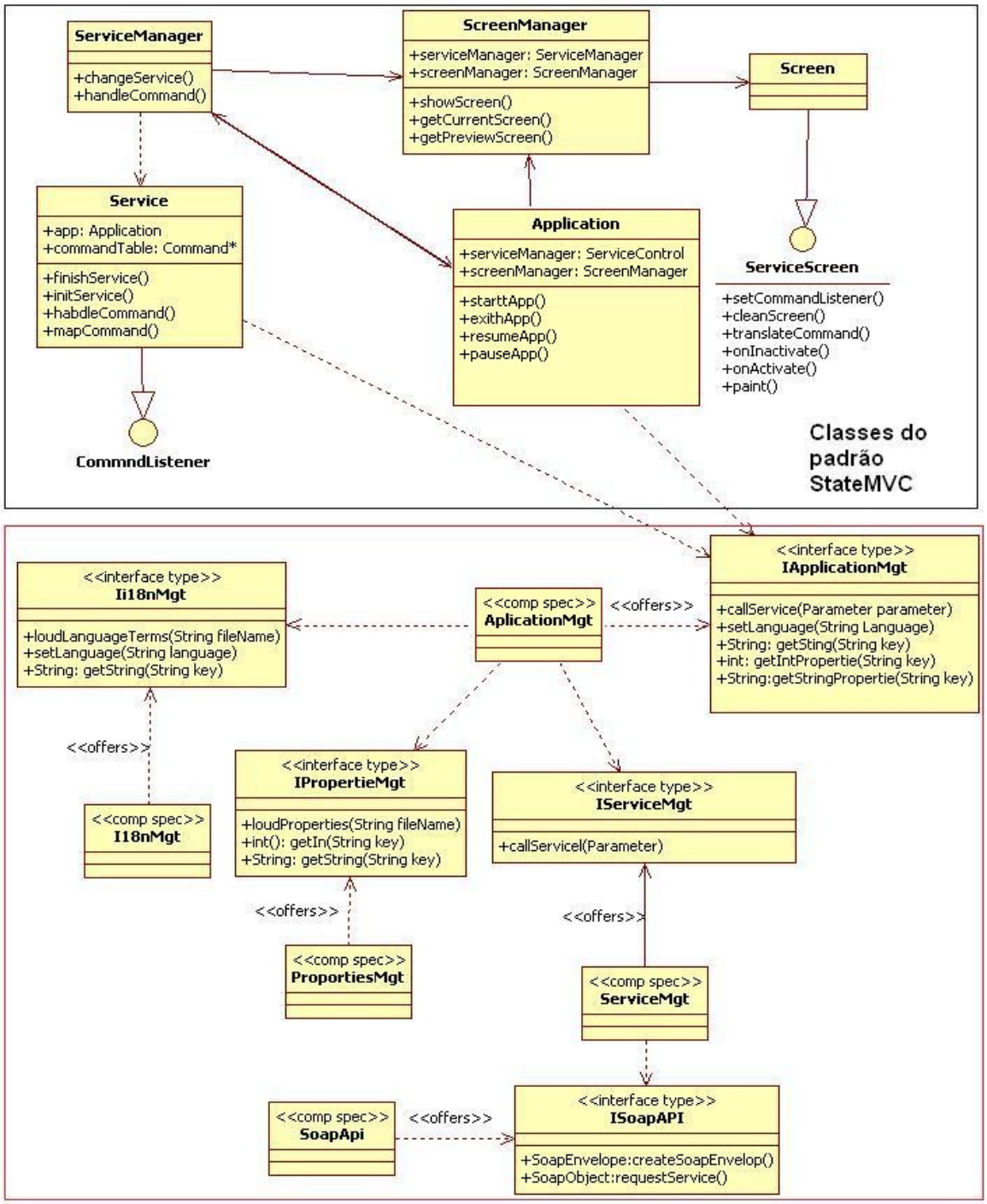


Figura 31 - Diagrama de classes do *framework*.

5.3.4. Implementação

A implementação do *framework* foi realizada com base nas versões CLDC 1.1 e MIDP 2.0 (ver Seção 4.1). O MoSOA foi implementado em 19 classes, divididas em 10 pacotes e com 887 linhas de código não comentadas (dados extraídos através da ferramenta Metrics[41]). O Arquivo JAR (Java Archive) teve seu tamanho final de 87 Kb.

A tabela 2 mostra o mapeamento das entidades modeladas (ver figuras 25, 26, 27, 28, 29 e 30) com as respectivas classes implementadas. Novas classes foram criadas para o tratamento de exceções.

	<<Estereotipo>> nome do tipo	Arquivo	Tipo	LOC
<<comp spec>> ApplicationMgt	<<interface type>> IApplicationMgt	IFrameworkMgt.java	Interface	32
	<<core>>Application	FrameworkManager.java	Classe	97
		ApplicationExeception.java	Classe	31
<<comp spec>> ServiceMgt	<<interface type>> IServiceMgt	IServiceMgt.java	Interface	13
	<<core>>Service	ServiceManager.java	Classe	56
	<<type>> Parameter	Parameter.java	Classe	59
	<<type>> Response	SoapObject.java	Classe	12
		ResponseListener.java	Classe abstrata	12
		ServiceException.java	Classe	8
<<comp spec>> PropertieMgt	<<interface type>> IPropertieMgt	IPropertieMgt.java	Interface	15
	<<core>>Propertie	PropertieManager.java	Classe	108
		PropertieExeception.java	Classe	8
<<comp spec>> I18nMgt	<<interface type>> Ii18nMgt	Ili18nMgt.java	Interface	15
	<<core>>Term	I18nManager.java	Classe	94
		I18nExeception.java	Classe	7
<<comp spec>> SoapApiMgt	<<interface type>> SoapApiMgt	ISoapApiMgt.java	Interface	13
		SoapApiManager.java	Classe	105
	<<type>> SoapObject	SoapObject.java	Interface da API	
	<<type>> SoapEnvelop	SoapEnvelop.java	Classe da API	

Tabela 2 - Mapeamento dos componentes com as classes implementadas (LOC: Lines of code).

Como mencionado na seção anterior, foram adicionadas classes para implementar o padrão State MVC. A tabela 3 mostra o mapeamento das classes do modelo de arquitetura final com as classes implementadas.

Modelo	Arquivo	Tipo	LOC
ServiceControl	ServiceControl.java	Classe	85
ScreenManager	ScreenManager.java	Classe	31
Application	ApplicationMIDlet.java	Classe Abstrata	39
Service	ServiceMIDlet.java	Classe Abstrata	37

Tabela 3 - Mapeamento das classes implementadas (LOC: Lines of code).

5.4. Construindo aplicações com o MoSOA

Para construir aplicações utilizando o *framework* MoSOA os seguintes passos devem ser seguidos:

1. **Implementar o MIDlet de entrada da aplicação:** a classe de entrada da aplicação deverá herda da classe ApplicationMIDlet e implementar os métodos abstratos startApp(), pauseApp(), resumeApp() e exitApp();

2. **Implementar as classes de serviço:** Todas as classes responsáveis pelos serviços da aplicação deverão herda da classe pacote br.com.mobilit.control.Service. Também deverá se implementados os métodos abstratos: initService(), finishService(), onPauseApp(), onResumeApp() e setCurrentScreensListener().

3. **Criação e adição dos serviços:** No método startApp() , na classe de entrada da aplicação, deverá ser chamado o método initialize(int maxNumberOfService), criaar todos os serviços da aplicação e chamar o método changeService(int it) para chamar o serviço inicial.

4. **Utilizando serviços dos componentes do *framework*:** Qualquer chamada aos serviços disponibilizados pelo *framework* deverá ser feita pela interface IApplicarionMgt. As classes ApplicationMIDlet e Services possuem como atributo essa interface. Para obter referencia a essa interface em qualquer outra parte do código, deve-

se se chamar o método estático getApplicationMgt () da classe ApplicationMIDlet.

5. **Mudanças de Serviços:** as mudanças de serviços são feitas através de chamadas ao método changeService(int Id) de SerciceControl. Apenas as classes que herdam de Service e ApplicationMIDlet do pacote br.com.mobilit.control devem possuir essa classe como atributo.

6. **Mudanças de telas:** as mudanças na tela da aplicação deverá ser feita exclusivamente pela classe ScreenMananer. Apenas as classes que herdam de Service e ApplicationMIDlet do pacote br.com.mobilit.control devem possuir essa classe como atributo.

5.5. Validação

Para validar o *framework* foi desenvolvida a aplicação **MobiServices**. Essa aplicação é composta por três serviços distintos: *Global Weather, Translator e Quote of the Day*. O primeiro serviço é responsável por obter informações climáticas de uma determinada cidade informada pelo usuário. O Segundo serviço é um tradutor de textos para diversas línguas. O terceiro serviço obtém a frase de do dia. A figura 32 apresenta os *screenshots* da aplicação.

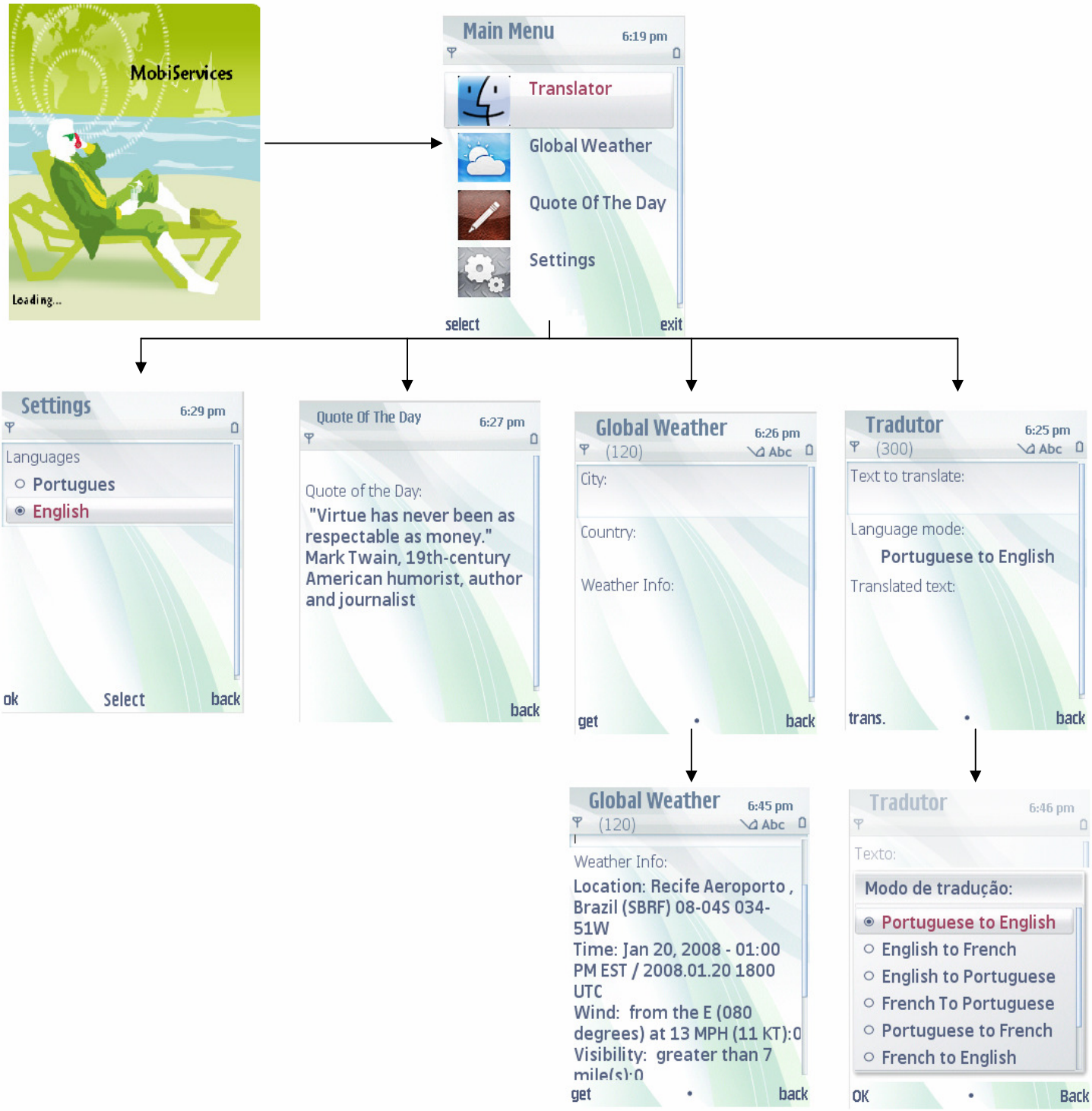


Figura 32 - *Screenshots* da aplicação MobiServices.

O Mobiservice foi instalado e funcionou corretamente nos seguintes aparelhos: Motorola v3i, Nokia 6101 e Sony Erricson z550i. O código fonte da aplicação está disponível no site do projeto [59].

6. Conclusão

Neste trabalho foi especificado, projetado e implementado o *framework* MoSOA, um *framework* para desenvolvimento de aplicações móveis aplicando os princípios de Arquitetura Orientada a Serviços para a plataforma Java ME utilizando o método UML *Components*.

Antes da construção do *framework*, foi apresentada uma breve introdução sobre reuso de software e desenvolvimento baseado em componentes, detalhando os principais conceitos e definições usados durante a implementação. Depois foi feita uma análise dos principais métodos de desenvolvimento baseado em componentes, a fim de escolher o mais adequado para o trabalho.

Em seguida foram detalhados os conceitos relativos à Arquitetura Orientada a Serviços (SOA, *Service Oriented Architecture*) e as vantagens em adotá-la. O objetivo foi esclarecer termos, definições e características relativas a SOA, muitas vezes usados indevidamente pelos profissionais de TI, que foram utilizados no desenvolvimento do *framework*.

Posteriormente, foi apresentada a tecnologia Java ME que serviu de base para implementação do *framework* proposto, mostrando uma visão geral da plataforma e o processo para construção de aplicações.

No final foi mostrado o processo de desenvolvimento do *framework*, detalhando os artefatos gerados e explicando cada fase do método UML *Components*. Depois, foi construída uma aplicação para validá-lo.

6.1. Contribuições do Trabalho

As principais contribuições do trabalho são listadas a seguir:

- Projeto de um *framework* para o domínio de aplicações móveis orientadas a serviço independente de tecnologia. Os diagramas de classes gerados no capitulo 5 não estão ligados a nenhuma tecnologia especifica, podendo ser usado como referencia para implementação em outras plataformas como *BREW* [62], *Symbian* [63] ou *Windows Mobile* [64].

- O *framework* MoSOA, resultado desse trabalho, é um projeto *open-source* (sob a licença GPL). Todos os artefatos gerados na construção do *framework* (documento de requisitos, modelo de casos de uso, diagramas de classes e seqüência e código fonte) podem ser encontrados na página do projeto [59]. Com isso, terceiros podem utilizá-los para construir novas aplicações, agilizando o processo de desenvolvimento, e contribuir para a evolução do projeto.

- Estudo dos principais métodos de desenvolvimento de software baseado em componentes existentes na literatura, a fim de encontrar o mais adequado para o contexto de aplicações móveis e desenvolvimento de *framework*s.

6.2. Trabalhos relacionados

A seguir são mostrados alguns trabalhos relacionados:

- **Componentização de Software em Java 2 Micro Edition**

Nesse trabalho [66] foi desenvolvido um *framework* que encapsula os componentes básicos de interface com o usuário para telefones celulares. O trabalho apresenta apenas a fase de implementação do método UML *Components* e não foram seguidas todas as fases proposta por esse método mostrada no capitulo 5.

- ***State* MVC: Estendendo o padrão MVC para uso no desenvolvimento de aplicações para dispositivos móveis**

Nesse trabalho [60] é apresentada uma extensão do padrão MVC para o desenvolvimento de aplicações para dispositivos móveis chamado *State* MVC (SMVC). Nele foi mostrada a implementação do padrão para a plataforma BREW. Esse padrão foi utilizado para definir a arquitetura base para construção das aplicações a partir do *framework* MoSOA.

- **Um estudo sobre o desenvolvimento orientado a serviços**

No trabalho [33] foi feito um estudo sobre o desenvolvimento de aplicações orientadas a serviços e construído um *framework* que oferece suporte ao desenvolvimento de aplicações SOA para plataforma J2EE.

6.3. Trabalhos futuros

Trabalhos futuros que poderão ser considerados são listados a seguir:

- Utilizar ferramentas e métodos de engenharia de domínio [15] para extrair características não identificadas neste trabalho no domínio de aplicações móveis orientadas a serviços.

- Implementar aplicações comerciais reutilizando o *framework* MoSOA e aplicar rodadas de testes formais para identificar erros não encontrados até o momento.

- Integrar o *framework* a IDE's (*Integrated Development Environment*) como o Eclipse ou NetBeans e automatizar o processo de desenvolvimento de aplicações através de ferramentas visuais presentes nessas ferramentas.

- Propor um processo para construção de *framework*s integrando métodos de engenharia de domínio e desenvolvimento baseados em componentes.

6.4. Considerações finais

Este trabalho se propôs a especificar, projetar e implementar um *framework* com objetivo principal de diminuir o tempo de desenvolvimento de aplicações mSOA (do inglês, *mobile* SOA). Para alcançar esse objetivo foi usado o método UML *Compoenents* e aplicado os princípios de arquitetura orientada a serviços.

Um ponto forte do trabalho foi esclarecer os conceitos relativos a arquitetura orientada a serviço, muitas vezes confundidos pelos profissionais de TI e sempre relacionados a grandes aplicações coorporativas e *web*, e adotá-los para o

desenvolvimento de aplicações para dispositivos móveis. Outro ponto de destaque foi a utilização do método UML *Components* que se mostrou muito útil para o desenvolvimento do *framework* gerando um projeto totalmente independente de tecnologia.

O *framework* MoSOA, resultado deste trabalho, será usado na implementação de aplicações comerciais pela empresa incubada do I.T.E.P MobilIT [67] com o intuito de reduzir o *time-to-market* , melhorar a qualidade dos softwares desenvolvido pela empresa a fim de submeter o *framework* a rodadas de testes formais, contribuindo para sua evolução e correções de *bugs*.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] **Teleco**, Disponível em http://www.teleco.com.br/pais/celular.asp , setembro 2007.

[2] **O Barriga Verde**. Disponível em http://www.adjorisc.com.br/jornais/obarrigaverde/noticias/index.phtml?id_conteudo=105196 , setembro 2007.

[3] **Java ME**. Disponível em http://72.5.124.55/javame/technology/index.jsp, setembro 2007.

[4] **Terra Tecnologia**. Disponível em http://tecnologia.terra.com.br/interna/0,,OI1262625-EI4796,00.html, setembro 2007.

[5] **IT Web**. Disponível em http://www.itweb.com.br/noticias/index.asp?cod=21361, setembro 2007.

[6] Thomas, E. **Service-Oriented Architecture: Concepts, Technology, and Design**, Prentice Hall, 2005.

[7] **Html Staff**. Disponível em http://htmlstaff.org/ver.php?id=7503 , outubro 2007.

[8] N. Bieberstein, S. Bose, M. Fiammante, K. Jones, and R. Shah, *Service-Oriented Architecture Compass: Business Value, Planning, and Enterprise Roadmap*, Prentice Hall PTR, Upper Saddle River, NJ (2005).

[9] **Tech Journal,** disponível em http://www.tracesistemas.com.br/tech_journal/2006/abril/artigo_destaque.htm, setembro 2007.

[10] **Software Engineering Institute (SEI),** disponível em http://www.sei.cmu.edu/, setembro 2007.

[11] **Association for Computing Machinery (ACM**), disponível em http://www.acm.org/, setembro 2007.

[12] **IBM**, disponível em http://www.ibm.com/us/, setembro 2007.

[13] **Centro de Estudos e Sistemas Avançados do Recife (C.E.S.A.R),** disponível em http://www.cesar.org.br/, setembro 2007.

[14] **Wikipedia**, disponível em http://pt.wikipedia.org/wiki/Framework , setembro 2007.

[15] Almeida, E. S.; Alvaro, A.; Garcia, V. C.; Mascena, J. C. C. P.; Burégio, V. A. A.; Nascimento, L. M.; Lucrédio, D; Meira, S. R. L. **C.R.U.I.S.E: Component Reuse in Software Engineering**, C.E.S.A.R e-book, Brazil, 2007.

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

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

[18] **PECOS Project.** Disponível em http://www.pecos-project.org/, setembro 2007.

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

[20] Thomas, E. **SOA Principles of Service Design**, Prentice Hall, 2007.

[21] **World Wide Web Consortium**. Disponível em http://www.w3.org/2002/ws/, setembro 2007.

[22] **J2ME Web Services Specification,** Disponível em http://jcp.org/en/jsr/detail?id=172 outubro 2007.

[23] **Java Plataform, Standard Edition.** Disponível em http://72.5.124.55/javase/index.jsp, setembro 2007.

[24] **Java Plataform, Enterprise Edition**. Disponível em http://72.5.124.55/javaee/index.jsp, em setembro 2007.

[25] **Java Card**. Disponível em http://java.sun.com/products/javacard/index.jsp, setembro 2007.

[26] **Java FX**. Disponível em http://java.sun.com/javafx/, setembro 2007.

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

[28] **JavaTM APIs for Bluetooth**. Disponível em http://jcp.org/en/jsr/detail?id=082 , setembro 2007.

[29] **Mobile Media API**. Disponível em http://jcp.org/en/jsr/detail?id=135 , setembro 2007.

[30] **Mobile 3D Graphics API for J2METM**. Disponível em http://jcp.org/en/jsr/detail?id=184 , setembro 2007.

[31] **The MIDP 2.0**. Disponível em http://developers.sun.com/mobility/midp/articles/pushreg/ , setembro 2007.

[32] **Arquitetura Orientada a Serviços – SOA Infraestrutura para a Inovação.** Disponível em http://www.pr.senai.br/posgraduacao/uploadAddress/Introducao%20ao%20SOA%5B31574%5D.pdf , novembro 2007.

[33] Machado, João**. Um estudo sobre o desenvolvimento orientado a serviços**. Rio de Janeiro, 2004. 89p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

[34] **Recommended Practice for Architectural Description of Software-Intensive Systems**; ANSI/IEEE Std 1471-2000.

[35] Thomas Erl, **Introdução às tecnologias Web Services. Disponível em ...**

[36] Alvaro, Alexandre. **Software component certification: a component quality model**. Recife, 2005. 124p. Dissertação de Mestrado – Centro de Informática, Universidade Federal de Pernambuco.
[37] Almeida, E. S. **RiDE: The RiSE Process for Domain Engineering**, Ph.D. Thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil, May, 2007.

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

[39] M. Ezran, M. Morisio, C. Tully, **Practical Software Reuse**, Springer, 2002, pp. 374.

[40] M. Broy, **The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems**, *IEEE Computer*, Vol. 39, No. 10, October, 2006, pp. 72-80.

[41] **Metrics**, disponivel em http://metrics.sourceforge.net/, janeiro de 2008.

[42] J. S. Poulin, **The Business Case for Software Reuse: Reuse Metrics, Economic Models, Organizational Issues, and Case Studies**, *Tutorial Notes*, Torino, Italy, June, 2006.

[43] V. R. Basili, H. D. Rombach, **Support for Comprehensive Reuse**, *Software* Engineering Journal, Special issue on software process and its support, Vol. 06, No. 05, April, 1991, pp. 306-316.

[44] W. B. Frakes, S. Isoda, **Success Factors of Systematic Software Reuse**, *IEEE Software*, Vol. 12, No. 01, September, 1994, pp. 15-19.

[45] C. W. Krueger, **Software Reuse**, *ACM Computing Surveys*, Vol. 24, No. 02, June, 1992, pp. 131-183.

[46] V. R. Basili, L. C. Briand, W. L. Melo, **How reuse influences productivity in object-oriented systems**, *Communications of the ACM*, Vol. 39, No. 10, October, 1996, pp. 104-116.

[47] W. B. Frakes, G. Succi, **An Industrial Study of Reuse, Quality, and Productivity**, *Journal of System and Software* (JSS), Vol. 57, No. 02, June, 2001, pp. 99-106.

[48] W. C. Lim, **Effects of Reuse on Quality, Productivity, and Economics**, *IEEE Software*, Vol. 11, No. 05, September, 1994, pp. 23- 30.

[49] A. Endres, **Lessons Learned in an Industrial Software Lab**, *IEEE Software*, Vol. 10, No. 05, September, 1993, pp. 58-61.

[50] M. L. Griss, **Software Reuse Experience at Hewlett- Packard**, *16th IEEE International Conference on Software Engineering* (ICSE), Sorrento, Italy, May, 1994, pp. 270.

[51] R. Joos, **Software Reuse at Motorola**, *IEEE Software*, Vol. 11, No. 05, September, 1994, pp. 42-47.

[52] G. T. Heineman, W. T. Councill, **Component-Based Software Engineering**, Addison-Wesley, 2001, pp. 818.

[53] C. Szyperski, **Component Software: Beyond Object- Oriented Programming**, Addison-Wesley, 2002, pp. 588.

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

[55] **Mobizes**, disponível em http://www.mobizines.com/index.html , janeiro 2008.

[56] **bwin Mobile**, disponível em https://www.bwin.com/page.aspx?view=mobile&select=mobile, janeiro 2008

[57] **MobUP**, disponível em http://sourceforge.net/projects/mobup/ , janeiro 2008.

[58] Java ME Open Source Softwares , disponível em http://ngphone.com/j2me/opensource/ui.htm, janeiro de 2008.

[59] **MoSOA: Mobile Service Oriented Architecture Framework**, disponível em https://sourceforge.net/projects/mosoa , janeiro 2008.

[60] Barros. T, Silva. Mauro, Espínola. E. **State MVC: Estendendo o padrão MVC para uso no desenvolvimento de aplicações para dispositivos móveis,** SugarLoaf-PLoP, 2007

[61] **UML 2.0 OCL** Specification, disponível em http://www.omg.org/docs/ptc/03-10-14.pdf , janeiro 2008.

[62] **Qualcomm Brew**, disponível em http://brew.qualcomm.com/brew/en/ , janeiro 2008.

[63] **Symbian OS**, disponível http://www.symbian.com/ , janeiro 2008.

[64] **Windowns Mobile**, disponível em http://www.microsoft.com/brasil/windowsmobile/default.mspx, janeiro 2008.

[65] **Info,** disponível em http://info.abril.com.br/aberto/infonews/102007/29102007-13.shl , janeiro 2008.

[66] Marque, L**. componentização de software em java 2 micro edition.** Trabalho de graduação. Universidade Federal de Pernambuco, Recife, Pernambuco, Brasil, 2005.

[67] MobilIT, disponível em http://www.mobilitsolutions.com/ , janeiro 2008.