FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPIDES DE MARÍLIA"
PROGRAMA DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

MARCOS AUGUSTO DE ROSSI PIVA

PLATAFORMA DE SUPORTE AO DESENVOLVIMENTO DE SOFTWARE BASEADO EM SERVIÇOS

Marília
2004

MARCOS AUGUSTO DE ROSSI PIVA

PLATAFORMA DE SUPORTE AO DESENVOLVIMENTO DE SOFTWARE BASEADO EM SERVIÇOS

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do Título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Edmundo Sérgio Spoto

Marília
2004

PIVA, Marcos Augusto De Rossi.

Plataforma de Suporte ao Desenvolvimento de Software baseado em Serviços / Marcos Augusto De Rossi Piva; Orientador: Prof. Dr. Edmundo Sérgio Spoto. Marília, SP: [s.n.], 2004. 138 f.

Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília - Fundação de Ensino Eurípides Soares da Rocha.

Palavras chave: Desenvolvimento de Software, Arquitetura Orientada a Serviços.

CDD: 004-12

MARCOS AUGUSTO DE ROSSI PIVA

PLATAFORMA DE SUPORTE AO DESENVOLVIMENTO DE SOFTWARE BASEADO EM SERVIÇOS

Banca examinadora da dissertação apresentada ao Programa de Mestrado em Ciência da Computação do Centro Universitário "Eurípides de Marília" – UNIVEM, mantido pela Fundação de Ensino "Eurípides Soares da Rocha", para obtenção do Título de Mestre em Ciência da Computação.

Resultado: **Aprovado**

ORIENTADOR: **Prof. Dr. Edmundo Sérgio Spoto**

1º EXAMINADOR: **Prof. Dr. Mario Jino**

2º EXAMINADOR: **Prof. Dr. Márcio Eduardo Delamaro**

Marília, 26 de Agosto de 2004.

AGRADECIMENTOS

Agradeço primeiro a Deus, que me deu oportunidade de estar vivo e participar das diversas coisas da vida. Aos meus Pais que me ajudaram a continuar vivendo nesse mundo cheio de imprevistos, ficando sempre ao meu lado. Aos meus irmãos Sérgio, Bela e Silvia; e ao meu sobrinho Luiz Henrique.

Ao professor Edmundo (Dino) pelo seu apoio, dedicação, compreenção, amizade, fidelidade e ao seu trabalho, orientando-me de maneira justa e sensata.

Ao professor Plínio Vilela, por ter me incentivado a desenvolver este projeto, emprestando suas idéias e dando o seu apoio quando necessário.

Ao professor Delamaro, por me mostrar que a Ciência da Comutação era algo bem mais complexo do que eu imaginava, o que me fez crescer bastante.

Aos professores Jorge, Marcos Mucheroni, Fátima e Edward, por me servirem de referência com lições de profissionalismo e decência.

Aos companheiros “Fora da Curva”, Leonardo, Claudia, Danda, Gislene, Rosiane, Adriane, Luis e Lucilena pela grande amizade que conseguimos cultivar neste período; Me sinto imensamente feliz por estar ao lado deles.

Aos amigos Gustavo Habermann, à Leninha, Érica, Juliana, professora Cristina, ao Wil, Guilerrmi, Paulo, Fernando Nerd, Ian, Victor, Capitão Rudrigu, Luiz Rogério, Cadu, Marcello Kera, Gustavo Rondina, Rodrigo (Gzus), entre outros.

À três pessoas que me acompanharam integralmente neste período, Cristina, Leonardo e Claudia; seria outra pessoa sem suas presenças.

À CAPES, por ter patrocinado grande parte deste projeto.

“É preciso força para sonhar
e perceber que a estrada vai
além do que se vê”
Marcelo Camelo

PIVA, Marcos Augusto De Rossi. **Plataforma de Suporte ao Desenvolvimento de Software baseado em serviços**. 2004. 137f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília - Fundação de Ensino Eurípides Soares da Rocha.

RESUMO

As características do desenvolvimento de software em grandes empresas levam à criação de software de difícil integração, baixo potencial de reuso e alto custo, em geral causado pela redundância na aplicação de esforços de desenvolvimento. Com o objetivo de minimizar os problemas de desenvolvimento de software em grandes empresas, novos paradigmas de desenvolvimento vêm sendo propostos. Dentre eles está a Arquitetura Baseada em Serviços, uma evolução das arquiteturas baseadas em componentes, que reduz ainda mais o acoplamento entre as diversas partes do software. O objetivo é aumentar a qualidade do produto final, a produtividade das atividades de desenvolvimento e minimizar os custos envolvidos. Esta dissertação apresenta a implementação de uma plataforma de suporte à implementação de software baseado em serviços, o Ambiente RGB Java, e o mecanismo de geração automática de serviços a partir de sua especificação em uma linguagem de alto nível, o sdlParser.

Palavras Chaves: Desenvolvimento de Software, Arquitetura Orientada a Serviços.

PIVA, Marcos Augusto De Rossi. **Plataforma de Suporte ao Desenvolvimento de Software baseado em serviços**. 2004. 137f. Dissertation (Computer Science Master's degree) - Centro Universitário Eurípides de Marília - Fundação de Ensino Eurípides Soares da Rocha.

ABSTRACT

The characteristics of the software development in large companies lead to the creation of a hard integration, low potential for reuse and high cost software, caused in general by redundancy in the application of development efforts. In order to reduce software development problems in large companies, new development and software architectures paradigms are proposed. Among them is the Service Based Architecture, an evolution of the component-based architectures which reduces the connection among the different parts of the software. Its aim is to increase the product quality and the productiveness of the development activities, as well as to reduce the costs incurred. This work is intended for presenting the implementation of a service-based software platform support, the RGB Java Environment, and the automatic service generation device from its specification in a high level language: the sdlParser

Key words: Software Development, Service-Oriented Architecture.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
BNF	Backus-Naur Form
BSD	Berkeley Software Distribution
CORBA	Common Object Request Brocker Architecture
DOM	Document Object Model
DTD	Document Type Definition
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
JavaCC	Java Compiler Compiler
JVM	Java Virtual Machine
OMG	Object Management Group
ORB	Object Request Brocker
RGB	Acrônimos de Red (vermelho), Green (verde) e Blue (azul).
RPC	Remote Procedure Call
SAX	Simple API for XML
SDL	Specification and Description Language
SOAP	Simple Object Access Protocol
SQL	Stuctured Query Language
TCP	Transport Control Protocol
UDP	*User Datagram Protocol*
UDDI	Universal Description Discovery and Integration
UML	Unified Modeling Language
URI	Uniform Resource Interface
W3C	World Wide Web Consortium
WSQL	Web Service Description Language
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language

SUMÁRIO

1. INTRODUÇÃO

1.1. Contexto

Durante as três primeiras décadas da era da computação, a principal preocupação era desenvolver hardware[1] de computador que reduzisse os custos de processamento e armazenamento de dados. O custo do hardware era predominante e a preocupação com o custo do software[2] era secundária. A partir da década de 80, os avanços na microeletrônica resultaram em maior poder computacional e de armazenamento que acabaram por contribuir para a redução dos custos envolvidos com o hardware.

> Com os computadores com um poder computacional e de armazenamento maior e com preços mais acessíveis, é natural pensar que o fator restritivo passaria a ser o software. Isso realmente aconteceu e, desde então, o software vem se tornando o componente central em muitas atividades complexas desenvolvidas em nossa sociedade. O enfoque do desenvolvimento computacional passou a ser reduzir o custo e melhorar a qualidade de soluções baseadas em computador - soluções implementadas por software. Além disso, atualmente, sistemas baseados em computação têm sido utilizados em todas as áreas da atividade humana; em conseqüência disso, aspectos de qualidade e produtividade somam-se à inerente dificuldade e à complexidade da atividade de desenvolvimento de software (MALDONADO, 1991).

Atualmente, sistemas baseados em computação têm sido utilizados em todas as áreas da atividade humana e, em conseqüência, aspectos de qualidade e produtividade somam-se à inerente dificuldade e complexidade da atividade de desenvolvimento de software; técnicas especializadas e eficientes são requeridas para a sua produção. A engenharia de software tem evoluído em resposta a essas necessidades e aplica muito dos métodos organizacionais e procedimentais da engenharia tradicional ao desenvolvimento de produtos de software.

Com o desenvolvimento das redes de computadores, o poder computacional disponível para os desenvolvedores deixou de estar exclusivamente em suas estações de trabalho[3], podendo ser distribuído na rede. Essa evolução tecnológica abre novas

[1] Equipamentos de eletrônicos.
[2] Programa de computador.
[3] Computadores com poder de executar aplicações utilizando o próprio processador.

possibilidades para o desenvolvimento de software de uma forma geral e em especial, para o software desenvolvido em grandes empresas.

Arquiteturas orientadas a serviços[4] e o paradigma de desenvolvimento de software associado a essas arquiteturas permitem reduzir o acoplamento[5] entre as aplicações (característica associada a software de qualidade) que constituem um sistema possibilitando que grupos especializados dediquem os esforços para a produção de serviços.

1.2. Motivação

O novo paradigma de desenvolvimento de software baseado em serviços propõe-se a resolver o problema do alto acoplamento contido nas ligações de um sistema a outro.

Se nas abordagens anteriores os programadores gastavam muito tempo com a integração dos sistemas, a abordagem orientada a serviços proporciona baixo acoplamento permitindo que o enfoque da comunicação entre as aplicações seja baseado na troca de mensagens do tipo texto, codificadas em estruturas de dados.

Esse baixo acoplamento é obtido com a troca de mensagens sintaticamente padronizadas por meio da Linguagem **XML**[6] (*Extensible Markup Language*), descrita mais detalhadamente no Capítulo 2 (**2.2.1**).

É possível descrever uma chamada de função, procedimento, método ou um conjunto de dados organizados em uma estrutura de dados por meio da Linguagem **XML**. Nesse formato, os dados não são codificados de forma proprietária a uma determinada linguagem de programação.

---

[4] Programa desenvolvido com propósito estrito, de conteúdo coeso, independente de plataforma, com um grau de acoplamento baixo e fácil de ser integrado aos ambientes operacionais (CILIA, et al, 2002).
[5] Referente a ligação entre os módulos, serviços, componentes ou objetos de software.
[6] Linguagem definida pela W3C, com o intuito de descrever documentos independente das linguagens de programação.

Outra característica de uma arquitetura orientada a serviços que auxilia a redução do acoplamento entre as aplicações, é que os programas são construídos para se comunicarem utilizando um ambiente. Esse ambiente gerencia a integração, fazendo com que os esforços para o desenvolvimento de software sejam reduzidos.

O baixo acoplamento surge na arquitetura orientada a serviços, pela utilização do ambiente quando todos os serviços estão preparados para serem conectados uns aos outros, dispensando formas específicas para cada integração.

Fazendo uma analogia da comunicação entre os sistemas com a comunicação realizada entre os seres humanos, pode-se notar que, por mais que duas pessoas conversem na mesma linguagem (por exemplo, o português) elas necessitam estar no mínimo falando sobre o mesmo assunto para que haja troca afetiva de informações.

Com essa relação, pode-se dizer que a ligação entre os sistemas deve obedecer a uma sintaxe e também a uma semântica. Por mais que sistemas utilizem a Linguagem **XML** como padrão de escrita na troca de informações, é necessário que a estrutura dos dados esteja organizada de uma forma que as duas partes consigam identificar as funcionalidades requeridas.

O que deve ficar evidente na abordagem orientada a serviços é que um determinado sistema necessita de outro para processar um conjunto de dados que não seria viável fazer localmente. Isto pode ser definido em poucas palavras: reuso de código.

A implementação de um ambiente de desenvolvimento de software baseado em serviços é motivada pelos seguintes assuntos enumerados: (1) reuso de Código; (2) ambiente de cooperação dentro de grandes empresas; (3) compartilhamento de recursos ou funcionalidades; (4) emprego de esforços reduzidos no desenvolvimento e; (5) desenvolvimento de software de qualidade.

1.3. Objetivo

O objetivo deste projeto é a construção do Ambiente RGB Java[7] e do Gerador de serviços RGB[8] Java, para que se contribua com a área do desenvolvimento de software baseado em serviços.

Dessa forma, foi possível escolher uma linguagem de especificação para os serviços, a Linguagem SDL[9] (*Specification and Description Language*); construir um ambiente de suporte ao desenvolvimento de software baseado em serviços, o Ambiente RGB Java e; implementar um gerador de serviços, para transformar especificações de sistemas em SDL para serviços RGB Java.

1.4. Organização

A implementação do Ambiente de desenvolvimento de software baseado em serviços foi chamada de **RGB Java**. O Ambiente é formado por três **API**s[10] (*Application Program Interface*), onde cada uma recebe o nome de uma cor: *Red*, a API para o Registro de serviços; *Green*, para a API do Provedor de serviços e; *Blue*, para a API do Cliente[11] de serviços.

Além do Ambiente RGB Java, foi desenvolvido um Gerador de serviços, utilizando a Ferramenta JavaCC[12] (*Java Compiler Compiler*), que transforma um sistema especificado na Linguagem **SDL** (*Specification and Description Language*) para uma implementação de serviços RGB Java.

---

[7] Ambiente desenvolvido neste projeto com o intuito de reproduzir as características de uma arquitetura de desenvolvimento de software baseada em serviços.
[8] Acrônimos referentes as cores Red (vermelho), Green (verde) e Blue (azul); que destina-se a fazer uma analogia a arquitetura organizada em três camadas.
[9] Linguagem utilizada para descrever e especificar sistemas com um grau de abstração elevado.
[10] Análoga às bibliotecas encontradas nas linguagem de programação do paradigma procedural.
[11] Programa que se conecta ao serviço com a finalidade de utilizar as suas funcionalidades.
[12] Ferramenta que gera os analisadores léxico e sintático, a partir de uma determinada gramática especificada na forma de Backus-Naur (**BNF**).

O desenvolvimento deste trabalho é dividido em três partes: a revisão bibliográfica (**Capítulo 2**), que trata dos assuntos que deram base ao desenvolvimento do projeto; a implementação do Ambiente RGB Java e do Gerador de serviços (**Capítulo 3**) e; dois estudos de caso, mostrando a utilização do Ambiente RGB Java e do Gerador de implementação de serviços (**Capítulo 4**).

2. REVISÃO BIBLIOGRÁFICA

A revisão bibliográfica teve o objetivo de encontrar tecnologias e padrões que dessem suporte ao projeto de uma arquitetura orientada a serviços. Ao longo desse processo, descobriu-se que, referente a esse assunto, não existe um consenso geral entre os grupos de pesquisa.

Coyle (2002) diz que é melhor utilizar o protocolo **HTTP**[13] (*Hypertext Transfer Protocol*) no tráfego de mensagens entre serviços e aplicações; o CORBA (*Common Object Request Broker Architecture*), para o mesmo propósito, utiliza um protocolo situado em uma camada mais baixa (mais básica), o TCP/IP[14] (*Transport Control Protocol / Internet Protocol*) (RICCIONI, 2000); Agrawal, et al (2002) propuseram o Vinci modificando a sintaxe das mensagens de **XML** para **Xtalk**, um formato proprietário.

Wiegers (1998) critica a necessidade de novos modelos de desenvolvimento de software, argumentando que as empresas ainda necessitam utilizar aqueles já propostos e considerados bons.

A justificativa dada por Wiegers baseia-se na falta de estímulos para a utilização dos modelos já propostos, afirmando que, somente devem ser criados outros modelos quando nenhum mais satisfaz os interesses do projeto.

O mesmo autor diz que menos de 10% dos membros da audiência da **IEEE**[15] (*Institute of Electrical and Electronics Engineers*) em engenharia de software, têm acessado a coleção de padrões da **IEEE** de engenharia de software. Esses dados mostram que o problema não é a falta de padrões adequados e sim a falta de conhecimento dos mesmos.

---

[13] Protocolo de rede localizado na camada de apresentação.
[14] Protocolo de rede localizado na camada de transporte.
[15] Instituto responsável por catalogar informações sobre padrões e inovações de caráter científico, referente a engenharia elétrica e eletrônica; oferece suporte aos seus membros por meio de serviços.

A intenção de projeto de uma arquitetura baseada em serviços é compatível com a idéia de Wiegers, pois os padrões são incluídos e não reinventados, utilizando formatos padronizados e bem difundidos.

Em relação ao projeto, sem esquecer que se trata de um novo paradigma de desenvolvimento de software, um novo modelo é mais adequado, pois a novidade trará não só mais uma metodologia e sim uma maneira mais aberta em que programas se comunicarão entre si.

Essa comunicação entre os programas independe da complexidade gerada pela diversidade dos ambientes. É baseada na troca de mensagens com um formato bem difundido, aplicáveis em diferentes sistemas operacionais e linguagens de desenvolvimento de software.

Esse tipo de abordagem faz com que os programas deixem de utilizar uma comunicação estabelecida por ligações "rígidas"[16], construídas com base nos seus próprios propósitos, para se apoiarem em um ambiente que gerencie a comunicação.

No momento em que as aplicações puderem estabelecer uma comunicação baseada em um ambiente, o foco do problema deverá ser o conteúdo das mensagens trocadas entre elas.

A pesquisa de Coyle (2002) refere-se a uma nova forma de se construir software utilizando caminhos simples e considerados consagrados.

Coyle (2002) define como elementos fundamentais de uma aplicação distribuída, o protocolo **HTTP** e a Linguagem **XML**, argumentando que é possível centrar o desenvolvimento de software mais nos dados do que nas linguagens de programação e nos ambientes dos sistemas operacionais.

[16] Referente a um grau de acoplamento elevado.

A idéia de Coyle é simplesmente manter o protocolo de comunicação **HTTP** como base para as trocas de mensagens entre máquinas e utilizar a Linguagem **XML** para estruturar os dados. Dessa maneira, o enfoque de uma comunicação estaria mais na estrutura dos dados do que na ligação entre os programas.

A diferença entre o que Coyle (2002) propôs e o desenvolvimento de serviços baseado em uma arquitetura é exatamente a arquitetura, que desta forma, estabelece e gerencia a comunicação entre duas aplicações.

Construir software segundo Coyle, seria algo amarrado a uma programação mais complexa; por mais que os programas tenham a capacidade de se conectar por estarem utilizando um padrão de escrita das mensagens, o desenvolvimento ainda teria que se ocupar com a construção de uma “ponte rígida” para ligar os programas.

O desenvolvimento de software baseado em uma arquitetura de serviços tem como visão a formatação das mensagens que devem ser trocadas entre os programas. A arquitetura se responsabiliza pela conexão, pelo gerenciamento das ligações, tanto quanto pela decodificação das mensagens sintaticamente formatadas.

Cilia, et al (2002) em suas pesquisas relacionadas a banco de dados, rebusca a idéia de uma arquitetura orientada a serviços para distribuir funcionalidades ativas em aplicações de comércio eletrônico. Além da arquitetura de serviços, Cilia utiliza a idéia de ontologias e disparo de eventos distribuídos.

A infraestrutura de ontologias possibilita o compartilhamento de um dicionário de termos entre os participantes, ajudando na troca dos eventos.

Por mais que exista uma padronização pela arquitetura orientada a serviços e um formato de mensagens, como as escritas em **XML**, segundo Cilia, é importante que as

mensagens tenham o mesmo significado nos diferentes pontos da rede, tornando uma necessidade a utilização, do que o autor chama, de um dicionário semântico (ontologia).

As computações, seguindo a mesma idéia dos **Web Services**[17], seriam distribuídas em "pequenas quantidades", de forma que cada pedaço de código distribuído caracterizaria-se pela realização de uma tarefa bem definida, coesa.

O alto nível de coesão, caracterizada nos serviços, permite que a reutilização de código seja feita de maneira mais eficiente, devido à redução da complexidade embutida no software.

Já o baixo acoplamento é caracterizado pela comunicação entre aplicações, utilizando uma arquitetura. Dessa forma, os programas não se conectam de maneira "rígida", baseando-se na troca de mensagens codificadas em um determinado padrão sintático.

Com isso, todos os programas que fossem projetados baseados na arquitetura, poderiam compartilhar as suas funcionalidades, com baixo custo de integração.

Essa idéia é suficientemente bem abordada por Agrawal, et al (2002), referindo-se a este tipo de conexão como *loose-coupling*, ou acoplamento "frouxo" (baixo acoplamento).

As tecnologias, padrões e ferramentas descritas em seguida neste capítulo serviram de base para a criação do Ambiente de desenvolvimento de software e do Gerador de serviços RGB Java.

2.1. Componentes e Objetos Distribuídos

Os componentes de software são baseados no paradigma orientado a objetos e servem para disponibilizar objetos, já prontos, com um certo grau de complexidade, para serem reutilizados (SUN MICROSYSTEMS, 2002).

[17] Serviços Web; Ambientes que utilizam a Web (a internet) como meio para a comunicação de serviços.

Já os objetos distribuídos são programas instanciados remotamente, acessíveis através de uma rede de computadores. Estes objetos trocam mensagens de uma forma parecida com a dos que se relacionam localmente, utilizando a computação distribuída para fazer reuso de código (RICCIONI, 2000).

A transmissão dos dados entre objetos remotos é realizada por chamadas de métodos com passagem de parâmetro, como se estivessem sendo executados na mesma máquina.

A invocação de um método pertencente a um objeto remoto é feita pelo envio dos dados em uma lista de parâmetros. O objeto remoto executa a computação especificada no método retornando o valor do processamento.

Como a transmissão dos dados entre objetos remotos é realizada por passagem de parâmetros ordenando, a ligação entre estes objetos se torna fortemente acoplada, *tightly coupled* (AGRAWAL, et al, 2002).

É importante distinguir os paradigmas de desenvolvimento de software pelo grau de acoplamento entre as aplicações. O acoplamento no paradigma baseado em componentes ou em objetos distribuídos apresenta grau mais alto do que aquele baseado em serviços.

2.1.1. JavaBeans

JavaBeans (SUN MICROSYSTEMS, 2002) (HORSTMANN, 2002) é uma tecnologia de desenvolvimento de componentes, que auxilia o desenvolvimento de software, disponibilizando objetos prontos para o reuso.

Por meio dela um programador de sistemas tem a capacidade de reutilizar funcionalidades especificadas nos componentes, reduzindo os custo de implementação de um produto.

Um *bean*, um componente Java, é composto por propriedades, métodos e eventos. As propriedades são os atributos, valores de um objeto; os métodos, as computações especificadas para a execução das funcionalidades; e os eventos, as mensagens que notificam outros objetos sobre alguma ocorrência predefinida.

Por meio desses três elementos: propriedades; métodos e eventos, um desenvolvedor estrutura o componente para que ele seja disponibilizado para o reuso por um programador de sistemas.

A programação orientada a componentes é uma das frentes de desenvolvimento de software que vem crescendo nos últimos anos, pelo uso de *Delphi*, *Visual Basic*, *Visual C++* e, por último, mas não menos importante o Java.

2.1.2. CORBA

A tecnologia de objetos distribuídos **CORBA** (*Common Object Request Broker Architecture*), especificada pela OMG (*Object Management Group*), oferece um protocolo de comunicação para possibilitar que diversos programas escritos em diferentes linguagens de programação orientada a objetos troquem dados pela invocação de métodos remotos (RICCIONI, 2000).

O que liga os objetos de diferentes linguagens ou plataformas, é uma Linguagem de Definição de Interfaces (**IDLs**). Esta linguagem permite que uma mensagem enviada de um objeto escrito em Java, por exemplo, seja convertida para o formato dos objetos especificados em C++.

O conversor transforma uma mensagem, escrita em uma determinada linguagem, em um formato comum, transmitindo-a pelo canal **ORB** (*Object Request Broker*). Ao chegar a

seu destino, a mensagem é novamente convertida para um outro formato específico, no qual deverá ser interpretada.

A diferença entre serviços e objetos distribuídos é exatamente o custo desta conversão. Os serviços comunicam-se utilizando uma linguagem comum (**XML**); dessa forma, o acoplamento entre eles é baixo (COYLE, 2002).

Utilizando o XML, as mensagens são formatadas como texto, podendo ser lidas em qualquer linguagem de programação e em qualquer plataforma. Um texto é compreendido em qualquer ambiente. O protocolo de comunicação pode ser algo básico como **TCP/IP** ou até mesmo **HTTP**.

Caso a mensagem codificada em **XML** não faça "sentido" em um serviço, este então, poderá responder, por exemplo, com uma mensagem contendo *"<error> functionality not found </error>*", sem que o programa encerre a execução.

Se o método de um determinado objeto remoto é acrescido de um parâmetro de entrada, a invocação antiga causará um erro enquanto que, com serviços, a funcionalidade pode ser executada da mesma forma, mesmo faltando um parâmetro, se para esse caso, a informação não for algo essencial para a sua execução.

2.2. *Web Service*

> Um *Web Service* é um sistema de software identificado por uma **URI**[18], cujas interfaces públicas e ligações são definidas e descritas usando **XML**. Sua definição pode ser descoberta por outros sistemas de software. Estes sistemas podem então interagir com o *Web Service* de uma maneira prescrita pela sua definição, usando mensagens baseadas em **XML** e transportadas através de protocolos de internet (W3C, 2002).

Segundo Coyle (2002), a evolução para o *Web Service* permite que o foco das empresas esteja apontado para as negociações e não para o transporte dos dados, onde programas podem interagir uns com outros sem que os usuários tomem conhecimento.

[18] *Universal Resource Interface* – endereço que identifica uma instância de serviço.

2.2.1. XML

XML, *Extensible Markup Language*, segundo **W3C**[19] (2004), é uma linguagem designada para descrever e estruturar documentos. Diferente do **HTML**, ela pode conter marcações (tags) adicionadas conforme a necessidade da estruturação dos dados, ou seja, é altamente dependente do contexto.

Para que aplicações especificadas em diferentes linguagens de programação sejam capazes de comunicar-se, o **XML** comporta-se como uma linguagem de plataforma neutra, transmitindo somente dados em forma de texto.

Nas pontas de uma conexão do tipo ***Socket TCP*** (**2.3.2**) por exemplo, programas que decodificam esses documentos são ligados às linguagens de maneira específica. Esses programas se chamam *parsers*[20].

Existem dois tipos de *parser* propostos para **XML** e bem difundidos: **DOM** (*Document Object Model*) e o **SAX** (*Simple API for XML*). Estes tipos de *parsers* são implementados e usados de forma diferente, cada um com suas vantagens e desvantagens dependentes do ambiente de aplicação.

O **SAX** é indicado para documentos extensos, pois não utiliza a memória principal de uma máquina como fonte repositória dos dados decodificados, lendo-os seqüencialmente.

O **DOM** necessita que um documento esteja completo em memória e, se o documento for muito extenso, poderá causar problemas referentes ao desempenho, pois após a decodificação, os dados ficam dispostos em forma de árvore na memória principal e podem ser acessados pelo programador.

---

[19] Consórcio com a intenção de definir e organizar os padrões utilizados na *Web*.
[20] Programas que analisam lexicamente e sintaticamente outros programas, com base na gramática específica de uma linguagem.

Cada tipo de *parser* tem suas vantagens e desvantagens: o **DOM**, por exemplo, utiliza muita memória **RAM**[21] (*Random Access Memory*) e é mais complicado que o **SAX**, para ser manuseado. Por outro lado, torna a utilização dos dados decodificados mais ágil.

O **SAX** dá ao programador, a cada iteração, informações de um elemento (nome da tag, conteúdo e atributos), sem os guardar em memória. Esta característica torna o **SAX** um *parser* mais fácil de se lidar, por não exigir do programador o conhecimento aprofundado de estrutura de dados.

As mensagens codificadas em **XML** que trafegam entre os serviços no Ambiente RGB Java utilizam como decodificador o *parser* **SAX**, que é implementado na Linguagem Java em forma de **API**. Por ser considerado um padrão de desenvolvimento de software bem difundido, atualmente as linguagens de programação possuem bibliotecas que permitem a utilização da Linguagem XML através de implementações de *parsers*.

XML atende necessidades da computação distribuída como: independência de linguagem e plataforma, por trafegar somente uma representação neutra e sintaticamente padronizada; e descarga do tráfego de rede, por resolver toda a complexidade da sua codificação e decodificação nas duas pontas, utilizando os *parsers* e transmitindo os dados no seu formato textual.

Na validação de documentos feita pelos *parsers*, um documento é "bem formado" quando obedece às regras sintáticas e um documento é válido quando está em conformidade com as regras de um **DTD** (*Document Type Definition*) e é "bem formado".

[21] Memória volátil; que armazena os dados de maneira temporária; auxilia a memória secundária (não volátil) na execução dos programas; comunica-se diretamente com o processador, de forma ágil; geralmente escassa; com custo elevado.

A finalidade do **DTD** é definir uma lista de elementos possíveis para um documento, assim como um dicionário léxico, formalizando as palavras pertencentes à linguagem, neste caso, a mensagem.

Como as mensagens escritas em **XML** não podem ser interpretadas diretamente por um navegador, a **W3C** (*World Wide Web Consortium*) propôs uma linguagem de formatação para **XML**, o **XSL** (*Extensible StyleSheet Language*), com o intuito de cuidar da apresentação dos dados contidos em um documento **XML**.

Como a Linguagem **XML** nesse projeto é utilizada pelas aplicações, diretamente relacionadas com os ambientes de desenvolvimento, a Linguagem **XSL** não será amplamente abordada, pois ela é utilizada pelos aplicativos navegadores, browsers.

2.2.2. WSDL

Web Service Description Language é uma linguagem baseada em **XML** projetada para descrever serviços Web: como os serviços funcionam, onde localizá-los e como invocá-los (BECKER, et al, 2001).

Um documento escrito em **WSDL** é dividido em quatro partes: **message** e **portType**, definindo quais operações um serviço fornece; **type**, definindo tipos de estrutura de dados; **binding**, servindo para mostrar como as operações são invocadas; e **service**, explicando onde o serviço está localizado.

WSDL é uma linguagem para especificar documentos contendo informações dos serviços, que podem ser emitidos a um registro de serviços, para que sejam armazenadas, centralizadas e disponibilizadas a partir de uma chamada de conexão.

2.2.3. UDDI

Universal Description Discovery and Integration tem como intuito principal servir de repositório para dados referentes a serviços Web. Os tipos de dados encontrados em um registro **UDDI** são: **businessService**, informações gerais sobre a organização; **businessEntity**, informações técnicas e descrição de serviços (pode conter um ou mais **bindingTemplate**); **bindingTemplate**, contém uma ou mais referências para um **tModel**, sendo este o que define as informações específicas de um serviço.

UDDI dá suporte a descrições de serviços feitas em **WSDL** e outras linguagens de descrição de serviços.

2.2.4. SOAP

Simple Object Access Protocol é um protocolo baseado em **XML** que permite comunicação entre aplicações utilizando o protocolo **HTTP**. Um documento **SOAP** contém três partes, descritas a seguir.

SOAP Envelope define o conteúdo da mensagem e os vários *namespaces* que são usados pelo resto da mensagem; **SOAP Header** (opcional) contém informações a respeito da autenticação, transação e contabilização; e **SOAP Body** contém informações a respeito de métodos e parâmetros a serem chamados ou respostas enviadas.

Quando o **SOAP** é utilizado com RPC[22] (*Remote Procedure Call*) esta parte possui um único elemento que contém o nome do método, os parâmetros e a identificação do serviço, assemelhando-se ao **CORBA**.

[22] Chamada de Procedimento Remota. Utilizada pelas linguagens de programação de maneira semelhante as chamadas de procedimento tradicionais. Implementada para que um programa se ligue a outro em execução em uma outra máquina da rede, por meio de uma chamada de procedimento.

Apesar dos tópicos acima serem pertinentes ao projeto (**WSDL**, **UDDI** e **SOAP**), não foram aprofundados pois o desenvolvimento desses levaria a uma discussão sobre *Web Services*. O projeto está relacionado ao desenvolvimento de uma arquitetura de software baseada em serviços e um gerador de serviços.

2.3. Arquitetura Orientada a Serviço

O conceito de arquitetura orientada a serviços, segundo Stal (2002); Gisolfi (2002); Agrawal (2001); Coyle (2002) e; Cilia, et al (2002), baseia-se na forma como as aplicações, em estado de execução numa determinada rede de computadores, se interconectam.

Esta forma de ligação é caracterizada pelo baixo acoplamento. Os programas se conectam por meio da arquitetura para que se possa reduzir os custos de integração das partes que necessitam trocar informações.

Dessa maneira, um programa deve ter a capacidade de se conectar ao Ambiente de serviços sem que necessite tomar conhecimento da localização destes, na rede.

Além do baixo acoplamento, o projeto de uma arquitetura orientada a serviços deve ter como foco a reutilização de código, possibilitada pelo alto grau de coesão com que os serviços devem ser construídos.

Com as tecnologias de objetos distribuídos e componentes de software, a reutilização de código é realizada com um alto acoplamento entre as aplicações. Os serviços, por utilizarem a Linguagem **XML** ao invés da invocação de métodos, tornam menos acoplados as ligações entre as partes.

Segundo Stal (2002), uma arquitetura de software deve ajudar os programadores de sistemas, escondendo a maior parte da complexidade existente na transmissão dos dados entre um programa e o outro remoto.

Para transmitir dados entre aplicações dispostas em uma rede, é necessário que se tenha conhecimento sobre diversas tecnologias e padrões. Fazer com que os desenvolvedores de software não tenham que se preocupar com essas questões e sim com o conteúdo das mensagens trocadas, reduz o esforço de desenvolvimento.

Segundo Coyle (2002), as empresas podem ter um ganho nas transações de negócios, abstraindo características pertinentes ao transporte dos dados pela rede, dando mais ênfase aos negócios.

Com isso, pode-se entender que serviços são instâncias[23] distribuídas que possuem entrada, processamento e saída, como qualquer programa comum. A grande diferença entre um serviço e programas comuns é que a sua associação com um ambiente é feita de forma "solta", com baixo acoplamento (COYLE, 2002).

Os serviços podem desempenhar um papel muito importante no desenvolvimento de software, trazendo o reuso de código e uma implementação mais segura e menos custosa. Se o desenvolvimento de um serviço for consolidado com técnicas mais apuradas de teste, por exemplo, um Cliente consegue ter uma fatia do seu código que não necessitará de teste funcional, somente de integração.

Mesmo sendo um paradigma de desenvolvimento de software com muitas vantagens, o desenvolvimento baseado em serviços apresenta também os problemas, que serão descritos a seguir.

Descrever um serviço ainda não possui um consenso geral; o padrão proposto para Web Service é o **WSDL**, ainda que de uma maneira incompleta, pois o **UDDI** necessita de mais atributos do que os encontrados no **WDSL** para registrar um serviço.

[23] Programas em estado de execução.

Buchmann (2002) relata em seu artigo sobre funcionalidades ativas de banco de dados algo parecido com a descrição de serviços. No caso, ele preocupou-se em atender o problema da semântica dos eventos distribuídos, espalhando funcionalidades ativas na forma de serviços.

Para resolver o problema do conhecimento compartilhado, ele propõe um dicionário de termos (ontologia) onde é disponibilizado um único termo para identificar as funcionalidades.

A semântica, segundo Cilia, et al (2002), é a chave para distinguir um serviço do outro. Cilia aplicou a semântica no ambiente para identificar as mensagens trocadas entre as aplicações. Dessa maneira, a mesma idéia pode ser utilizada para identificar serviços.

2.3.1. Vinci, um Ambiente de desenvolvimento de software baseado em serviços.

Agrawal, et al (2002) propuseram um ambiente de desenvolvimento de software baseado em serviços, juntamente com a **IBM**, chamado Vinci.

O artigo descreve o ambiente baseando-se no desenvolvimento de serviços, tanto para internet como para redes empresariais, apresentando a estrutura de maneira simplificada.

Para a troca de mensagens entre os serviços, o Vinci utiliza um padrão proprietário, ou seja, criado exatamente para o seu propósito, chamado **XTalk**, simplificando a Linguagem **XML**. A justificativa para essa criação é baseada na decodificação das mensagens trocadas pelos serviços, afirmando que com o **XTalk** essa decodificação é efetuada com uma melhor desempenho.

Sob o ponto de vista da utilização dos padrões, o Vinci traz grandes problemas para sua utilização, pois sua implementação é baseada em padrões proprietários. Desta maneira, o

Vinci torna-se um Ambiente de difícil integração com os outros conceitos já amplamente difundidos.

Segundo Moore (2002), os ambientes comerciais, devido à concorrência, desenvolvem softwares com funcionalidades aparentemente melhores do que os outros que são baseados em padrões bem difundidos. Essa prática traz problemas referentes à integração dos produtos, quando a falta de utilização dos padrões é freqüente.

Os padrões podem desempenhar o papel de fornecer as informações exatas, considerando a compatibilidade dos produtos, tornando mais simples a concepção das inovações, por serem baseados em padrões bem difundidos.

Apesar dessa característica, de utilizar uma linguagem de descrição de mensagens proprietárias, outros dois padrões referentes a *Web Service* são acoplados na arquitetura do Vinci: o **SOAP** (**2.2.3**) e o **UDDI** (**2.2.4**).

A utilização do Ambiente RGB Java e do Vinci é feita de maneira semelhante, diferenciando-se em dois aspectos fundamentais: um serviço RGB Java que é baseado na especificação de um sistema **SDL**, e o Vinci que contém bibliotecas para que se possa desenvolver serviços e programas clientes de serviços em diversas linguagens.

2.3.2. Sockets TCP

O protocolo **TCP** (*Transport Control Protocol*) localiza-se na camada de transporte da rede IP e tem a função de empacotar campos que permitem uma comunicação mais segura, ou seja, com controle de entrega de pacotes IP.

Sockets são implementados em aplicações para servirem como interface de uma comunicação remota.

É possível utilizar dois tipos de **Sockets**: **TCP** e **UDP** (*User Datagram Protocol*). Como dito acima, o **TCP** (utilizado pelo Ambiente RGB Java) tem uma característica peculiar; ele organiza os pacotes **IP** recebidos por uma aplicação remota por meio de avisos de recebimento; já o **UDP** não implementa essa característica.

Se por acaso um pacote for perdido no meio de uma transmissão, o aviso do recebimento não acontecerá, fazendo com que o emissor retransmita até acontecer um aviso do recebimento.

Para conectar duas aplicações instanciadas em máquinas distintas, porém localizadas em uma mesma rede, é necessário saber o endereço **IP** e a porta do sistema operacional dessas aplicações. Essas informações, o endereço **IP** e a porta, apontam para um específico processo em execução em uma máquina da rede.

Passam-se essas informações pelo **Socket** e a conexão é feita. Dessa maneira, as mensagens entre aplicações remotas podem ser trocadas, usando-se o canal estabelecido (TANEMBAUM, 2002) (SILBERSCHATZ, 2001).

Os **Sockets TCP** podem ser facilmente implementados nas linguagens de programação como Java, C, C++ e outras. Por esta característica e outras já citadas, esse protocolo foi escolhido como o meio de comunicação entre partes da arquitetura desenvolvida.

2.4. Linguagem de Descrição e Especificação de Sistemas – SDL

A Linguagem **SDL** foi definida pela **CCITT** (*Comité Consultatif International Telegraphique at Telephonique*), atual **ITU** (*International Telecommunication Union*), com a intenção de solucionar problemas de especificação de serviços para ambientes de telecomunicações.

Desde que as redes de computadores tornaram-se ambientes complexos, por suportarem inúmeras funções, a definição, especificação e implementação de serviços e protocolos tenderam a ser realizadas de maneira trabalhosa (ELLSBERGER, 1997).

A utilização da Linguagem **SDL** é adequada, no âmbito da sua definição, para resolver problemas de especificação de aplicações relacionadas com telefonia, por prover um nível de abstração da implementação de produtos de software.

Contudo, sua estrutura possibilita a especificação de serviços distribuídos em uma rede, que tem como característica principal os serviços, o baixo acoplamento e um nível elevado de coesão.

Especificar serviços com um nível elevado de abstração permite que o seu desenvolvimento seja baseado no entendimento das suas funcionalidades, reduzindo o grau de complexidade do projeto de software.

Os problemas pertinentes à implantação de um serviço em ambientes complexos podem ser resolvidos com a utilização de plataformas de desenvolvimento de software, assim como a proposta neste projeto de mestrado, o Ambiente RGB Java.

Os benefícios da utilização da Linguagem **SDL** em projetos de software, segundo Ellsberger, são: conceituação bem definida; especificações concisas, limpas e não ambíguas; habilidade para a correção de especificações complexas; habilidade para checar aspectos da implementação com os da especificação; habilidade para checar a consistência da especificação; adequação para o uso de ferramentas de desenvolvimento baseadas em computação para validar, verificar, manter, analisar e simular.

Como um sistema pode ser especificado em **SDL** em diferentes níveis de abstração, sua análise pode ser segmentada, auxiliando as tomadas de decisão nos detalhamentos e nos níveis mais abstratos.

Uma especificação **SDL** concentra-se em um sistema, System, sendo subdividida em blocos que agregam processos. Os processos descrevem o comportamento do sistema, baseando-se em uma máquina de estados finitos.

Os blocos comunicam-se com o ambiente ou com outros blocos, através de canais de entrada e saída. Esses canais transportam sinais, que descrevem o conteúdo da comunicação.

Os sinais funcionam como estímulos, dentro dos processos, possibilitando a transição dos estados, respeitando a gramática imposta pelas entradas.

Um estado recebe sinais, os sinais efetuam transições com base no conjunto de entradas especificadas para o estado. Caso um estado receba um sinal x, a transição ocorrerá se a entrada x tiver sido especificada para o estado.

Cada entrada gera um ramo, uma mudança no fluxo de controle, que acaba com uma transição de estado (a instrução *nextstate*) ou com um desvio incondicional (a instrução join).

O ramo é o intervalo que possibilita a especificação de outras instruções **SDL**, como: tarefas (*task*), saídas (*output*), estruturas de decisão (*decision*), rótulos (*label*), desvios incondicionais (*join*), etc.

Uma *task* é usada para especificar uma computação, um trecho de código. O conteúdo de uma tarefa não necessita estar relacionado com a Linguagem **SDL**, sendo assim uma área livre de interpretação. Uma *task* pode especificar uma conexão de banco de dados, uma soma de variáveis, comentários ou outras instruções de qualquer linguagem.

A especificação de um *output* (saída) faz com que um sinal seja enviado, utilizando um canal ou uma rota de sinal. As rotas de sinais são canais que interligam os processos e os canais, as estruturas que ligam os blocos ao ambiente.

As estruturas de decisão possibilitam que valores ou variáveis sejam comparados, da mesma forma que ocorre com a instrução *if* (se) na Linguagem Pascal ou Java.

Os rótulos e desvios incondicionais trabalham em conjunto. Um rótulo marca a posição no ramo e o desvio retrocede no fluxo de controle, até que se atinja o rótulo. *Labels* e *joins* podem ser usados para especificar uma estrutura de repetição.

Para que se consiga especificar uma estrutura de repetição condicionada ao valor de uma variável deve-se associar a essa uma estrutura de decisão.

As instruções "*set*" e "*reset*" são utilizadas para que sinais sejam ativados dentro do mesmo processo, onde o *set* valida o sinal e *reset* o invalida.

A instrução "dcl", utilizada para a declaração de variáveis, deve ser especificada antes do estado "*start*", que representa o início da execução do processo.

O gerador de serviços RGB Java suporta, por meio dos processos **SDL**, a especificação das instruções *label*, *join*, *decision*, *task*, *output*, *input*, *nextstate*, *state*, *set*, *reset*, e *dcl*.

2.5. JavaCC

A Ferramenta JavaCC, desenvolvida pela Sun Microsystems, atualmente disponível pelo projeto **Open Source**[24], sob a licença do **BSD**[25], tem como funcionalidade principal gerar analisadores léxicos[26] e sintáticos[27], na Linguagem Java.

Um *parser* é um gerador sintático de código que pode ser construído a partir de uma gramática. O JavaCC utiliza a forma de *Backus-Naur* (**BNF**) para que se possa definir uma gramática da linguagem, que será a entrada para o *parser*.

[24] Projeto destinado a difusão do Código Aberto.
[25] *Berkeley Software Distribution* (Distribuição Linux) – Projeto de desenvolvimento de software baseado em Código Aberto.
[26] Análise baseada no dicionário de palavras definidos pela linguagem.
[27] Análise baseada nas estruturas, sintaxes, permitidas pela linguagem.

A Ferramenta JavaCC foi utilizada neste projeto com o intuito de auxiliar o desenvolvimento de um Gerador de serviços, que transforma um sistema especificado na Linguagem SDL em serviços RGB Java.

Para compreender melhor a Ferramenta JavaCC foi utilizado um material didático (DELAMARO, 2003), onde foi possível estudar a construção das gramáticas empregadas no *parser* SDL, desenvolvido por Vilela (2002).

2.6. Considerações Finais

Os assuntos abordados neste capítulo foram de extrema importância pra desenvolvimento do projeto, de modo a toma-los como base para a construção do Ambiente RGB Java e do Gerador de Serviços RGB Java.

A construção do projeto de uma arquitetura de desenvolvimento de software baseado em serviços fundamentou-se nos trabalhos, idéias e conceitos de autores como Coyle (2002), Cilia, et al (2002), Wiegers (1998), Agrawal, et al (2001), Bosak (2002), Moore (2002) e Stal (2002).

A tecnologia de componentes e objetos distribuídos serviram de base para a evolução do novo paradigma de desenvolvimento de software, o baseado em serviços. Dessa forma, os conceitos sobre componentes e objetos distribuídos foram aproveitados para a compreensão do paradigma que antecedeu o baseado em serviços; a compreensão do CORBA auxiliou no desenvolvimento do projeto, por ter uma intenção análoga a de um ambiente de desenvolvimento de software baseado em serviços.

Baseado na arquitetura dos *Web Services* foi possível compreender a mudança do paradigma orientado a componentes para o orientado a serviços; substituindo os objetos e componentes pelos serviços; a invocação destes não se daria na forma de uma especificação

em uma linguagem de programação e sim pelo conteúdo de mensagens, estruturada conforme um padrão de escrita, o XML, bem difundido.

Como a comunicação entre os serviços não é feita por meio de uma linguagem de programação específica, a arquitetura deve ter a funcionalidade de resolver os problemas de conectividade entre os serviços; pensar em uma arquitetura de software baseada em serviços é transformar em um nível mais abstrato, a complexidade da comunicação entre serviços distribuídos em uma rede de computadores.

A partir do ambiente Vinci, desenvolvido por Agrawal, et al (2002), foi possível compreender na prática os conceitos de uma arquitetura de software baseada em serviços; como o Ambiente RGB Java foi desenvolvido na sua forma mais básica, até este momento, é inviável comparar detalhadamente os dois ambientes.

O projeto do Ambiente RGB Java foi baseado no desenvolvimento de software, portanto a utilização de uma Linguagem de Descrição e Especificação de Sistemas, o **SDL**, auxiliou no requisito “facilidade de escrita” dos Serviços RGB Java, por ser uma linguagem que abstrai os aspectos específicos da programação em si, delegando ao Gerador de Serviços a transformação da descrição abstrata do serviço em uma linguagem mais próxima do ambiente operacional.

Para a construção do Gerador de Serviços RGB Java foi utilizada a Ferramenta JavaCC, que transforma produções de uma gramática na forma de *Bachus-Naur* (**BNF**) em analisadores léxicos e sintáticos implementados na Linguagem Java.

Desta forma, as tecnologias, os padrões, as ferramentas e os conceitos investigados no decorrer do projeto, cada de uma forma específica, contribuiu para o desenvolvimento do Ambiente RGB Java e o Gerador de Serviços RGB Java.

3. IMPLEMENTAÇÃO DO AMBIENTE RGB JAVA

Neste capítulo, são discutidos os aspectos da implementação do Ambiente **RGB** Java e do Gerador de Serviços.

3.1. Descrição Geral do Ambiente

O Ambiente RGB Java, simbolicamente representado pela **Figura 1**, é formado por três partes fundamentais: Registro de Serviços, Provedor de Serviços e Cliente de Serviços.

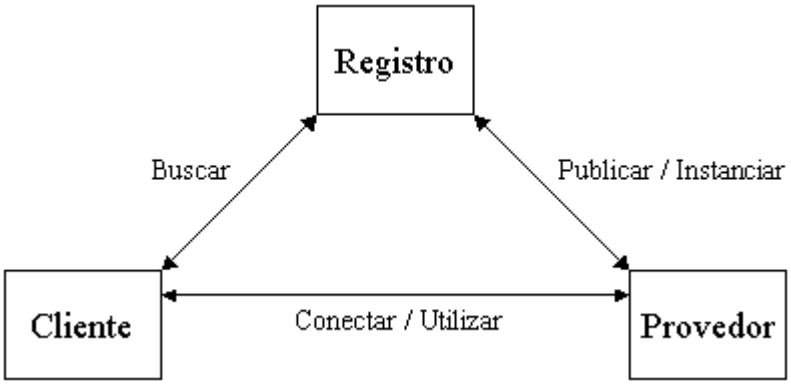

Figura 1: Componentes fundamentais do Ambiente RGB Java.

Essas três partes formam os pilares de um Ambiente do desenvolvimento de software orientado a serviços (**2.3**). Ao longo deste trabalho, a palavra “Ambiente” significará exatamente a relação entre os componentes citados.

Além das partes descritas acima, existe outro módulo que se liga ao Ambiente por meio do Provedor de serviço, o Gerador de serviços, para fornecer implementações dos serviços. Essas implementações compõem as funcionalidades do Ambiente.

O Gerador de serviços transforma uma especificação de sistema em **SDL** para Java, que é utilizada pelo provedor para que se instancie os serviços no ambiente, ou seja, o Gerador de serviços gera os serviços que deverão ser utilizados no Ambiente RGB Java (**Figura 2**).

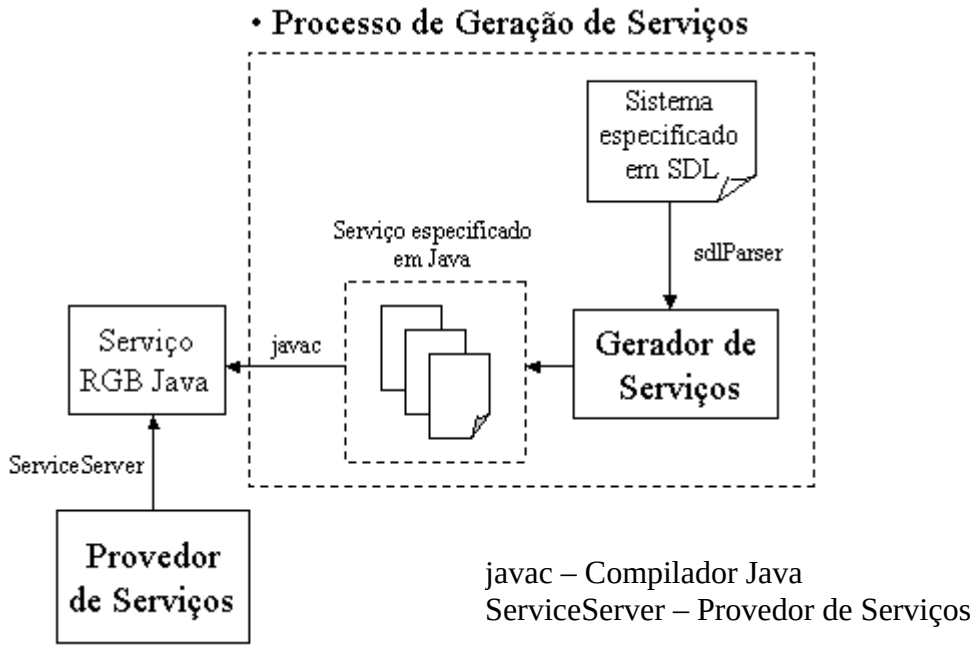

Figura 2: Processo de geração de serviços para o Ambiente RGB Java.

Essa funcionalidade do Gerador de serviços torna-o parte do ambiente, pois somente com a sua utilização é possível obter os serviços RGB Java.

A **Figura 3**, ilustra as funcionalidades do Ambiente RGB Java, utilizando o diagrama de casos de uso, *use case*, da Linguagem **UML**[28] (*Unified Modeling Language*).

O diagrama acima, descreve seis funcionalidades primárias, nas quais o projeto se apóia para assumir a implementação de um Ambiente de desenvolvimento de software baseado em serviços.

O ator Desenvolvedor de Serviços, interage com o sistema para gerar serviços RGB Java e os publica, para que outros atores possam localizá-los.

Um Administrador de Serviços, pode instanciar e terminar a execução dos serviços, dentro do Ambiente RGB Java. O Ambiente foi projetado para que as funcionalidades de "instanciar serviços" e "terminar serviços" desenvolvam tarefas dentro do Registro de Serviços.

[28] Linguagem destinada a especificação do projeto de software; utilizada por meio de diagramas.

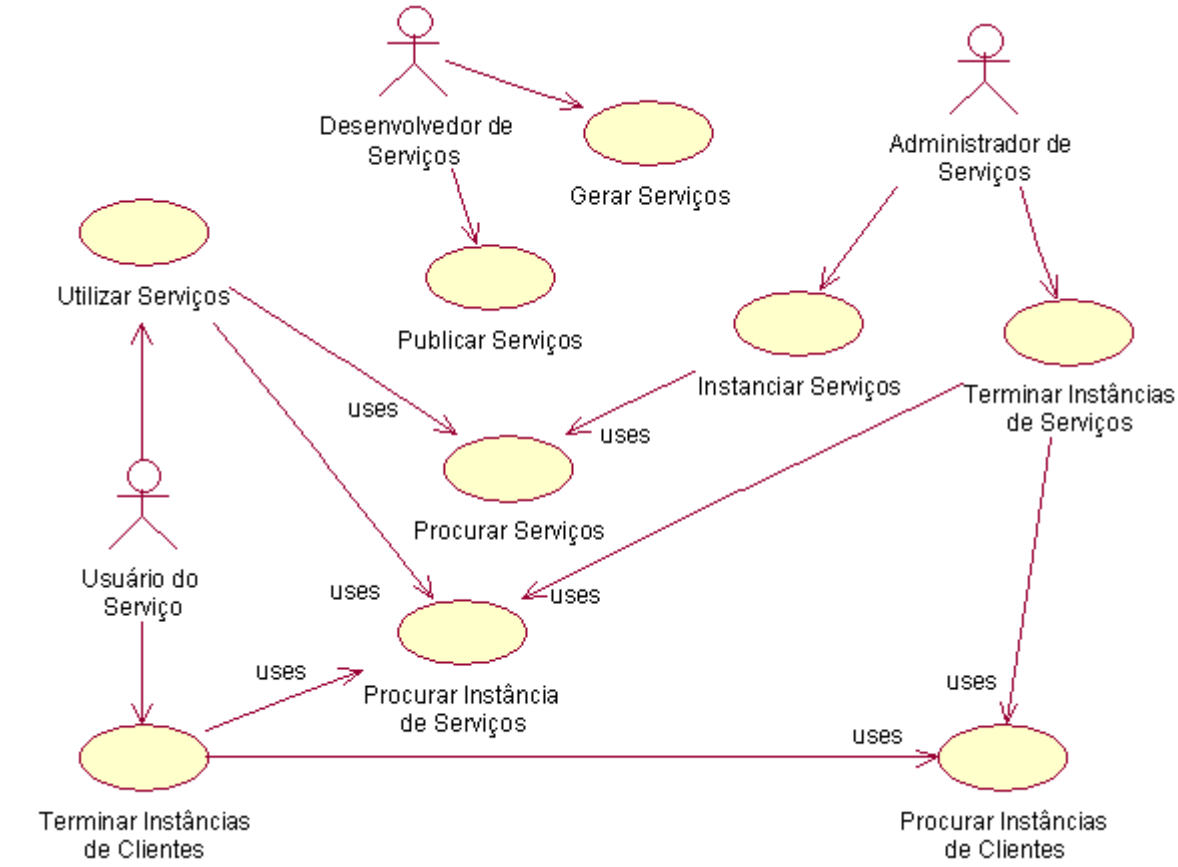

Figura 3: Diagrama de caso de uso UML como uma visão ampla do projeto.

Dessa forma, as interações do Administrador de Serviços refletem-se no Ambiente de modo a possibilitar a "utilização do serviço", tarefa realizada por Usuários e desenvolvedores de aplicações clientes.

O desenvolvedor de programas clientes constrói uma aplicação que fará uso das funcionalidades de um determinado serviço, apoiando-se no Ambiente.

A aplicação construída, fazendo uso de uma funcionalidade provida por um serviço, ou seja, o Cliente, deve desempenhar o papel para o qual foi proposto, sem que os seus usuários tenham o conhecimento das execuções de tarefas em pontos remotos da rede.

Contudo, um Usuário, por operar uma aplicação cliente, termina a utilização do serviço, quando desliga a máquina em que está trabalhando, fechando a aplicação, da mesma forma que os instancia, quando inicia o sistema.

As funcionalidades do Ambiente RGB Java serão explicadas com mais detalhes, por meio dos diagramas de seqüência e de classes, provenientes da Linguagem **UML**, nos próximos itens; porém, antes que se faça isso é necessário explicar alguns aspectos do comportamento das partes do Ambiente, com um baixo nível de detalhamento: do Registro de Serviços, do Provedor de serviços e do Cliente de Serviços.

O Registro de serviços tem a função de gerenciar as conexões dos Clientes com os serviços, armazenando os dados de localização (IP e porta do sistema operacional) e disponibilizando-os para o Ambiente.

Esses dados são gravados em tabelas do banco de dados **MySQL**[29], onde o Registro se conecta. Existem três tabelas: **Service**, para os dados de publicação dos serviços; **ServiceInstances**, para os dados de instância dos serviços e; **ClientInstances**, para os dados de instância do Cliente de serviços.

Os comandos **SQL**[30] e o *script* de criação do Banco de Dados ***Registry*** estão disponíveis no **Apêndice E**.

O Provedor de serviços, depois de instanciado, conecta-se com o Registro para que seus dados de localização sejam publicados. O Cliente utiliza o Registro para recuperar esses dados de localização e conecta-se ao serviço que foi instanciado pelo Provedor.

Quando um serviço é encerrado, o Ambiente do Provedor envia uma mensagem ao Registro para que se gerencie o Ambiente, fazendo com que os Clientes, que estavam conectados a essa instância de serviços, não parem de funcionar.

Caso exista uma outra instância disponível para o uso, o Registro executa uma troca de conexão dentro do Cliente, sem que o usuário tenha conhecimento. Caso não exista outra

[29] Sistema gerenciador de banco de dados distribuído gratuitamente para os fins de pesquisa.
[30] Linguagem utilizada como interface entre o sistema gerenciador de banco de dados e o usuário, possibilitando a interação com os dados.

instância do mesmo serviço, o Cliente aguarda até que uma outra se instancie. Quando isso ocorre, automaticamente o Registro faz com que o Cliente se conecte à nova instância do serviço.

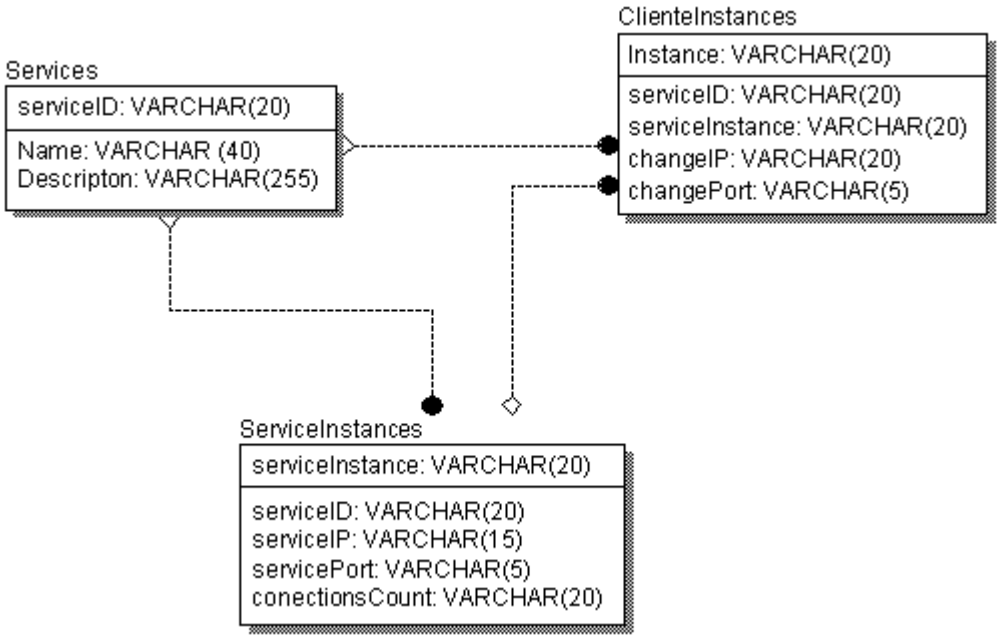

Figura 4: Diagrama de Entidade Relacionamento do Banco de Dados **Registry**.

Os serviços são identificados por um número, o "**serviceID**", publicado no Registro. Esse identificador representa unicamente um serviço para o Ambiente. Quando um serviço é instanciado, o Provedor deve informar esse identificador para o Registro.

Do outro lado, no Cliente, todas as requisições de serviços também devem informar esse identificador, para que uma instância seja localizada no Ambiente.

Dessa maneira, duas instâncias do mesmo serviço caracterizam-se assim, por terem o mesmo número de identificação e o de instância distintos.

Os números de instância são utilizados pelos serviços e pelos Clientes de serviços, quando esses estão disponíveis para a conexão, ou seja, quando estão sendo executados.

Quando existem duas instâncias do mesmo serviço, disponíveis para a conexão no Ambiente, e uma delas já está sendo utilizada por um Cliente, o Registro oferece a outro pedido de conexão Cliente, a instância que ainda não recebeu conexões.

Assim, o Ambiente “distribui”, de maneira igualitária, as conexões entre Cliente e serviços, para que algumas instâncias de serviços não se sobrecarreguem mais do que as outras. Em outras palavras, o Registro, gerenciando o Ambiente, balanceia a utilização dos serviços.

Quando um Cliente invoca a conexão de um serviço e não existem instâncias disponíveis para o uso, o Registro faz com que o Cliente aguarde até que uma instância seja carregada. Quando isso acontece, o Cliente que esperava é automaticamente conectado à nova instância do serviço.

O Ambiente é disponibilizado na forma de API Java (*Application Program Interface*) para que os desenvolvedores de serviços e Clientes de serviços possam criar suas aplicações, utilizando as funcionalidades da arquitetura. Essa abordagem no desenvolvimento de software auxilia na redução do tempo de desenvolvimento por abstrair as características de mais baixo nível que requerem uma conexão entre serviços distribuídos.

Neste capítulo, o Ambiente RGB Java será descrito, para que os aspectos referentes à implementação possam ser descobertos e explicados, subdividindo-se nas seis funcionalidades descritas pelo diagrama de casos de uso da **Figura 3**.

3.2. Descrição das funcionalidades do Ambiente RGB Java

As funcionalidades ilustradas na **Figura 3** são descritas na forma de diagramas de seqüência e de classes (**Apêndice F**) utilizando-se a Linguagem **UML**; são elas: gerar

serviços, publicar serviços, instanciar serviços, utilizar serviços, terminar instâncias de serviços e terminar instancias de clientes.

3.2.1. Gerar Serviços

A geração de serviços é feita pelo Gerador de serviços RGB Java, módulo acoplado ao Ambiente, para que se desenvolvam implementações dos serviços RGB Java, instanciados pelo Provedor de Serviços.

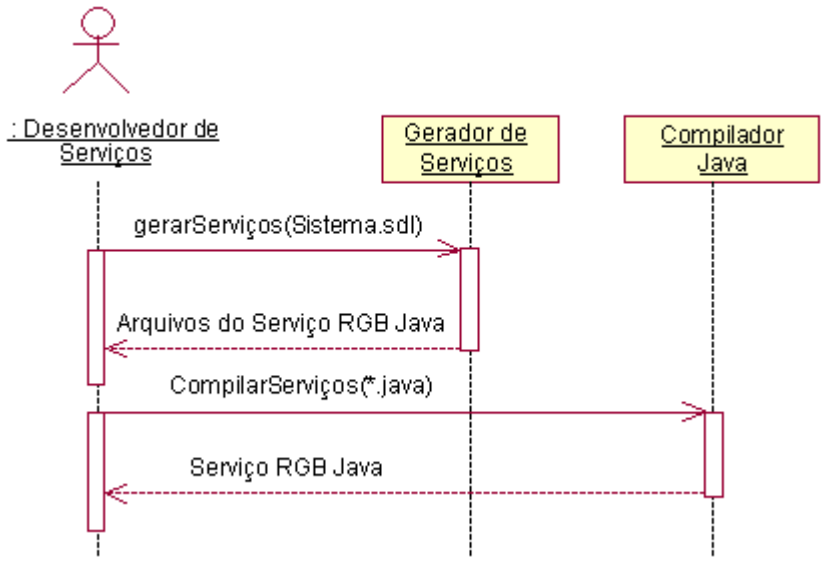

Figura 5: Diagrama de seqüência **UML** – Gerar Serviços

O Desenvolvedor de serviços, ator que interage com a geração dos serviços RGB Java, envia um arquivo contendo uma especificação de sistema em **SDL** para o Gerador de serviços, o **sdlParser**; processa a especificação e devolve os arquivos especificados na Linguagem Java, que deverão ser as entradas para o Compilador[31] Java (**Figura 5**).

O **sdlParser**, o Gerador de serviços, foi implementado em Java, utilizando-se a Ferramenta JavaCC; os analisadores léxico e sintático, construídos por Vilela (2002),

[31] Analisador e Gerador de Código baseado na gramática da Linguagem Java.

contribuíram para o desenvolvimento do Gerador, objetivado pela transformação de uma especificação de sistema SDL em serviços RGB Java.

O resultado da transformação é explicado no momento em que os diagramas **UML**, instanciar serviços, são descritos. Dessa maneira, o processo de instanciação de serviços ilustra a estrutura do Ambiente e a utilização das classes geradas.

Para que se compreenda a transformação de um sistema especificado em SDL em um Serviço RGB Java, o serviço KitchenTimer[32] foi gerado e seus arquivos disponibilizados no **Apêndice G**; O código Java do serviço KitchenTimer contém comentários que auxiliam a compreensão da transformação da especificação **SDL** em Java. Desta forma, o conteúdo do **Apêndice G** ajuda a explicar as equivalências entre uma especificação de sistema em SDL e um Serviço RGB Java.

3.2.2. Publicar Serviços

Para que um serviço RGB Java possa ser instanciado, primeiro ele deve ser publicado dentro do Registro de Serviços. A **Figura 6** ilustra este processo por meio do diagrama de seqüência UML.

O Desenvolvedor do serviço, por conhecer suas características funcionais, publica o serviço no Ambiente RGB Java, especificando o seu conteúdo de forma descritiva.

O processo de publicação de serviços não foi automatizado no ambiente RGB Java, pois não há um consenso entre os grupos de pesquisa sobre a maneira de descrever um serviço, ou seja, de como o serviço deve ser representado em um diretório global (**Seção 2.3**).

Porém, o Ambiente RGB Java utiliza alguns campos da tabela **Services** (**Figura 4**), para obter a existência da publicação dos serviços, identificando-os pelo do campo **serviceID**.

[32] Serviço que simula as funcionalidades de um relógio de cozinha.

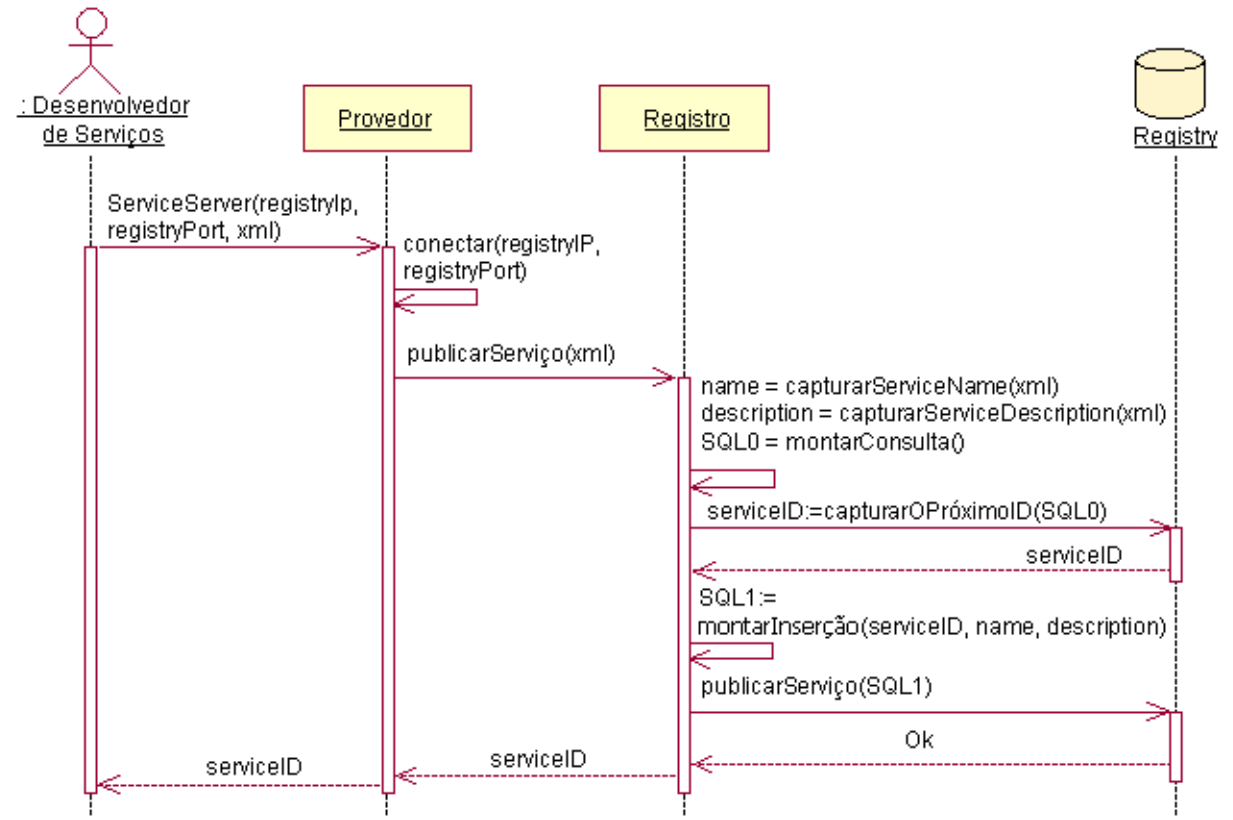

Figura 6: Diagrama de seqüência UML – Publicar Serviços.

Dessa forma, o Desenvolvedor de serviços, deveria estruturar uma mensagem **XML**, que contenha os campos da tabela **Services** (**Figura 4**), passando essas informações ao Provedor de serviços, juntamente com as informações de localização do Registro, o IP e a Porta do sistema operacional (**Figura 6**).

O Provedor de serviços conecta-se ao Registro para que seja enviada a mensagem especificada em **XML**. Ao receber essas informações, o Registro deve consultar a tabela **Services**, buscando o próximo identificador livre, que será associado ao serviço publicado.

O Registro deve incluir as informações enviadas pelo Provedor na tabela de serviços publicados, a **Services**, para que as outras funcionalidades do Ambiente RGB Java tenham a capacidade de consultá-la, na medida em que forem sendo executadas.

Ao final do processo de publicação de um serviço, o Registro deve informar ao Desenvolvedor, utilizando o Provedor, o identificador do serviço, o **serviceID**, que será utilizado em todas as outras etapas, nas interações dos atores com o Ambiente RGB Java.

Como a publicação do serviço consiste unicamente na inserção de um registro na tabela **Services**, esse processo pode ser realizado manualmente, utilizando a interface do **MySQL** e os comandos **SQL**, que estão disponíveis para consulta no **Apêndice E**.

3.2.3. Instanciar Serviços

A instanciação dos serviços no Ambiente RGB Java, depende das etapas de geração e de publicação e tem o objetivo de disponibilizar as informações de instância do serviço na rede, para que os Clientes possam se conectar.

O termo Administrador de serviços, utilizado pelo ator na **Figura 7**, é empregado, pois não é necessário que esse seja um desenvolvedor. Nessa etapa do projeto, os serviços são controlados pelo Ambiente e precisam somente ser encaminhados para tornarem-se instâncias de serviços.

O Administrador deve fornecer quatro informações de entrada ao Provedor para que se possa instanciar um serviço: Endereço IP do Registro, porta do Registro, identificador do serviço (**serviceID**), e a localização do serviço na máquina (pacote.classePrincipal).

As duas primeiras, o **IP** e a **porta**, servem para que o Provedor possa se conectar com o Registro; o identificador, o **serviceID**, é o número que assegura, nesta etapa, que o serviço foi devidamente publicado; e a localização da implementação do serviço (pacote.classePrincipal) indica o caminho que o provedor deverá ter para colocar o serviço em execução (na máquina).

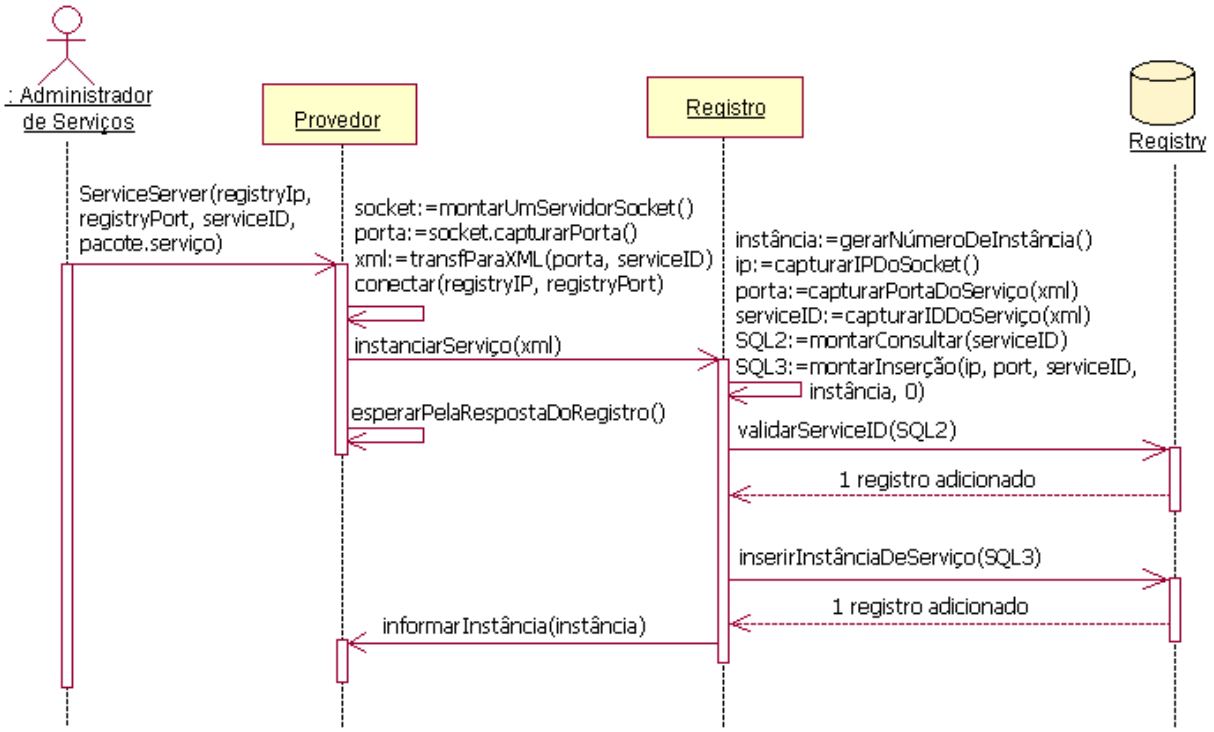

Figura 7: Diagrama de seqüência UML – Instanciar Serviços.

Quando o Provedor recebe essas informações, elas são extraídas e somadas a outras, que são geradas para que seja produzida uma mensagem ao Registro.

A primeira tarefa é a criação de um servidor **Socket TCP** para disponibilizar entradas aos Clientes de serviço. A porta do sistema operacional é selecionada de maneira seqüencial, ou seja, é escolhida com base em uma constante (1024). Caso esta já esteja sendo usada, o Provedor deverá tentar criar o servidor com a próxima porta, a de número 1025, e assim por diante.

Depois que a porta foi definida, essa será somada ao identificador do serviço, o **serviceID**, provido pelo ator Administrador em uma estrutura codificada em **XML** e enviada ao Registro pela da conexão montada entre as duas partes, que se baseou no **registryIP** e na **registryPort**.

O Registro, ao receber essa mensagem, gera um número de instância seqüencial, para poder representar a instância do serviço; captura o endereço IP através do ***socket*** e a porta do

serviço, que foi envida na mensagem codificada em **XML**, formando o endereço que deverá ser usado para conectar-se ao Provedor pelos Clientes de serviços; captura o **serviceID** da mensagem codifica em **XML;** e monta dois comandos **SQL**, um para verificar se existe o serviço publicado e outro para inserir as informações de instância do serviço, na tabela **ServiceInstances** (**Figura 4**), caso o **serviceID** seja encontrado na tabela de serviços publicados, a **Services**.

Ao final deste processo, o Provedor será informado com a instância do serviço. Caso o **serviceID** não represente algum serviço publicado no Ambiente RGB Java, o Registro deve notificar o Administrador, pelo do Provedor, que não foi possível encontrar serviços publicados com esse identificador.

O diagrama de seqüência, ilustrado pela **Figura 7**, representa uma parte do processo de instanciação de um serviço, a parte que preenche o Registro com as informações de localização do servidor **Socket TCP**, que é usado para permitir conexões com os Clientes.

A outra parte refere-se à instanciação do serviço na máquina, onde será preparado o Ambiente do Provedor, para que os Clientes possam se conectar. Os diagramas que descrevem a implementação dessa parte podem ser vistos nas **Figuras 8**, **9** e **10**.

O Ambiente do Provedor instancia ***threads***[33] para atender às requisições das conexões provenientes dos Clientes. As ***threads*** possibilitam ao Provedor independência na utilização dos serviços, fazendo com que esses usos sejam executados concorrentemente para os Clientes.

Cada Cliente que requisita uma conexão com o Provedor é responsável pela instanciação de uma ***thread***, sendo então esse processo, executado após o pedido de conexão do Cliente.

[33] Linhas de execução; utilizadas para promover a computação concorrente.

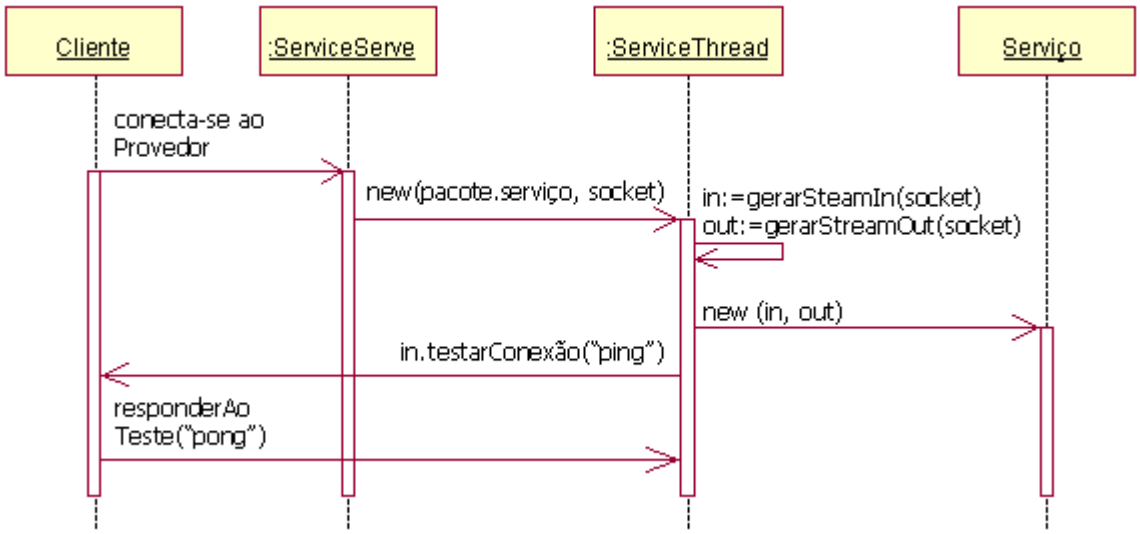

Figura 8: Diagrama de seqüência UML – Instanciar *thread* de Serviço.

Quando um Cliente requisita uma utilização de serviço, o Provedor, representado pela classe **ServiceServer**, instancia uma classe **ServiceThread**, passando por parâmetro a localização da classe gerada pelo Gerador de serviços e o objeto **Socket**, o qual contém os canais de entrada e saída de uma conexão.

A classe **ServiceThread**, ao receber esses objetos, separa as classes de entrada e saída em duas: ***in*** e ***out***.

Dessa forma, os objetos ***in*** e ***out*** são passados por parâmetro para o serviço, que no diagrama da **Figura 8** é representado pela entidade "Serviço", que no diagrama da **Figura 9** é subdividido em classes.

A classe **ServiceThread**, instancia o serviço utilizando o endereço que localiza as classes produzidas pelo Gerador de Serviços e compiladas pelo Java.

O Serviço no diagrama da **Figura 8** é uma representação genérica de todas as classes ilustradas na **Figura 9;** a classe Serviço na **Figura 9** representa a classe principal do serviço, que é produzido pelo Gerador de Serviços.

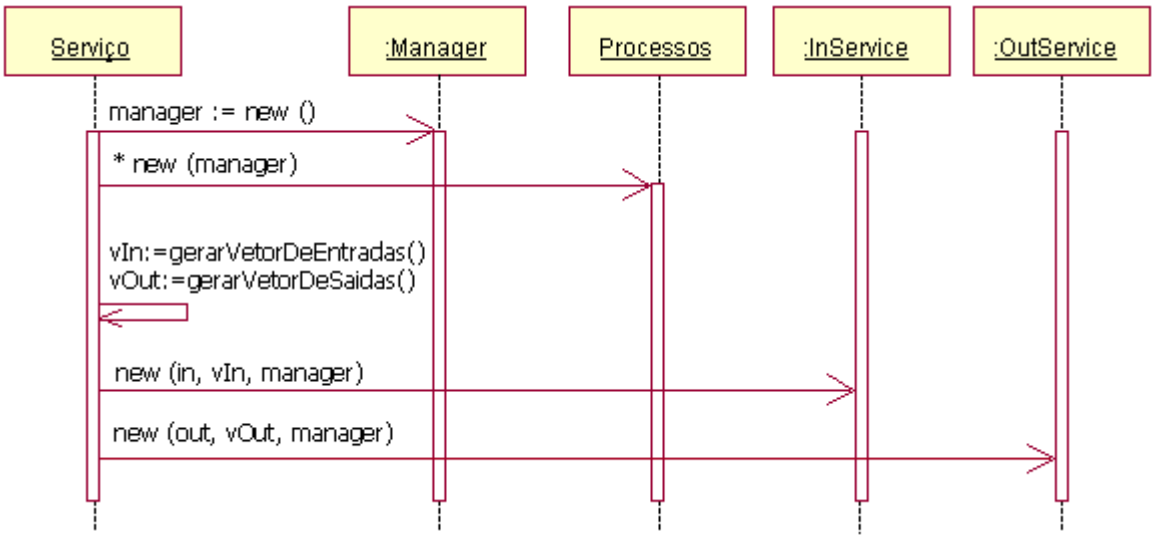

Figura 9: Diagrama de seqüência UML – Instanciação das classes produzidas pelo Gerador de serviços.

O diagrama da **Figura 9** o Provedor de serviços instancia as classes que são produzidas pelo Gerador de serviços, estruturando assim o Ambiente onde as funcionalidades deverão ser executadas.

O Ambiente é composto por, no mínimo, cinco classes Java: a classe principal, a **Manager**, a **InService**, a **OutService** e pelo menos um processo RGB Java (gerados a partir dos processos **SDL**).

O termo “Ambiente” é utilizado para expressar a interação entre as classes citadas acima neste contexto, sendo o Ambiente, a comunicação entre os processos RGB Java e o Ambiente RGB Java.

A classe principal leva o nome do sistema especificado em **SDL** e dá origem a todas as outras classes do Ambiente, distribuindo corretamente canais de entrada e saída para as classes **InService** e **OutService**, respectivamente.

A classe **Manager** funciona como um centralizador de mensagens, onde os processos podem tanto produzir como consumir essas mensagens, com o intuito de realizarem saídas e entradas de dados e sinais.

O objeto **Manager** é criada a partir da classe principal, sendo passada por parâmetro para todas as outras classes do Ambiente: Os processos e as classes de entrada (**InService**) e saída (**OutService**).

Os processos, resultados da transformação dos processos especificados em um sistema SDL, comportam-se de maneira a simular as ações que dão origem às funcionalidades do serviço.

Representados pela entidade “Processos”, no diagrama da **Figura 9**, eles são criados a parir da classe principal e recebem como parâmetro, na sua construção, uma referência ao objeto **Manager**. Dessa forma, os processos estão preparados para trocar mensagens entre si e com os Clientes.

A classes de entrada (**InService**) e saída (**OutService**), segundo o diagrama da **Figura 9**, recebem três parâmetros na sua construção: os canais de entrada (**in**) ou saída (**out**); vetores que descrevem as mensagens; e uma referência do objeto **Manager**, para que esses possam trocar mensagens com os processos e com os clientes.

Os vetores que descrevem as entradas e saídas do Ambiente são estruturas de dados, criadas dentro da classe principal, geradas pelo Gerador de Serviços segundo a descrição dos canais de entrada e saída do sistema especificado em **SDL**.

Para o objeto **InService** é passado o vetor **vIn**, e para o **OutService**, o **vOut**. O **vIn** auxiliam a decodificação de mensagens provenientes do Cliente, para serem convertidas em produções dentro do objeto **Manager**. O **vOut** corresponde ao vetor de decodificação do que é consumido dentro do objeto **Manager**, para serem transformadas em mensagens ao Cliente.

Dessa forma, uma ***thread*** representa a conexão com um Cliente; o Ambiente embutido no Provedor executa as funcionalidades do serviço, utilizando um gerenciador de mensagens (**Manager**). Esse Ambiente é gerado a partir da ***thread***, funcionando

concorrentemente com outras ***threads***, que devem ser instanciadas para atender as requisições de outros Clientes.

No diagrama da **Figura 10** é possível visualizar a comunicação entre os processos e o Ambiente RGB Java por meio das classes de entrada e saída, pela simulação de uma parte do sistema **KitchenTimer**, especificado em **SDL** no **Apêndice A**.

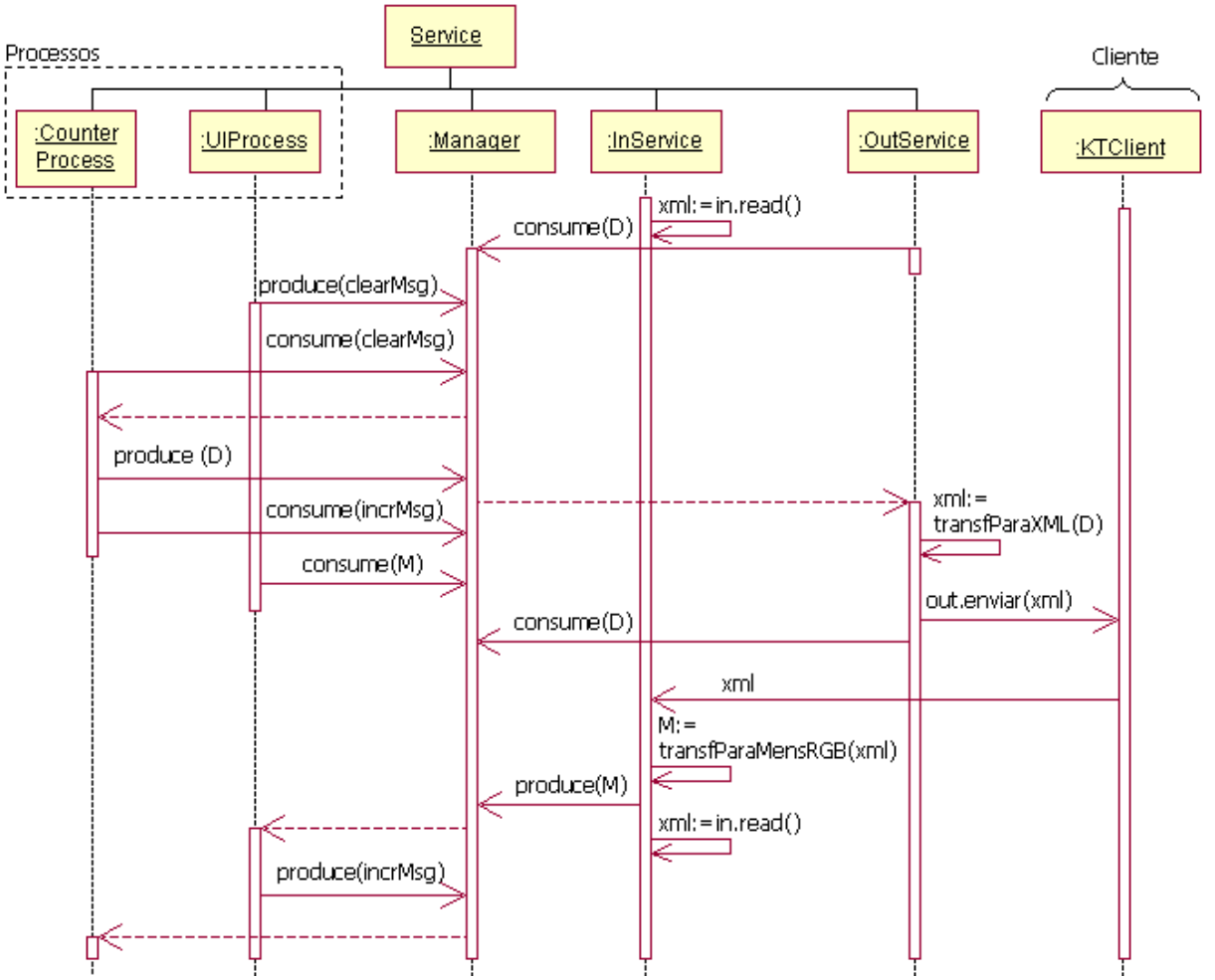

Figura 10: Diagrama de seqüência UML – Simulando o funcionamento de um serviço.

No diagrama ilustrado pela **Figura 9**, foi possível compreender a interação das classes que contribuem para a execução das funcionalidades de um serviço; porém, é mais simples examinar a relação entre elas por um exemplo real de especificação de serviço.

Na **Figura 10**, o diagrama de seqüência propõe a simulação da execução de uma pequena parte do sistema especificado em **SDL**, o **KitchenTimer** (disponível para consulta

no **Apêndice A**), com o intuito de mostrar as trocas de mensagens entre os processos e com um dado programa cliente, **KTClient** (**Apêndice B**).

O serviço, como dito anteriormente, é formado pelos processos, canais de entrada e saída e um gerenciador de mensagens. No diagrama da **Figura 10**, os processos são **CounterProcess** e **UIProcess**; os canais de entrada e saída, as classes **InService** e **OutService**, respectivamente; e o gerenciador de mensagens, a classe **Manager**.

A classe principal, a **Serviço**, ilustrada no diagrama da **Figura 9**, não será abordada na simulação pois ela não interage com as outras classes que provêm as funcionalidades do serviço, tendo somente a tarefa de dar origem a essas.

A simulação inicia-se com a classe **InService** esperando por mensagens do cliente **KTClient** utilizando o canal de entrada **in** (*xml:=in.read()*). Desta forma, o Ambiente que executa as funcionalidades, abre uma porta de entrada para as mensagens provenientes do Ambiente RGB Java.

O objetivo da classe **InService** é “ler” as mensagens da rede, pelo objeto **in** e produzir mensagens RGB Java no **Manager**; o da classe **OutService** é consumir as mensagens do gerenciador (**Manager**), transformar essas em código **XML** e enviar para o Cliente, utilizando o objeto **out**.

A segunda mensagem da simulação (**Figura 10**) é exatamente o que foi descrito acima: o consumo de um sinal “**D**”, proveniente de algum processo, intermediado pelo gerenciador de mensagens.

O objeto **vOut**, o vetor de saída, recebido na instanciação da classe **OutService**, descreve os sinais que devem ser lidos no gerenciador para serem enviados ao Cliente, assim como o objeto **vIn**, agregado pela instância de **InService**, contêm os sinais que podem ser recebidos de um Cliente, para serem produzidos dentro do gerenciador.

Essas duas classes, a **InService** e a **OutService**, aguardam os eventos requeridos, até que eles ocorram, executando suas funções para que as entradas e saídas nesse Ambiente, produzam o efeito de comunicabilidade com o meio externo.

O processo **UIProcess** é o primeiro a produzir uma mensagem no gerenciador. Mesmo que não exista uma requisição, essa mensagem deve ficar armazenada, possibilitando que outros processos ou canal de saída consumam-na futuramente.

A próxima mensagem, enviada para o gerenciador, tem o intuito de consumir o sinal que acaba de ser produzido pelo processo **UIProcess**. A classe **CounterProcess** recebe então, o sinal **clearMsg** e o gerenciador o retira de seus registros pendentes.

Os processos são instâncias de classes **Java**, concorrendo entre si para a utilização do processador; os processos devem ter fatias de tempo com prioridades controladas pelo sistema operacional.

O processo **CounterProcess**, após consumir o sinal **clearMsg**, teve “tempo” de executar outras de suas funcionalidades, como produzir um sinal **D**, antes que outro processo fosse escalonado para utilizar o processador.

Esse diagrama (**Figura 10**) não deverá descrever as trocas dos processos em execução, caracterizadas pela concorrência, pois tem a função de ilustrar a seqüência dos acontecimentos no Ambiente dos serviços; desta forma, o escalonamento apresenta-se como uma característica nativa do sistema operacional.

A classe **OutService**, que havia requerido um sinal **D**, nesse momento pôde consumi-lo, pois o processo **CounterProcess** o produziu dentro do gerenciador de mensagens. Após o consumo, o sinal **D** será transformado em **XML** e enviado ao Cliente.

Concorrentemente ao envio da mensagem especificada em **XML** para o Cliente, os processos **CounterProcess** e **UIProcess** tentam consumir os respectivos sinais, **incrMsg** e **M** no gerenciador; porém, esses ainda não foram produzidos.

Quando uma instância de **InService** receber uma mensagem do Cliente, essa deve ser transformada em sinal e produzida no gerenciador (sinal **M**).

O processo **UIProcess** consome o sinal **M** e produz o sinal **incrMsg** no gerenciador para que o processo **CounterProcess** possa consumi-lo.

As classes que realizam as entradas e saídas no Ambiente voltam ao seu estado de "consumir", tanto da rede (**InService**) quanto do gerenciador (**OutService**) após terem finalizado as suas execuções.

Desta forma, pôde-se simular parte da execução de um serviço, cobrindo as principais possibilidades de ocorrência, que são executadas no Ambiente.

3.2.4. Utilizar Serviços

Nos próximos seis diagramas será descrito o processo de utilização de serviços, baseando-se em dois diferentes níveis de detalhamento. Os dois primeiros enfocarão a utilização de serviços de forma a abstrair alguns detalhes do processo; os outros quatro apresentarão as interações com as classes da API Blue, que é utilizada pelos programas clientes.

O primeiro diagrama (**Figura 11**), que ilustra a utilização dos serviços, com baixo nível de detalhamento, foi desenvolvido para demonstrar os efeitos de uma requisição Cliente, de utilização de serviços, dentro do Registro.

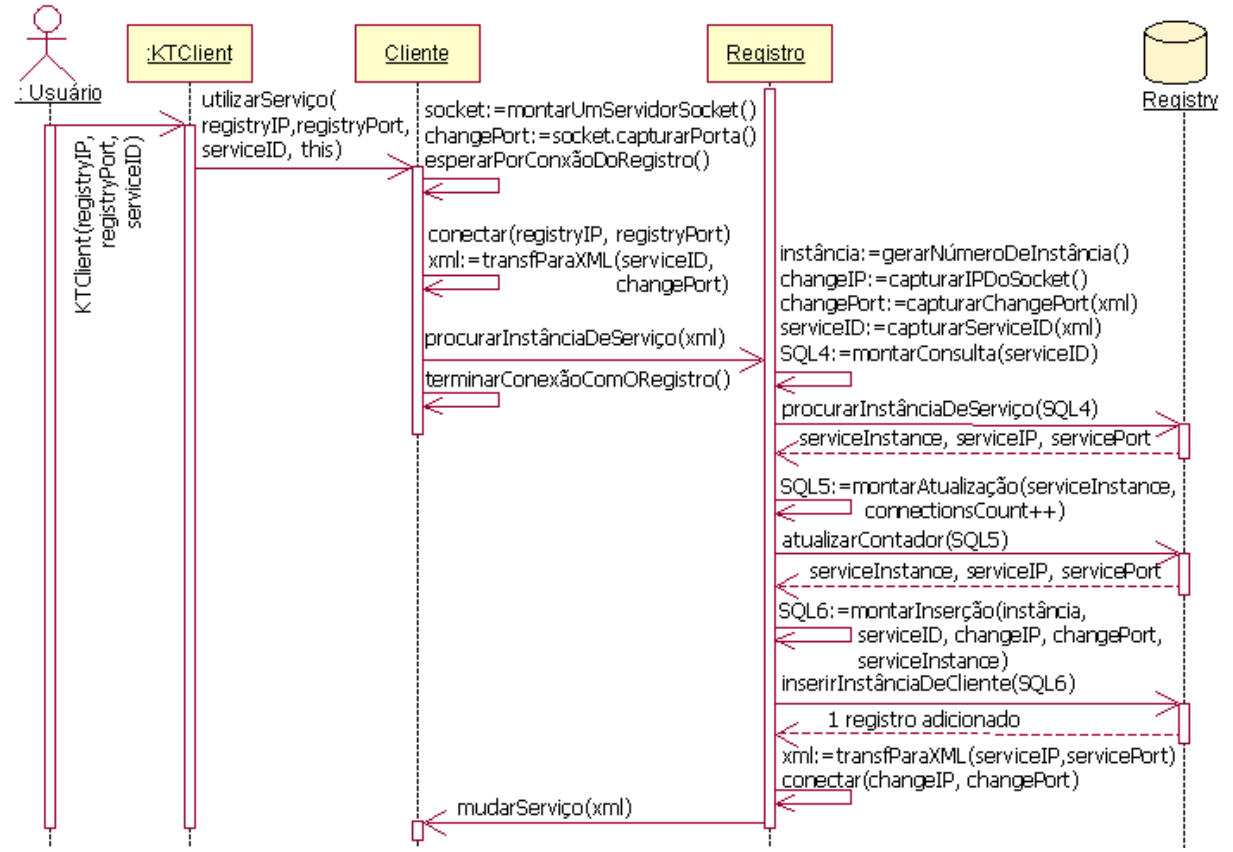

Figura 11: Diagrama de seqüência UML – Utilizar Serviços - A interação do Cliente com o Registro.

O diagrama da **Figura 12**, demonstra a interação do Cliente com o Provedor, após ter recebido as informações de instância de um serviço, provenientes do Registro, ilustrada no diagrama da **Figura 11**.

O Usuário inicia o programa **KTClient** com as informações de localização do Registro, IP e porta, sendo que esse deve estar conectado à **API Blue** (API Cliente). O Cliente, a **API**, recebe como parâmetro, na sua construção, o endereço IP do Registro (**registryIP**), a porta (**registryPort**), o identificador do serviço (**serviceID**) e uma referência ao programa **KTClient** (***this***).

A referência do programa KTClient pode ser explicada nos outros quatro diagramas que ilustrarão a utilização dos serviços de forma mais detalhada.

Antes que o Cliente conecte-se ao Registro, esse monta um servidor **Socket TCP**, para que o Registro tenha a capacidade de comunicar-se com o Cliente em tempos esporádicos, dependente de eventos de término de instância de serviços, por exemplo.

O processo de instanciação do servidor **Socket** gera um número de porta que deverá se chamar **changePort** (porta de mudança). Por meio desta porta e do endereço IP do Cliente, o Registro poderá fazer mudanças de serviços em tempos esporádicos.

Enquanto o Cliente espera por essa conexão, ele conecta-se com o Registro, monta uma mensagem estruturada em **XML**, contendo o identificador do serviço e a porta de mudança (**changePort**).

Após enviar a mensagem, o Cliente termina a conexão com o Registro e aguarda a resposta, o endereço IP e porta da instância de serviço, através da porta de mudança.

O Registro, ao receber o pedido de localização de uma instância de serviços, baseado no seu identificador, gera um número de instância para a requisição do Cliente, captura o endereço IP através do objeto **Socket** (conexão), a porta contida na mensagem codificada em **XML** e o identificador do serviço.

Com o identificador, o **serviceID**, o Registro monta uma consulta **SQL** (**Apêndice E - SQL4**) com o intuito de procurar instâncias do serviço, ordenada pelo campo **connectionsCount** (**Figura 4**), que disponibilizará como primeira resposta, a instância que menos conexões estiver atendendo.

A consulta é realizada com base na tabela de instâncias de serviços, a **serviceInstances** (**Figura 4**) retornando o número de instância do serviço (**serviceInstance**), o endereço IP da instância (**serviceIP**) e a porta do sistema operacional (**servicePort**). Após ter encontrado uma instância de serviços, o Registro atualiza o campo **connectionsCount**, somando-o em uma unidade.

Antes que o Cliente receba as informações de localização do serviço (IP e porta), o Registro insere na tabela de instâncias de Clientes (**clientInstances**) os cinco campos descritos no diagrama da **Figura 12**, para que o Cliente possa transformar-se em uma instância dentro do Registro. Os campos são: número de instância do Cliente, **serviceID**, **changeIP**, **changePort** e **serviceInstance**.

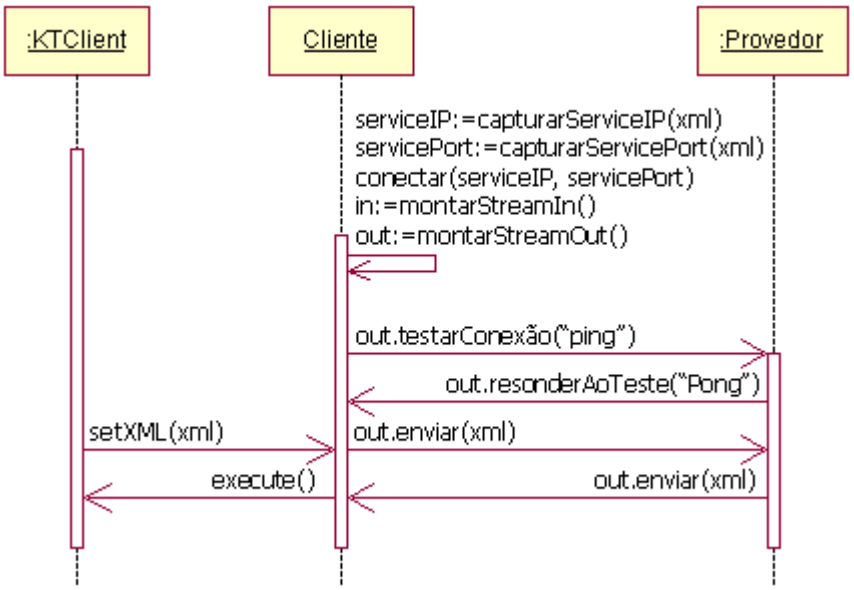

Figura 12: Diagrama de seqüência UML – Utilizar Serviços - A interação do Cliente com o Provedor de serviços.

Ao final do processo de utilização de um serviço, o Registro conecta-se ao Cliente através do seu endereço IP (**changeIP**) e da porta de mudança (**changePort**) e, envia o endereço de localização da instância de serviços (**serviceIP** e **servicePort**) codificado em **XML**, para que ele possa fazer uso de suas funcionalidades.

O diagrama ilustrado na **Figura 12**, demonstra a capacidade do Cliente de conectar-se ao serviço, após ter recebido as informações, provenientes do Registro, que apontam a sua localização, o **serviceIP** e o **servicePort**.

Quando o Cliente recebe a mensagem do Registro, através da porta de mudança, a localização da instância de serviço, ele as extrai e conecta-se a instância. Neste momento os canais de entrada e saída são divididos em dois objetos, **in** e **out**.

A comunicação entre Cliente e Serviço é feita de maneira assíncrona, ou seja, não é necessário que exista uma mensagem de entrada para ser produzida uma de saída; ou pode-se ter inúmeras mensagens para um Serviço quando somente uma é produzida como resposta ao Cliente, os canais de entrada e saída deverão ser distribuídos em classes distintas, para que exista uma independência na ordem em que as mensagens alcançam o Cliente e aquelas que são encaminhadas ao Serviço.

Esse tipo de estrutura do Ambiente RGB Java (com relação à utilização dos serviços), as classes e os canais de entrada e saída, serão descritos com maior detalhamento nos próximos quatro diagramas.

Após ter se conectado a instância de serviço, o Cliente envia uma mensagem ao Provedor, requisitando um “teste” na conexão. Essa abordagem possibilita resolver o problema da sincronização, ou seja, força o programa **KTClient** enviar as mensagens ao serviço, somente após esse estar pronto para recebe-las.

Quando o “teste” é respondido, o programa **KTClient**, pode estruturar as mensagens e enviá-las ao serviço. Como as saídas de dados no Cliente são desassociadas com as entradas, a resposta da instância de serviços deve ser executada, dentro do Cliente, pela invocação de um método (o ***execute()***).

Por esse motivo, o Ambiente Cliente deve ter uma referência ao programa Cliente, o **KTClient**, para que se possa invocar o método **execute()**. Dessa forma, o **KTClient** deve ser especificado para que as possíveis respostas da instância do serviço, encontrem uma correspondência dentro do método **execute()**, realizando as tarefas necessárias.

Essas características também serão abordadas com maiores detalhes nos próximos quatro diagramas subseqüentes e nos estudos de casos, encontrados no **Capitulo 4**.

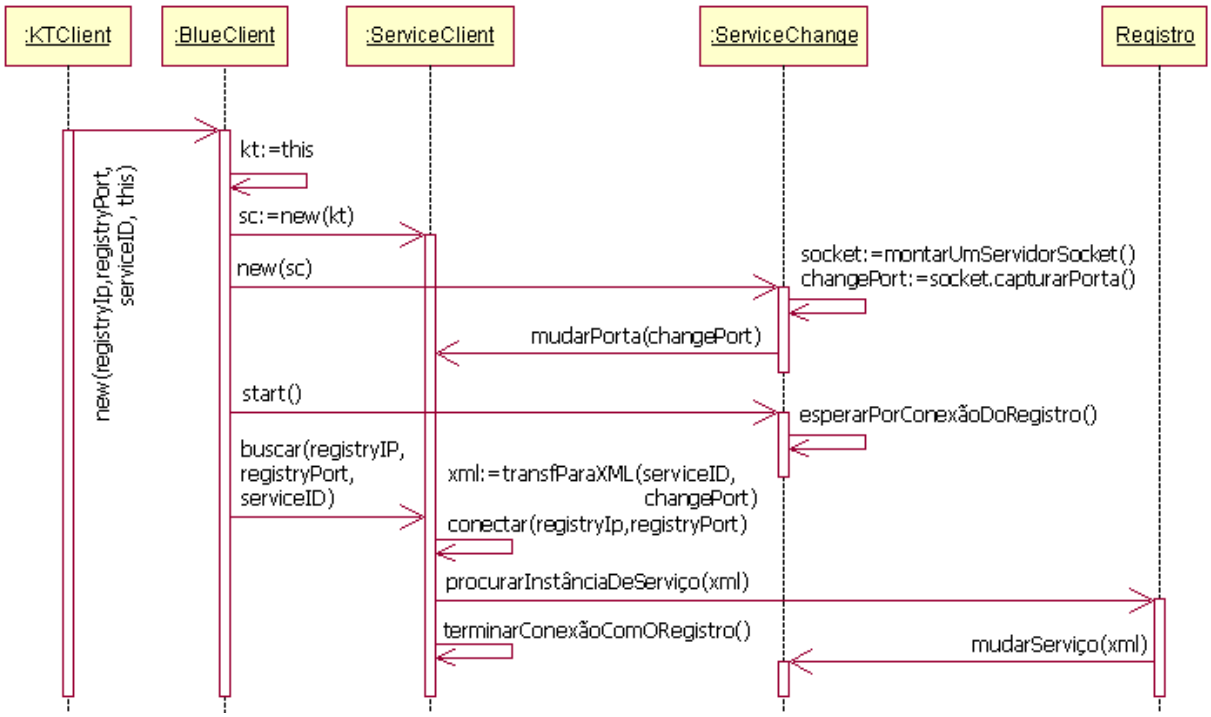

Figura 13: Diagrama de seqüência UML – Utilizar Serviços - A interação do Cliente com o Registro de serviços – diagrama com nível de detalhamento elevado.

Fazendo uma analogia com os diagramas anteriores, o **da Figura 13**, é equivalente ao da **Figura 11**, porém, o mais recente, é enriquecido com detalhes referentes a estruturação das classes dentro do Cliente. Por outro lado, esse diagrama não ilustra o tratamento dos dados, provenientes do Cliente, no Registro de serviços, abstraindo essa etapa, por já ter sido amplamente abordada em sua equivalência, na **Figura 11**.

Para que o Cliente se torne mais coeso e tenha a capacidade de executar funções concorrentes, nesse diagrama, ele foi subdivido em três classes: a **BlueClient**, a **ServiceClient** e a **ServiceChange**.

Através da classe **BlueClient**, o programa **KTClient** se conecta, para enviar requisições de utilização de serviços e encaminhar as mensagens que originam as execuções das funcionalidades na instância do serviço.

A classe **ServiceClient**, é utilizada pelo Ambiente Cliente, para conectar-se ao serviço. Nela estão contida os canais de entrada e saída, sendo que, para o canal de entrada, ela cria outra classe. A instanciação dessa outra classe, somente poderá ser vista no próximo diagrama, quando for possível ilustrar a interação do Cliente com a instância de serviço.

Além de conectar-se com as instâncias de serviços, a classe **ServiceClient** tem a capacidade de fazer as requisições ao Registro.

Como o Cliente deve aguardar por conexões do Registro, é necessário que exista uma classe para atender esses pedidos de conexão, sendo executada, concorrentemente às outras classes que compõe o Cliente. Essa classe é a **ServiceChange**.

Seguindo agora o diagrama da **Figura 13**, a classe BlueClient utiliza uma variável, a **kt**, para armazenar uma referência ao programa **KTClient**. Futuramente o método **execute()** deverá ser acessado, utilizando essa referência.

A classe **ServiceClient** é instanciada pela **BlueClient** recebendo a referência ao programa **KTClient**, através da variável **kt**. A **ServiceChange**, no momento de sua criação, recebe uma referência à instância **ServiceClient**.

Como a classe **ServiceChange**, deve informar ao Cliente, sobre mudanças de instâncias de serviços, provenientes do Registro, essa deve ter como acessar os métodos da **ServiceClient**, para que se execute a troca.

Após ter sido instanciada, a classe **ServiceChange**, monta um servidor **Socket TCP**, e envia a porta de mudança (**changePort**) à instância da classe **ServiceClient** e espera por conexões originadas pelo Registro.

A **ServiceClient**, reúne as informações necessárias (**serviceID** e **changePort**) para um pedido de utilização de serviço em uma mensagem **XML**, conecta-se com o Registro, envia essa mensagem e termina a conexão.

O Registro, processa esse pedido, retornando ao Cliente, através de uma conexão com o servidor **Socket** (na classe **ServiceChange**), o endereço IP (**serviceIP**) e a porta (**servicePort**) da instância de serviços, agregados em uma mensagem codificada em **XML**.

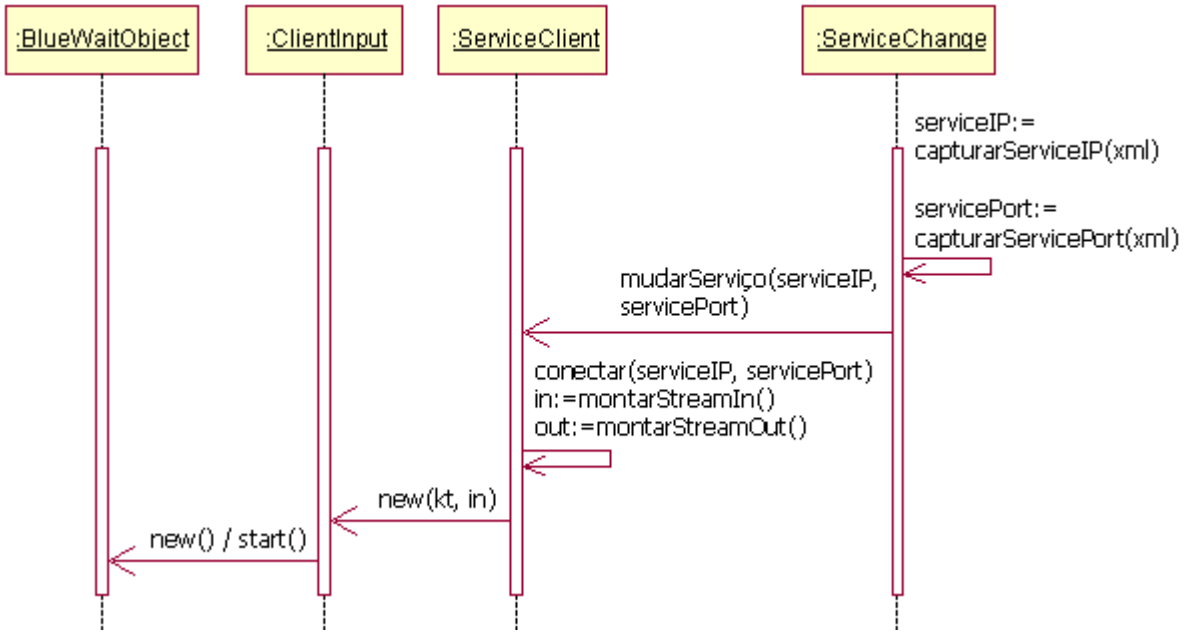

Figura 14: Diagrama de seqüência UML – Utilizar Serviços - Conectando-se ao Provedor de serviços – diagrama com nível de detalhamento elevado.

Assim como o diagrama da **Figura 13** é equivalente ao da **Figura 11**; os diagramas das **Figuras 14**, **15** e **16**, equivalem ao da **Figura 12**, utilizando um nível de detalhamento mais elevado.

Após a **ServiceChange** ter recebido a mensagem codificada em **XML**, proveniente do Registro, contendo o endereço IP (**serviceIP**) e a porta (**servicePort**) da instância do serviço, essas informações são extraídas e depositadas nas suas respectivas variáveis.

A **ServiceClient**, utilizando essas informações, conecta-se à instância do serviço, obtendo os canais de entrada e saída (**in** e **out**).

O canal de entrada (**in**) é passado para a instância da classe **ClientImput**, que garantirá independência no recebimento das mensagens assíncronas, provenientes da instância de serviços.

Juntamente com o canal **in**, a classe **ServiceClient**, enviará a referência ao programa **KTClient**, transformando a instância da classe **ClientImput** em uma entidade com capacidade de receber mensagens provenientes da instância do serviço, e executá-las após terem sido decodificadas.

A classe **BlueWaitObject**, funciona como um paralisador do consumo de entrada na **ClientImput**, fazendo essa deixar de "ler" as entradas, provenientes da instância do serviço, até que alguma outra mensagem termine de executar o método **execute()**.

O canal de saída, o **out**, ficará disponível para utilização, na classe **ServiceClient**. Dessa forma, quando o programa **KTClient**, enviar alguma requisição à instância do serviço, essa deverá encontrar o canal **out**, dentro da classe **ServiceClient**.

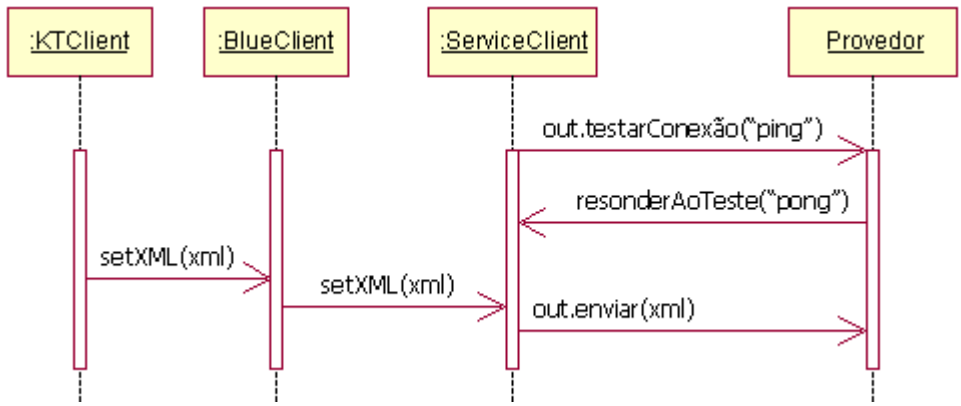

Figura 15: Diagrama de seqüência UML – Utilizar Serviços - A interação do Cliente com o Provedor de serviços – diagrama com nível de detalhamento elevado.

Como podem ser visualizados no diagrama da **Figura 15**, os "testes" são realizados entre Cliente e Provedor, para que exista a garantia de que o programa **KTClient**, somente enviará mensagens a instância do serviço, após a conexão, entre eles, encontrar-se devidamente estabelecida.

Uma mensagem originada no **KTClient**, deve acessar a classe **BlueClient**, para que ela, encontre o canal de saída (**out**), disponível na classe **ServiceClient**.

O próximo diagrama (**Figura 16**) ilustrará a execução de funcionalidades contida no Cliente, utilizando a invocação do método **execute()**, após esse ter recebido mensagens da instância do serviço.

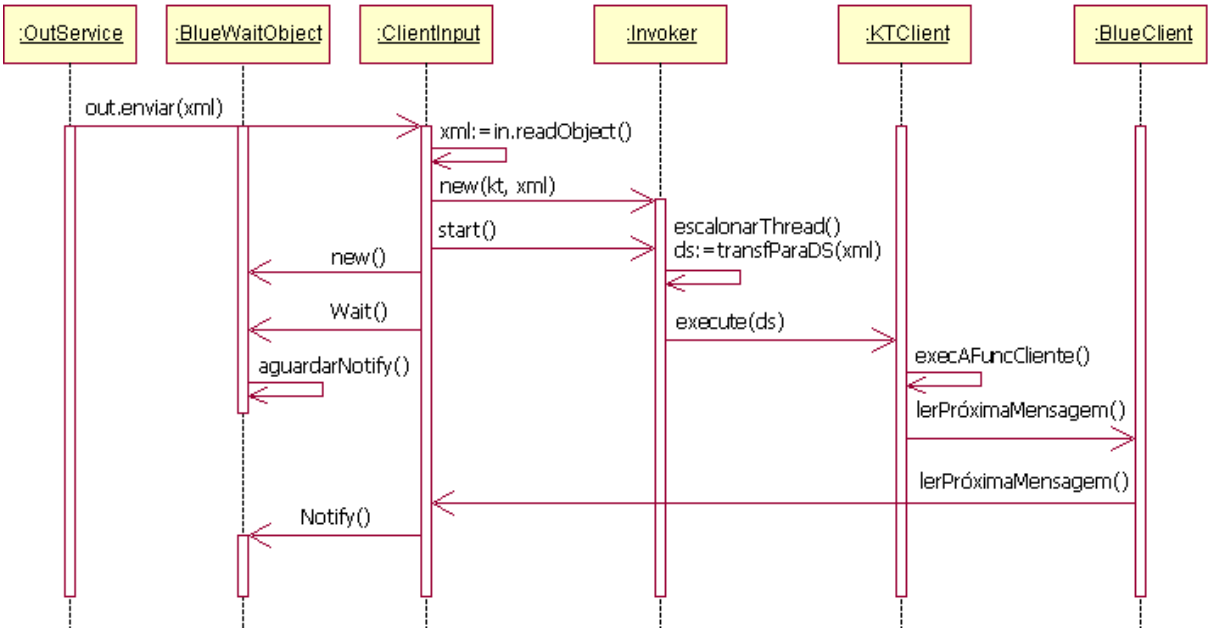

Figura 16: Diagrama de seqüência UML – Utilizar Serviços - Invocação do método **execute()** para a ocorrência de uma entrada no Cliente – diagrama com nível de detalhamento elevado.

Como pôde ser visto na simulação da execução do sistema **KitchenTimer**, especificado em **SDL**, a classe **OutService**, pertencente ao Ambiente Provedor, funciona enviando mensagens para o Cliente.

No diagrama acima (**Figura 16**), essa mesma classe é utilizada para fornecer mensagens ao Ambiente Cliente, tendo como destino, a classe **ClientImput**, onde está localizado o canal de entrada (**in**).

Dessa forma, uma mensagem proveniente de uma instância de serviço, deve ser encaminhada pela classe **OutService** e recuperada pela **ClientImput**, que estará formatada na Linguagem **XML**.

No momento em que uma mensagem é "lida" (**in.read()**), a classe **Invoker** é instanciada, recebendo, como parâmetro, o conteúdo da mensagem (**xml**) e a referência do programa **KTClient** (**kt**).

Logo em seguida, a instância da classe **Invoker** (uma ***Thread***), é colocada em execução (**start()**) e escalonada na memória principal do computador, para que concorra com os outros processos, pela utilização do processador. Assim será enviada para última posição, na fila de espera (***yield***[34]).

Esse mecanismo, possibilita que a próxima execução seja a instanciação da classe **BlueWaitObject**, fazendo com que a instância da classe **Invoker**, espere para executar, até que a **ClientImput** entre em "estado de espera" (**Wait()**).

Quando a instância da classe **Invoker** volta a utilizar o processador, ela decodifica a mensagem recebida do serviço, disponibilizando-a em uma estrutura de dados (**DataStructure**) e; invoca o método **execute()**, utilizando a referência ao programa **KTClient** (**kt**), passando a estrutura de dados (**ds**) como parâmetro.

A partir dessa estrutura, é possível descobrir o conteúdo da mensagem, proveniente da instancia do serviço, utilizando o método **get()**, representado no diagrama (**Figura 16**) pela execução do método **execAFuncCliente()**.

Após a execução, possível pela implementação do método **execute()**, deverá haver uma chamada de método à classe **BlueClient** (**lerPróximaMensagem()**), para que outras mensagens, aguardando serem "lidas" pela classe **ClientImput**, possam participar do processo de execução das funcionalidades Cliente.

[34] Do inglês, conceder; refere-se a troca de processos em execução, onde o processo que executa o método *yield*(), está dando vez aos outro processos da fila de espera.

No diagrama da **Figura 16**, não é mostrada a classe **ServiceClient**, porém, é ela que entrega à classe **ClientImput**, a tarefa de executar o método **Noify()**, contido na **BlueWaitObject**, e não a classe **BlueClient**.

A execução do método **Notify()**, configura o estado da **ClientImput** para “pronto para ler”, possibilitando que outras mensagens acessem o Ambiente Cliente.

3.2.5. Terminar Instâncias de Serviços

A finalização de uma instância de serviço pode ser dada pelo término de sua execução do seu Provedor. No momento em que o Provedor é finalizado, esse envia uma mensagem ao Registro, notificando o término da sua execução.

Para que essa mensagem seja enviada, antes que o processo deixe de existir na memória principal do computado, algumas tarefas são realizadas. Não é considerado término da execução de um serviço, a falta de energia elétrica, pois dessa maneira, as tarefas que realizam a notificação ao Registro ficariam impossibilitadas de serem executadas.

As tarefas que precedem a finalização de uma instância de serviço são realizadas para que uma mensagem codificada em **XML**, contenha o número de instância do serviço e para que uma conexão seja estabelecida com o Registro (**Figura 17**).

O Registro, ao receber a notificação, captura da mensagem codificada em **XML**, o número de instância do serviço, que está sendo terminado.

Utilizando a tabela de instâncias de serviços, a **ServiceInstances** (**Figura 4**), o Registro recupera o identificador do serviço (**serviceID**). Esse número, posteriormente servirá para procurar outras instâncias do mesmo serviço. Os Clientes, que utilizavam a instância que foi terminada, podem receber, pela porta de mudança (**changePort**), as informações de localização da nova instância.

Com relação aos acessos do Registro ao Banco de Dados **Registry** (**Figura 4**): o registro da tabela de instância de serviços, é apagado (**SQL8**); outra instância do mesmo serviço é procurada (**SQL9**); são buscadas as instâncias de Clientes que utilizavam aquela instância de serviço, que acabou de terminar (**SQL10**); o número de instância de serviços, contido na tabela de instancias de Clientes, é substituído pelo novo e; os Clientes são notificados com o endereço de localização da nova instância de serviço.

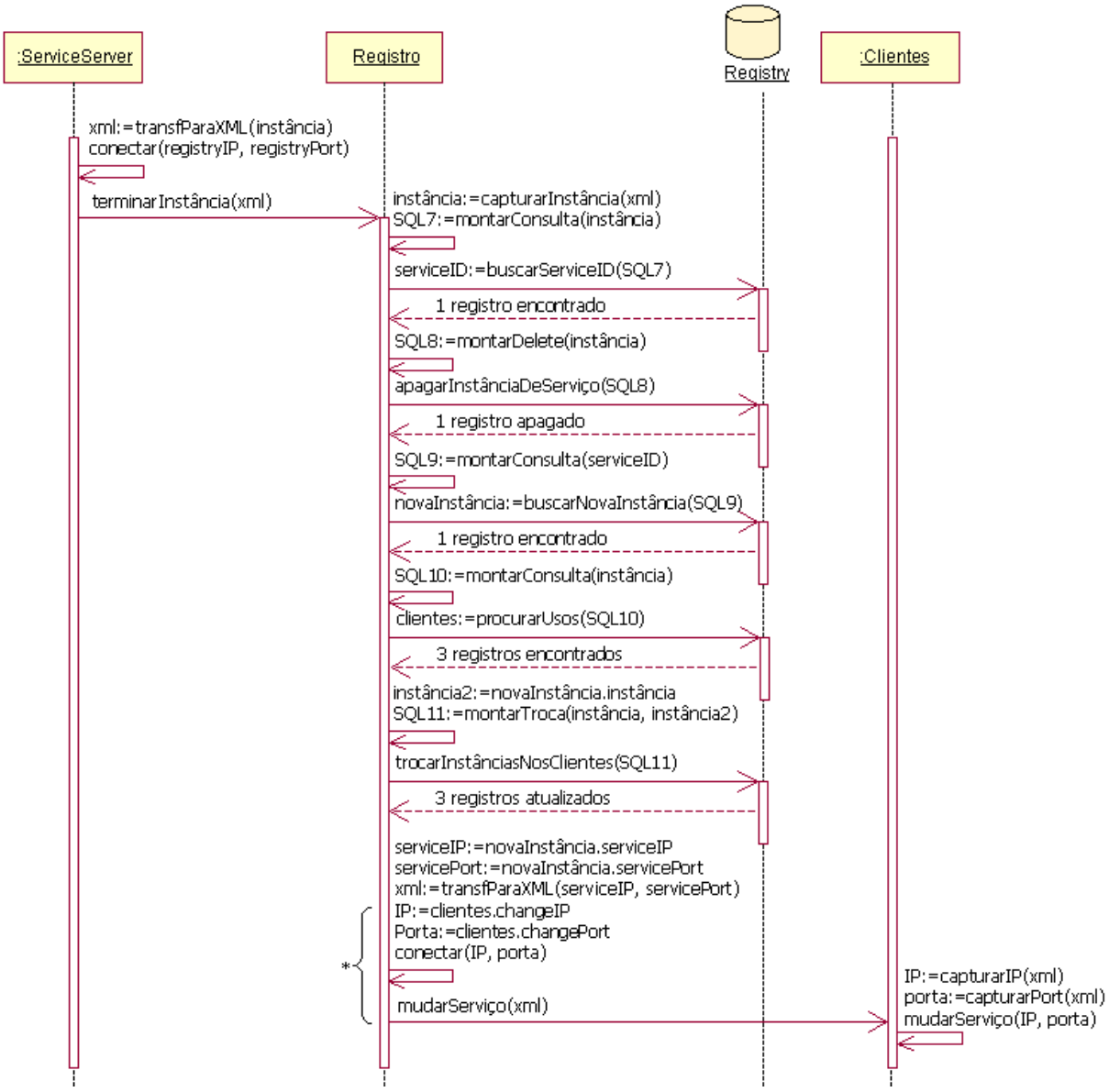

Figura 17: Diagrama de seqüência UML – Terminar Instâncias de Serviços.

Caso não existam instâncias do mesmo serviço, disponíveis para que os Clientes as utilizem, esses deverão ser notificados de que o serviço encontra-se “fora do ar”. Quando uma nova instância daquele serviço é disponibilizada, o Registro encarrega-se de notificar os Clientes com os endereços de localização, possibilitando a conectividade de Clientes e instância de serviço.

No diagrama ilustrado pela **Figura 17**, asterisco (*****) simboliza a repetição das tarefas compreendidas na chave. Essa repetição faz com que os Clientes sejam notificados, com o endereço de localização da nova instância de serviço.

3.2.6. Terminar Instancias de Clientes.

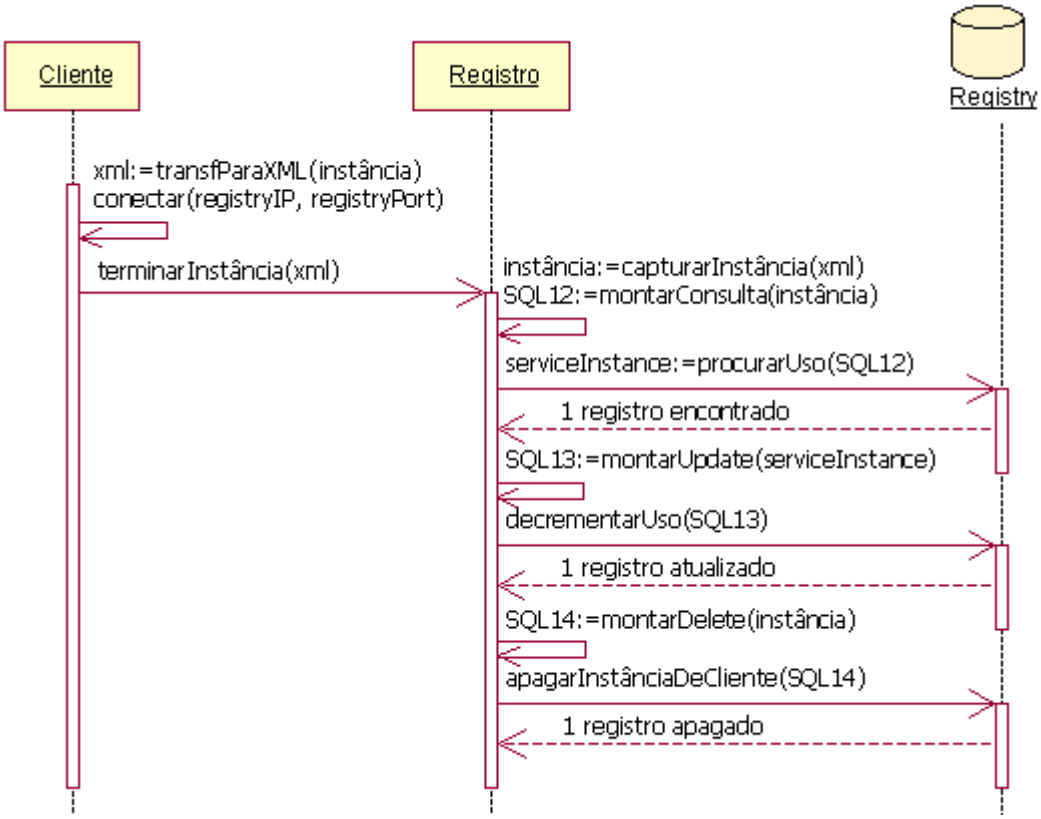

Figura 18: Diagrama de seqüência UML – Terminara Instâncias de Clientes.

Terminar a execução de um Cliente, é a tarefa mais simples que o Ambiente RGB Java executa. Ela tem a função de diminuir, em uma unidade, do contador de utilização da

instância de serviços, encontrado na tabela de instâncias de serviços e apagar o registro da tabela **ClientInstances**, que representa o Cliente desistente.

O Cliente, antes de terminar a sua execução, conectar-se ao Registro (**Figura 18**) para notificar o seu número de instância. Esse captura da mensagem codificada em **XML**, o número de instância e acessa o Banco de Dados do Registro, para atualizar os campos necessários.

Seguindo a ordem dos acessos ao Banco de Dados Registry (**Figura 4**), o Registro procura pela instância do Cliente na tabela **ClientInstances** (**SQL12**); atualiza o contador de utilização de serviços (**SQL13**) e; apaga o registro da tabela de instâncias de Clientes, que corresponde ao Cliente que terminou sua execução (**SQL14**).

3.3. Considerações Finais

Neste capítulo foi possível visualizar os aspectos da implementação do Ambiente RGB Java através da demonstração de suas características funcionais, utilizando-se dos diagramas de classes (**Apêndice F**) e dos de seqüência da Linguagem **UML**, de modo a dar uma explicação mais formal ao desenvolvimento do Ambiente.

A implementação do Gerador de Serviços foi baseado nos analisadores léxico e sintático desenvolvidos por Vilela (2002). Dessa forma, a demonstração dos aspectos de implementação do Gerador de Serviços é feito somente no que se refere a geração de código Java, estando disponível na forma de diagramas de classes UML no **Apêndice F**.

No **Apêndice G** é possível visualizar o sistema KitchenTimer (VILELA, 2002), especificado em SDL, na forma de Serviço RGB Java; Os comentários disponíveis na especificação do serviços KitchenTimer é utilizado para que se possa compreender a equivalência entre um Serviço RGB Java e a especificação deste em SDL.

4. ESTUDOS DE CASO

A seqüência dos itens neste capítulo, obedece à ordem de como um serviço RGB Java deve ser tratado para que se utilizem suas funcionalidades.

4.1. Configurando o Ambiente RGB Java

Para o uso da API RGB Java faz-se necessária à inclusão dos arquivos **red.jar**, **green.jar**, **blue.jar** e **sdlParser.jar** na variável de ambiente **classpath**[35], o que permite aos usuários do Ambiente acesso ao Registro, ao Provedor, ao Cliente e ao Gerador de serviços.

Figura 19: Configurando os arquivos que compõe a API RGB Java na variável de ambiente **classpath**.

Se os arquivos que compõem o Ambiente RGB Java estiverem localizados no diretório **RGB** dentro na unidade "**e**", então a variável de ambiente **classpath** deverá conter os seguintes endereços, (a **Figura 19** apresenta a Interface de configuração do **classpath**):

"**e:\RGB\red.jar; e:\RGB\green.jar; e:\RGB\blue.jar; e:\RGB\sdlParser.jar;.**"

Dentro dos arquivos com extensão "**.jar**", encontram-se classes agrupadas em pacotes. Estas classes podem ser instanciadas pelos usuários do Ambiente RGB Java ou por outras classes.

[35] Variável de ambiente utilizada pelo Java para apontar (contendo os endereços) os pacotes a serem importados por outras classes.

Um usuário instancia uma classe do Ambiente quando esta é o Registro, o Provedor ou o Gerador de serviços. Uma classe do Ambiente pode ser instanciada por outra classe quando ela é o meio para se ter acesso ao Cliente de serviços.

A classe Registry, contida no pacote **com.rgb.red**, por exemplo, é executada pelo usuário do Ambiente para que o Registro de serviços entre no estado de execução. A classe **BlueClient**, contida no pacote **com.rgb.blue**, é instanciada por outra classe quando esta é um programa que terá acesso a um determinado serviço, usando a **BlueClient** como meio.

Portanto, o Ambiente RGB Java é formado por um conjunto de classes que se interligam no momento em que estão instanciadas, comunicando-se umas com as outras, com a finalidade de reduzir os esforços do desenvolvimento de sistemas, implementando as funcionalidades predefinidas para um Ambiente de serviços.

É importante lembrar que, apesar da **Figura 19** ter sido retirada de um sistema baseado em software Microsoft[36], o Ambiente, composto pela API RGB Java, pode ser utilizado em qualquer sistema operacional que tenha, para este, uma **JVM**[37] (*Java Virtual Machine*), devido à característica de portabilidade do Java.

Essa característica de portabilidade está de acordo com o conceito da comunicação dos Ambientes de serviços, quando possibilita que o problema a ser resolvido, seja em nível das mensagens trocadas entre serviços e Clientes de serviços e não, em nível da plataforma em que os programas são construídos para serem executados.

[36] Empresa norte americana de desenvolvimento de software.
[37] Máquina virtual Java; utilizada pelo Java para ler os *bytecodes* e transforma-los em linguagem de máquina.

4.2. Estudo de Caso 1: KitchenTimer

O **KitchenTimer**, que pode ser encontrado no **Apêndice A**, é um sistema especificado na Linguagem **SDL**, que descreve as funcionalidades de um relógio de cozinha.

Um relógio de cozinha, através da interação com um usuário, deve ter a capacidade de acumular segundos, sendo posteriormente usado como o tempo total de uma contagem regressiva.

Como esta tarefa é realizada por meio de uma contagem regressiva, a variável que o contador do relógio manipula, deve tender ao valor zero. No momento em que o valor zero é atingido pelo seu contador de tempo, o relógio de cozinha entra no estado de “alarme”, emitindo um som de alerta, o qual tem a função de notificar o término da contagem.

Esta sistemática pode ser comparada à de um relógio despertador, cuja função é emitir um sinal de alarme, representado por um estado "despertar", quando o tempo corrente atinge a marcação predefinida pelo usuário.

A diferença entre estes dois sistemas é que o relógio de cozinha, ao contrário de um despertador, conta os segundos de maneira regressiva até que se atinja o valor zero.

Para melhor compreender os sistemas especificados na Linguagem **SDL**, estes são subdivididos em blocos compostos por processos inter-relacionados (**Seção** 2.4).

Os canais de entrada e saída representam a interface de comunicação do sistema com o Ambiente, assim como podem ser vistos na **Figura 20**.

Este estudo de caso é baseado na utilização do sistema **KitchenTimer** como um serviço RGB Java. Desta forma, as saídas produzidas pelo relógio deverão ter como destino um programa cliente conectado ao Ambiente RGB Java e as entradas, as saídas deste mesmo cliente de serviço.

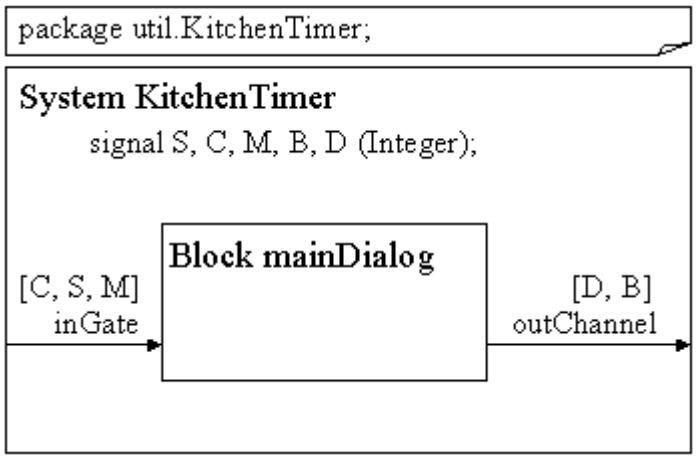

Figura 20: Representação gráfica KitchenTimer no nível de sistema.

Os sinais de entrada do canal **inGate**, têm a função de limpeza (*Clear*), início da contagem (*Start*) e incremento do contador (*More*) respectivamente ilustrados na **Figura 20**. Os de saída, que usam o canal **outChannel** como meio de acesso ao Ambiente, são os de apresentação (*Display*), representado pela letra **D** e, alarme (*Beep*), com a letra **B**.

Os processos, por sua vez, são os responsáveis pela execução das tarefas computacionais, ou seja, as instruções que definem as funcionalidades de um sistema, são especificadas no nível dos processos.

A interação entre eles, no sistema KitchenTimer, podem ser visualizadas na **Figura 21**, através das rotas de sinais e seus respectivos sinais.

O sistema **KitchenTimer**, descreve três processos (**Figura 21**), subdivididos de acordo com as funcionalidades específicas, referentes ao seu funcionamento. São eles: a interface com o usuário (**UIProcess**), o contador (**CounterProcess**) e o disparador de alarme (**RingProcess**).

O primeiro, a interface com o usuário, **UIProcess** (*User Interface Process*), graficamente representado pela **Figura 22**, tem como finalidade receber os sinais de incremento (**M**) , limpeza (**C**) e início da contagem (**S**).

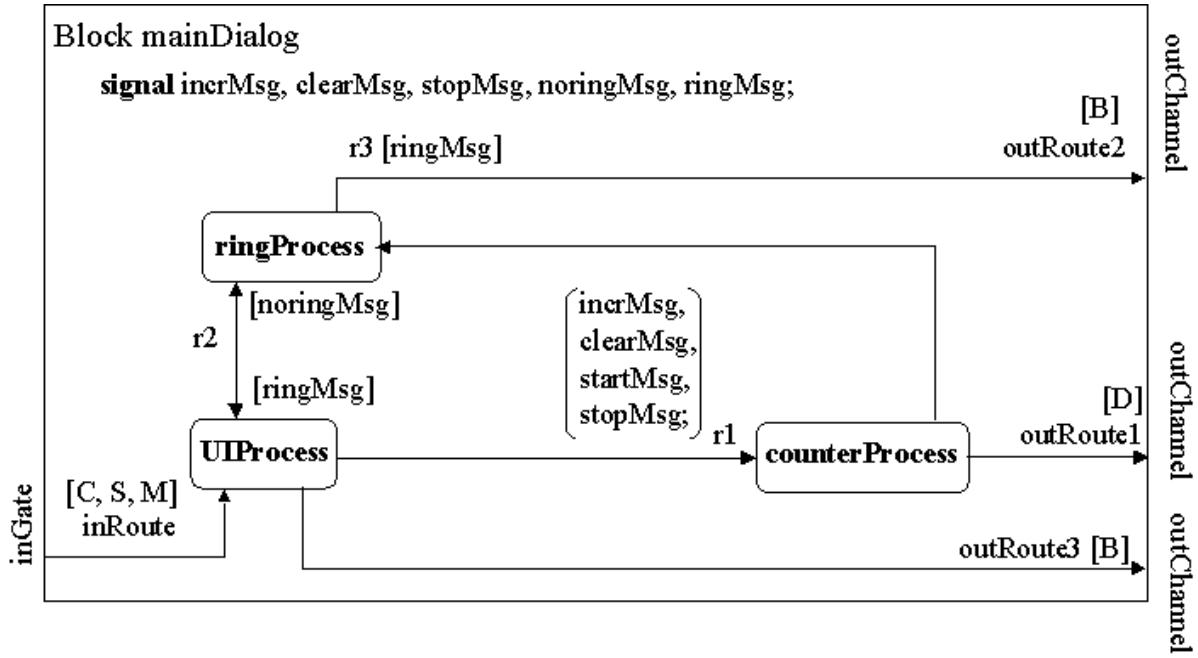

Figura 21: Representação gráfica da interação entre os processos no KitchenTimer.

A parada da contagem é representada pela entrada do sinal **S**, no momento em que o relógio estiver executando o estado *counting* (ver **Figura 23**). Por outro lado, a parada do alarme é definida por qualquer outra entrada no processo **UIProcess**.

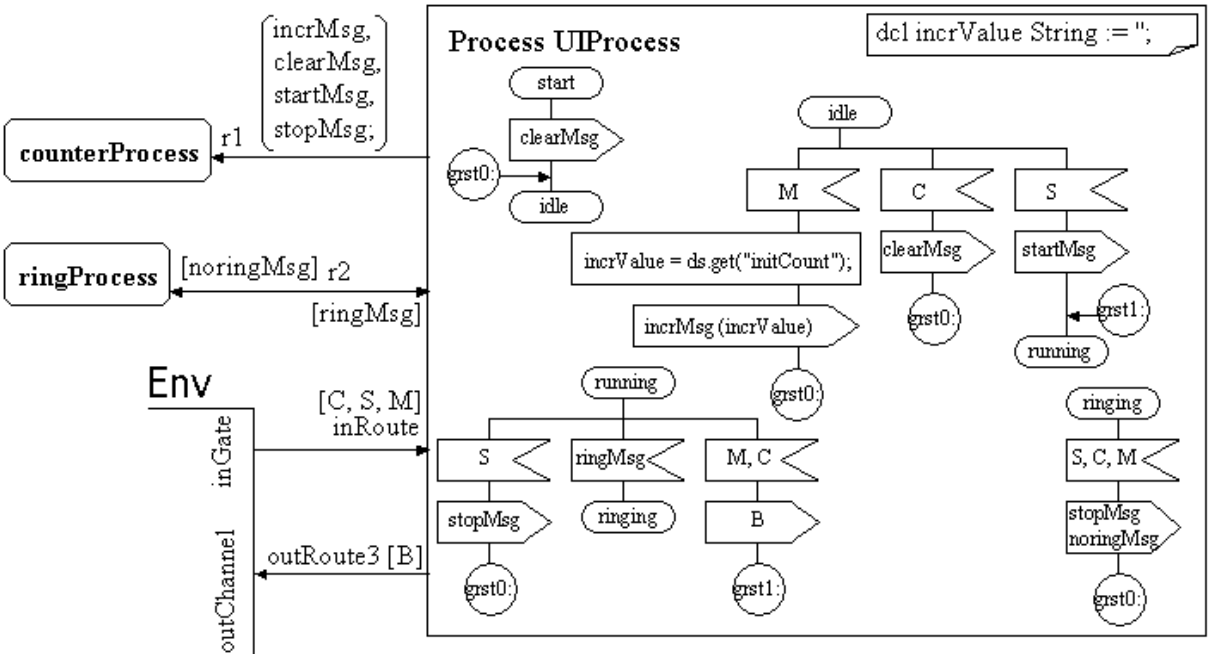

Figura 22: Representação gráfica de processo UIProcess.

Quando um sinal **M** estiver disponível na rota de sinal **inRoute**, conectado ao canal **inChannel**, e o estado corrente for o *idle*, a entrada é consumida e um sinal de incremento (**incrMsg**) é produzido na rota **r1** para que o **CounterProcess** possa somar uma unidade a uma variável contadora.

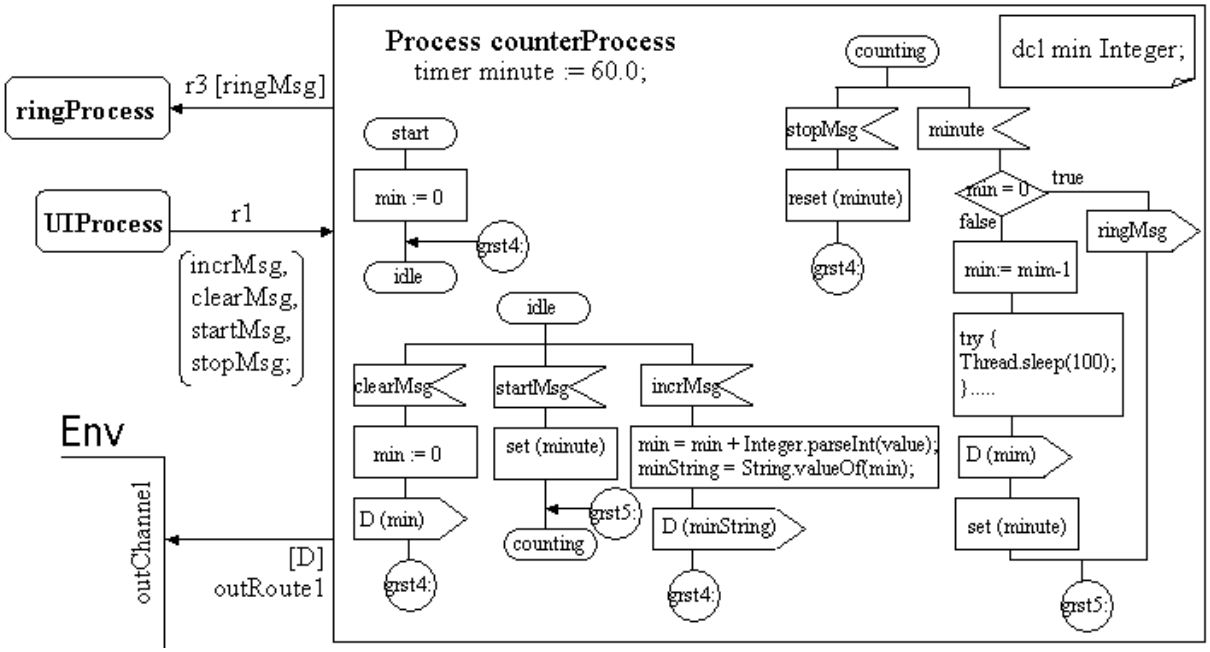

Figura 23: Representação gráfica de processo contador (CounterProcess).

Interligado ao **UIProcess**, como é possível ver por meio da rota de sinal **r1**, representada na **Figura 23**, o processo contador (**CounterProcess**) tem como função armazenar os segundos produzidos pelos sinais de entrada e, contá-los.

Para cada sinal de incremento emitido pelo usuário, o **UIProcess** envia um sinal ao **CounterProcess** e este incrementa o valor de uma variável descrita como **min** na especificação do processo contador.

O valor desta variável representa o tempo total que o relógio deverá contar. Quando um sinal de início de contagem (**S**) for detectado no **UIProcess**, este produz o sinal de **startMsg** para o **CounterProcess**, o que dá início ao mecanismo de contagem.

Este mecanismo faz com que o valor da variável **min** seja comparado a zero para cada iteração descrita no estado *counting*. Caso essa premissa seja falsa, ou seja, caso a variável **min** não contenha o valor zero, esta é decrescida em uma unidade.

A seguir, o processo deverá ser interrompido pelo tempo de um segundo. Esta pausa faz com que o relógio emita sinais ao usuário, a um programa cliente, contendo o valor da variável **min**, de forma temporizada.

A instrução que representa a pausa está descrita na tarefa (*task*) com o conteúdo

```
try {
     Thread.sleep(1000);
}catch(InterruptedException exception){}
```

e está especificada na Linguagem Java.

A emissão do valor da variável **mim** para um programa cliente é descrita na instrução de saída (**output**) "**D(min)**", onde **D** (*Display*) é o sinal que representa a saída de **mim** passando como parâmetro, o seu valor.

Quando o resultado da comparação for verdadeiro, ou seja, quando a variável **min** estiver armazenando o valor zero, o **CounterProcess** enviará um sinal ao processo disparador de alarme (**RingProcess**) e este emitirá um sinal sonoro.

O relógio de cozinha, como um serviço, deverá emitir este sinal sonoro de forma silenciosa, ou seja, não com uma instrução que resulte som, e sim com uma representação deste som na forma de dado.

O sinal de saída **B**, contido no estado *ringing* do processo **RingProcess** (**Figura 24**), é a representação do som de alarme em forma de sinal. Quando um programa cliente, baseado no serviço **KitchenTimer**, recebe este sinal, significa que a instância de serviço relógio, entrou no estado disparando, ou seja, *ringing*.

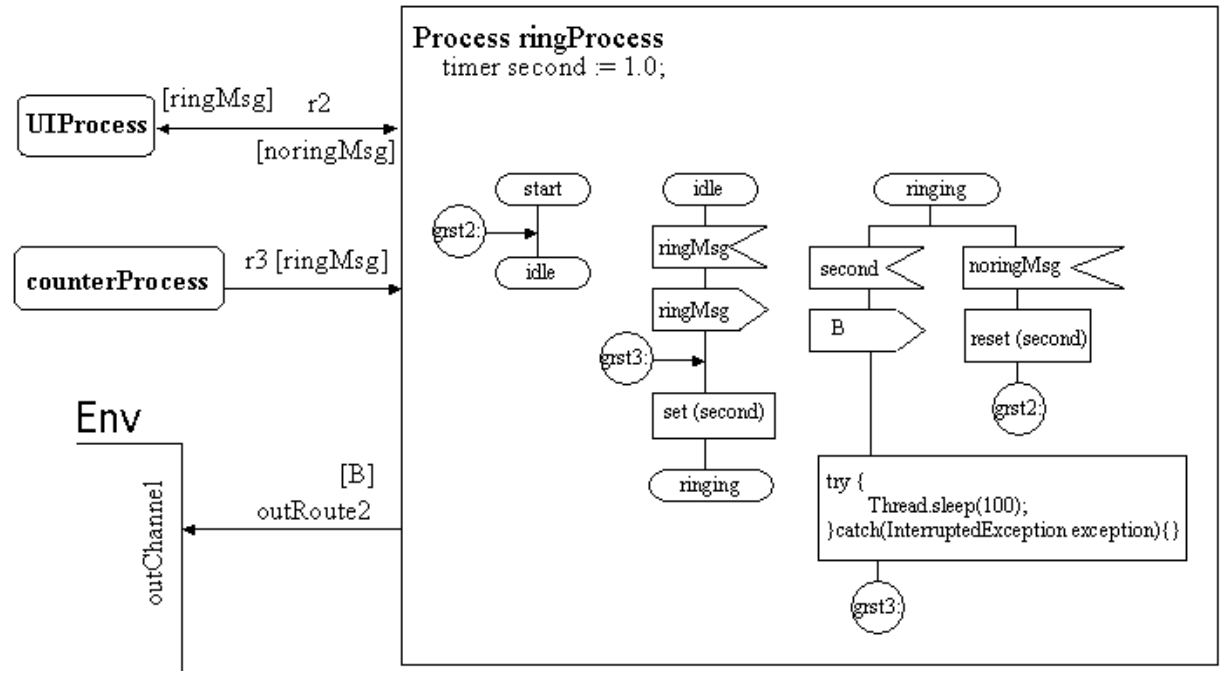

Figura 24: Representação gráfica de disparador de alarme (**RingProcess**)

A transição deste estado ocorrerá quando um sinal de entrada **noringMsg** for enviado pelo **UIProcess**, utilizando-se o canal **r2**. Quando ocorrer esta transição, o relógio retornará ao estado inicial e aguardará por entradas que simbolizam incremento, contagem ou limpeza.

Assim, como discutido anteriormente, esta parada é representada pelos sinais de entrada **C**, **S** e **M** respectivamente simbolizados como **clearMsg**, **startMsg** e **incrMsg**, quando o relógio encontra-se nos estados de *ringing* e *counting*.

Para o estado *ringing* do relógio, as entradas **C**, **S** e **M**, simbolizam o final da execução do alarme. A funcionalidade semelhante a esta, é encontrada no estado *counting*, interrompendo a contagem, com a entrada do sinal **S**, sendo os sinais **M** e **C**, interpretados como entradas inválidas.

Desta forma, o sinal que representa o alarme (**B**) no estado *ringing*, no momento da contagem, significa uma entrada inválida de sinal.

4.2.1. Especificando o Serviço em SDL

Este item, tratará de questões referentes à especificação de um sistema em **SDL**, o qual deverá ser utilizado como entrada para o Gerador de serviços RGB Java.

A especificação do sistema **KitchenTimer** está disponível no **Apêndice A**, a qual não será tratada neste momento. As questões levantadas neste item, serão referentes às limitações presentes no gerador de serviços e as melhores metodologias, em função da estrutura do gerador, para se especificar um serviço RGB Java em **SDL**.

É importante compreender que a especificação dos sistemas em **SDL**, no que se refere aos serviços do Ambiente RGB Java, depende primordialmente das características do Gerador de serviços RGB Java.

As limitações presentes na geração dos serviços RGB Java são justificáveis através dos conceitos de baixo acoplamento de alta coesão, aplicáveis na produção de software com qualidade.

Uma das limitações é referente ao número de blocos especificados para agregar os processos **SDL**. Estes devem ser reduzidos a uma unidade, ou seja, os processos devem estar contidos em um único bloco (coesão).

A outra limitação é a falta do mecanismo de herança aplicáveis nos processos; a herança possibilita a utilização dos atributos especificados nos níveis superiores ao processo, nos blocos e no sistema.

Contudo, outras características da Linguagem **SDL** foram preservadas para que o desenvolvimento dos serviços não tornasse o processo de especificação ambíguo ao que se pretendia atingir com o projeto.

As declarações de variáveis na especificação de serviços devem seguir os padrões de escrita da Linguagem **SDL**, porém, é importante ressaltar que estes serviços serão executados por uma máquina virtual Java, após serem gerados e compilados pelo Java.

As variáveis definidas nos processos, tornar-se-ão atributos privados de uma classe. Conseqüentemente os tipos destas variáveis deverão estar de acordo com a especificação Java.

Um sistema deve necessariamente pertencer a um pacote. Esta é a maneira como o Ambiente, o provedor, invocará este serviço, utilizando o nome do pacote e o da classe principal, que terá o mesmo nome que o sistema.

Esse tipo de abordagem auxilia na criação dos serviços de forma a organizá-los em pacotes, para a identificação dos nomes de classes, as quais serão geradas pelo parser. Caso dois serviços tenham processos com o mesmo nome, os pacotes poderão estar reduzindo esta ambigüidade.

A definição das classes importadas é realizada no nível do sistema, portanto, cada uma das classes resultantes na geração do serviço, conterá a mesma declaração de importação. O que foi definido no sistema, tem validade para todos os processos.

Esta questão implica na maneira como as tarefas (*task*) utilizarão classes pertencentes a outros pacotes.

As tarefas são blocos contidos nos processos, onde é possível fazer comentários e especificar uma computação em outras linguagens de programação.

Se uma das funcionalidades de um serviço requerer o acesso a um determinado Banco de Dados, as instruções para que tal tarefa se realize, podem estar especificadas dentro de uma *task*.

Um processo de serviço RGB Java deve ter a capacidade de utilizar, por exemplo, uma classe **Vector**[38], contida no pacote **java.util**. Sendo assim, o sistema deverá especificar a importação deste pacote.

Um processo deve conter um ou mais estados, além do estado *start*[39]. Cada um desses estados é guiado pelas entradas, já que estas representam as entradas de uma máquina de estados finitos.

Desta maneira, os processos devem ser compostos por estados e os estados devem ter uma ou mais entradas, para que uma transição possa ocorrer.

Não é obrigatório que cada estado tenha uma saída, porém, o conjunto de processos que compõe um sistema, deve especificar uma saída para o ambiente (**env**[40]). Assim como o conjunto de processos deve especificar uma entrada que receba sinais do ambiente. Somente desta maneira, um Cliente, poderá enviar e receber as mensagens com as informações processadas pelo serviço.

Os processos comunicam-se por especificações das instruções de entrada e saída (*input* e *output*), pelas das rotas de sinais. A existência de dois ou mais processos em um bloco requer que tais instruções sejam utilizadas.

Os rótulos e saltos incondicionais, *label* e *join*, devem ser especificados para que se implementem laços de repetição dentro dos processos de serviços RGB Java.

A **Figura 25**, ilustra de maneira didática, como um serviço deverá ser implementado em Java, quando os rótulos e desvios incondicionais são especificados em **SDL**.

[38] Contida no pacote **java.util** da Linguagem Java, esta classe destina-se a encadear objetos de maneira indexada.
[39] Estado comum a todos os processos especificados na Linguagem **SDL**; simboliza o início em uma simulação do processo.
[40] Ambiente; fora dos limites de uma especificação do sistema;

O desvio, representado pelo ultimo nó **SDL** do item "**b**", é especificado em Java, pelo retorno do método, e o rótulo, penúltimo nó do item "**a**", por um método, qual contém uma estrutura de repetição "enquanto" (*while*).

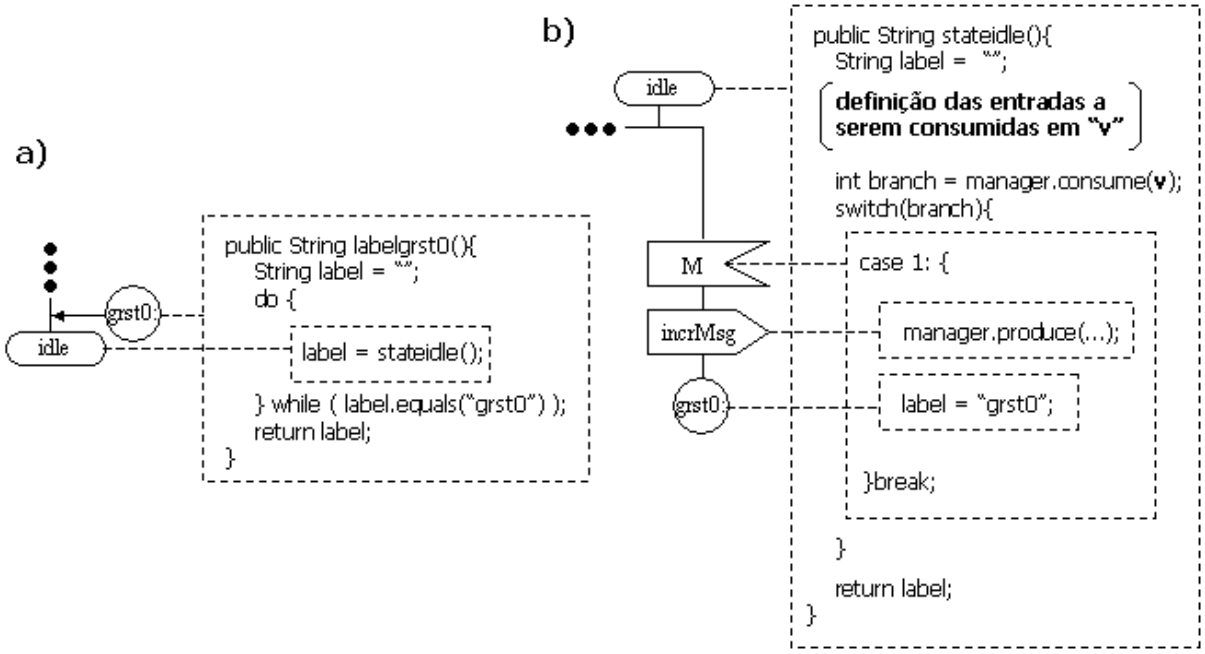

Figura 25: Uso de rótulos e desvios (label/join) em serviços do Ambiente RGB Java.

No momento em que ocorre um desvio, a variável nomeada como **label**, recebe o conteúdo do rótulo a ser alcançado. No final do método que descreve o estado *idle*[41], o valor desta variável **label** é retornado para a chamada que originou a transição do estado.

A variável **label**, descrita no item "**a**" da **Figura 25**, deve conter o valor de um desvio, que procura por um rótulo. Caso este valor seja igual ao do rótulo corrente, uma nova chamada ao estado *idle* deverá ser feita, fazendo com que o fluxo do programa siga a ordem da especificação, caso contrário, este valor é retornado a um nível superior.

Um estado é alcançado através da invocação do seu respectivo método, utilizando-se a produção *nextstate* (próximo estado), especificável em **SDL**. Desta forma, é preferível a utilização de rótulos e desvios, no que se refere aos laços de repetição.

[41] Espera; estado que simboliza a espera por entradas.

Um serviço especificado em **SDL** não deve utilizar as produções *nextstate*, para retornar a posições anteriores, e sim para se alcançar estados subseqüentes.

Após a compreensão das questões limitativas, abordadas acima, será possível especificar serviços na Linguagem **SDL** para que este sirva de entrada no processo de geração dos serviços RGB Java.

4.2.2. Gerando o Serviço RGB Java

Após ter configurado o Ambiente RGB Java e ter especificado um serviço na Linguagem **SDL**, é possível gerar a implementação do serviço RGB Java correspondente a esta especificação.

Para exemplificar a geração de um serviço, será utilizada a especificação do relógio de cozinha, o **KitchenTimer**, contida no **Apêndice A**.

Essa especificação deverá ser armazenada em um arquivo com a extensão "**sdl**", como **KitchenTimer.sdl**, colocada num diretório chamado **util**.

A ordem dos diretórios, definida acima, é dependente da especificação do pacote contida no **KitchenTimer.sdl**. Nesse caso, o pacote foi definido como **util.KitchenTimer**.

Quando um pacote contém mais de um nome, separados por ponto, o arquivo fonte, a especificação **SDL**, deverá estar armazenada dentro do diretório com o mesmo nome da penúltima palavra do pacote.

Os arquivos resultantes do processo de geração de um serviço serão armazenados em um diretório subseqüente, sendo esse diretório, nomeado com a última palavra da especificação do pacote, neste caso, o **KitchenTimer**.

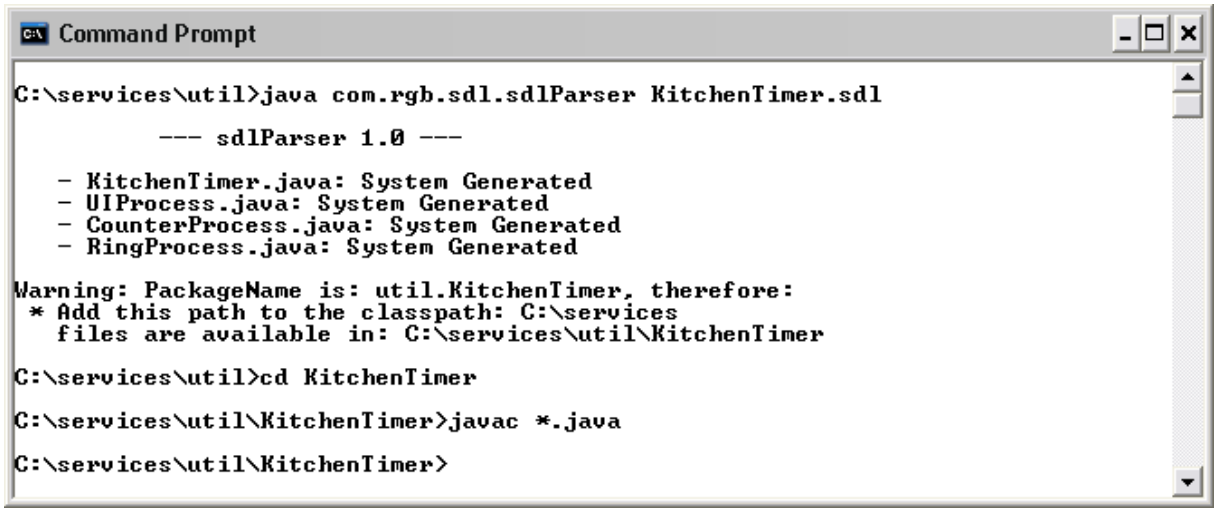

Figura 26: Processo de geração do serviço **KitchenTimer**.

As classes do Gerador de serviços, assim como foi mostrado na **Seção 4.1**, estão armazenadas dentro de um arquivo **JAR**, para que possam ser configuradas no sistema, invocáveis através do pacote **com.rgb.sdl**.

A classe **sdlParser**, contida no pacote do Gerador de serviços, é executada (**Figura 26**) pelo Java, recebendo como parâmetro de entrada, o arquivo **KitchenTimer.sdl**, o qual contém a especificação **SDL**.

Ao final deste processo, o gerador cria o diretório **KitchenTimer**, para que se possa armazenar a implementação do serviço, na forma de arquivos Java.

Essa implementação contida nos arquivos Java, após ser transformada em *ByteCode*[42], deverá ser configurada na variável de ambiente **classpath**, tornando-se assim, um serviço RGB Java.

O processo de criação dos *ByteCodes*, também ilustrado na **Figura 26**, é representado pelo acesso ao diretório KitchenTimer e pelo comando "**`javac *.java`**". Caso não existam mensagens de erro, significa que o serviço foi gerado de acordo com uma especificação Java.

[42] Linguagem utilizada pelo Java para expressa um código livre de plataforma.

4.2.3. Configurando o Serviço no Ambiente RGB Java

O Provedor de serviços instancia uma implementação de serviço através da invocação do seu pacote, seguido da sua classe principal, a qual é descrita com o mesmo nome da especificação do sistema em **SDL**.

Para que um serviço seja invocado pelo provedor, o seu pacote deve estar sendo referenciado pela variável de ambiente **classpath**.

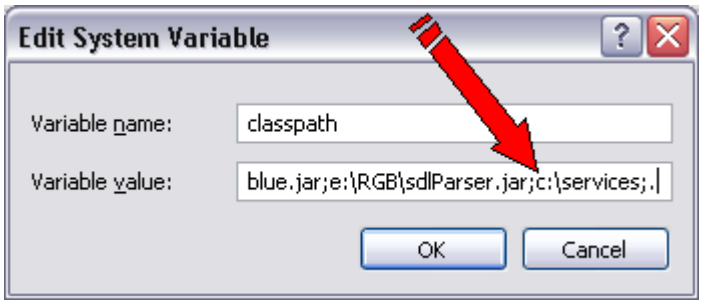

Figura 27: Configurando o serviço **KitchenTimer** na variável de ambiente **classpath**.

No caso do **KitchenTimer**, como o diretório **util** foi criado a partir de um outro chamado service, a variável de ambiente **classpath** (**Figura 27**) deve então, conter o caminho "**c:\service**", para que se atinja o pacote **util.KitchenTimer**.

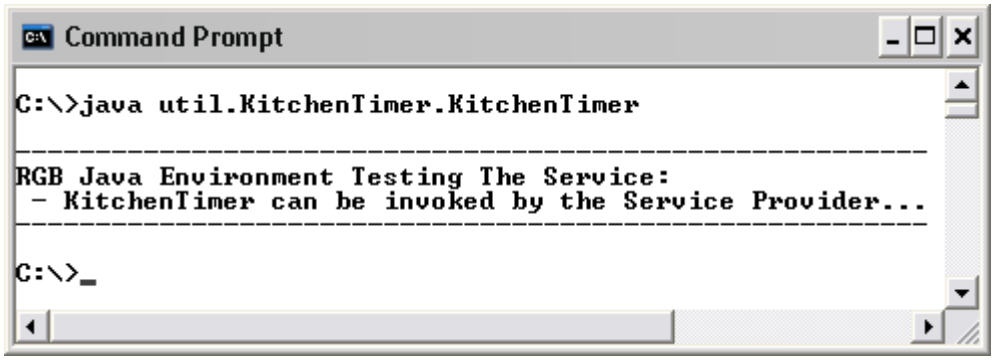

Figura 28: Conferindo se o serviço KitchenTimer está configurado corretamente no ambiente.

Para conferir se o serviço **KitchenTimer** está corretamente configurado no Ambiente, é possível executar a sua classe principal, utilizando a descrição do seu pacote, assim como foi ilustrado na **Figura 28**.

O caminho **util.KitchenTimer** é referente ao pacote do serviço e o próximo **KitchenTimer**, referente ao nome da classe principal, que tem o mesmo nome do sistema especificado na Linguagem **SDL**. "`java util.KitchenTimer.KitchenTimer`" retirado da **Figura 28**.

Da mesma maneira que esse serviço deverá ser invocado pelo Provedor, que não depende de um diretório específico para ser executado, o processo acima poderá ser realizado em qualquer parte da máquina, como por exemplo, na raiz da unidade **C**.

Se a mensagem exibida na execução da classe, for igual à ilustrada na **Figura 28**, significa que o serviço está corretamente configurado no Ambiente RGB Java.

4.2.4. Instanciando o Registro de Serviços

O Registro de serviços é a parte do Ambiente RGB Java responsável por armazenar as informações de instância dos serviços e Clientes de serviços. Essas informações indicam a localização na rede das aplicações que se conectam ao Ambiente, ou seja, o número IP e a porta do processo em execução.

Quando um serviço torna-se uma instância, o Provedor comunica-se com o Registro e informa sua localização na rede. Quando um Cliente faz um pedido de conexão dessa instância de serviços, o Registro responde com as informações de localização.

De uma forma simplificada, o Registro de serviços é a interface entre Clientes de serviços e serviços, para que se reduzam os esforços desses, quando há um pedido de conexão por parte do Cliente.

Além de armazenar as informações de instâncias, das aplicações conectadas a arquitetura, o Registro armazena também a publicação dos serviços, para que um usuário

tenha a capacidade de identificar e escolher um serviço baseado em uma descrição de suas funcionalidades.

Para que um serviço seja publicado, o Provedor deve informar ao Registro, o seu nome e um resumo descritivo das suas funcionalidades. Após a publicação, o Registro deve retornar o número identificador (**serviceID**), que representar este serviço, quando um programa cliente faz um pedido de conexão.

Apesar do Registro armazenar as informações de publicação, não foram criados no Ambiente RGB Java mecanismos para que esta tarefa seja feita através de uma aplicação, de forma automatizada.

O processo de publicação é realizado com a inclusão de um registro na tabela **Services**, de forma manual (**Figura 29**), dentro do Banco de Dados **Registry** (**Figura 4**), que representa o Registro de serviços. O *script* de geração do Banco de Dados **Registry** pode ser encontrado para consulta, no **Apêndice E**.

A falta desse processo automatizado, para a publicação de um serviço, é explicada pelo fato de não existir, no projeto de um Ambiente de desenvolvimento de software baseado em serviços, um consenso sobre a forma como um serviço deve ser descrito.

O Registro usa um Banco de Dados **MySQL** para armazenar as informações de instância dos serviços e dos Clientes de serviço. O sistema gerenciador de banco de dados **MySQL** foi escolhido para armazenar as informações do Registro de serviços, pois sua utilização para fins de projeto é considerada gratuita.

A **Figura 29** ilustra um registro da tabela **Services**, do Banco de Dados **Registry** sendo adicionado. Esse processo representa a publicação de um serviço e, é o único que deverá ser realizado de forma manual, pois as informações de instância são armazenadas no Banco de Dados através do Ambiente.

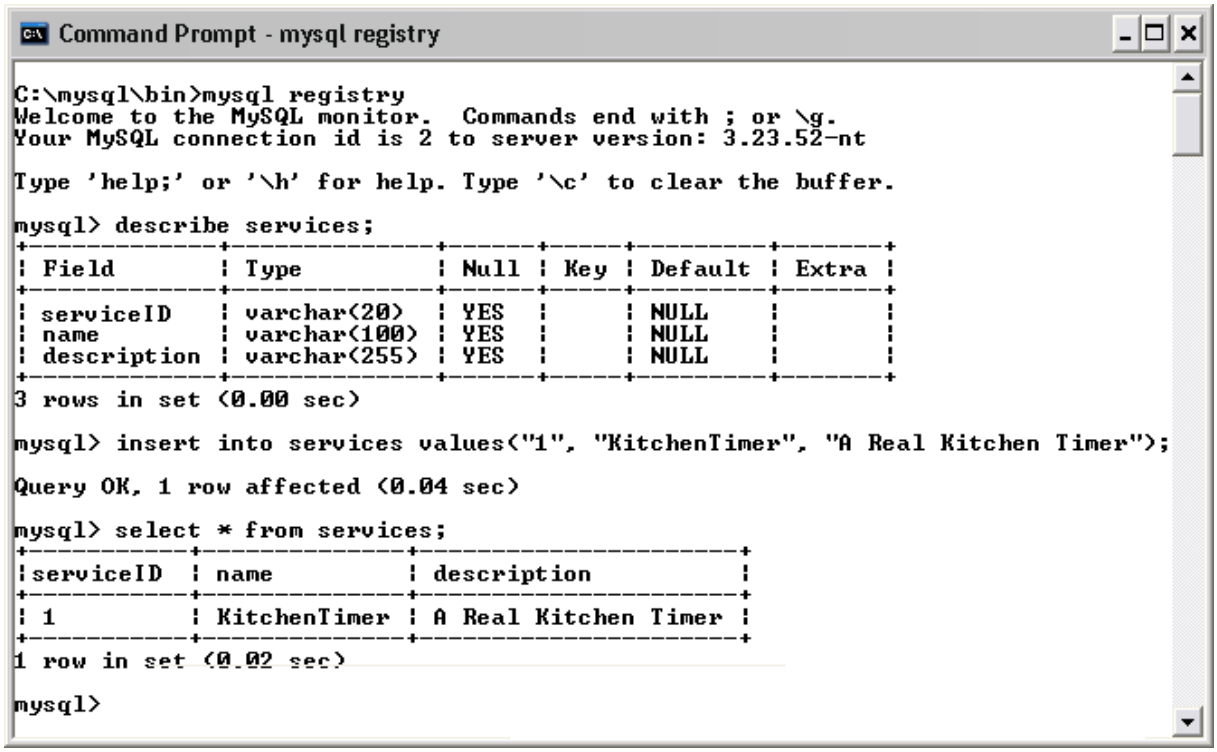

Figura 29: Publicando o serviço **KitchenTimer** de forma manual.

A partir do momento em que o serviço **KitchenTimer** foi publicado, uma instância desse serviço pode tornar-se disponível através do Ambiente RGB Java, representada pelo identificador, o **serviceID**.

Para que o serviço **KitchenTimer** seja disponibilizado para o uso de suas funcionalidades, através do Ambiente RGB Java (do Provedor de serviços), o Registro deve estar em estado de execução. Essa tarefa pode ser realizada assim como mostra a **Figura 30**.

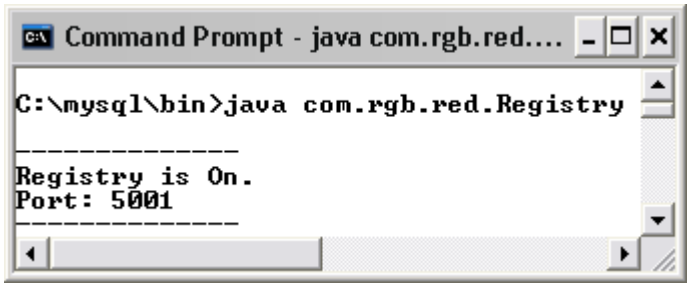

Figura 30: Instanciando o Registro de serviços.

O Registro, ao entrar em execução, reserva uma porta do sistema operacional para que outras partes do Ambiente tenham a capacidade de se conectarem com ele. Essa informação é mostrada na **Figura 30**, o número 5001.

O Provedor de serviços, obtendo as informações de localização do Registro na rede (o endereço IP e a Porta do sistema operacional), pode instanciar o **KitchenTimer**, fazendo com que seus dados sejam informados ao Registro.

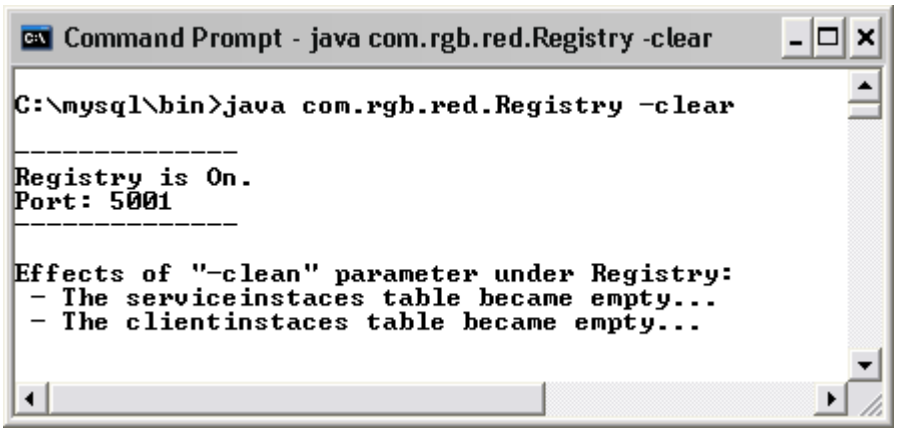

Figura 31: Apagando as informações de instância na execução do Registro de serviços.

Se o Registro for instanciado com a opção **-clear** (**Figura 31**), as informações de instância, tanto dos serviços quanto dos Clientes de serviço, serão apagadas. A utilização dessa opção torna-se interessante quando o registro contém informações inconsistentes.

O Registro pode ser instanciado em qualquer parte da máquina, desde que o caminho para o seu pacote esteja configurado na variável de ambiente **classpath**.

4.2.5. Instanciando o serviço KitchenTimer com o Provedor de Serviço

A etapa de instanciação de um serviço é dependente de todas outras citadas acima. Como o estudo de casos é baseado na utilização do serviço **KitchenTimer**, instanciá-lo torna-se possível, pois o Ambiente foi configurado, o serviço especificado em **SDL**, transformado

em um serviço do Ambiente RGB Java, compilado, "testado", adicionado no Ambiente, publicado e o Registro de serviços está em estado de execução, aguardando por requisições dos Provedores ou dos Clientes de serviço.

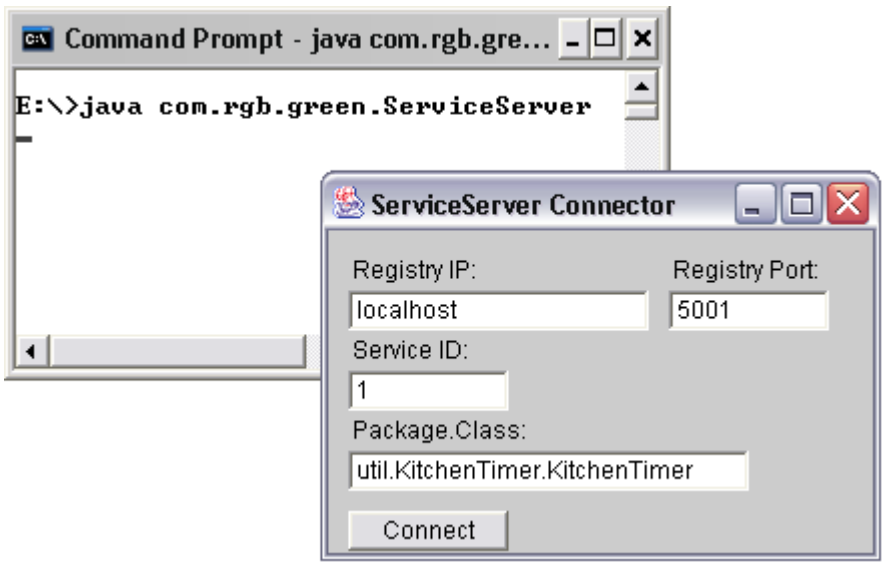

Figura 32: Localizando o serviço **KitchenTimer** no Ambiente local para instanciá-lo.

Na **Figura 32**, os campos "**Registry IP**" (endereço IP do Registro) e "**Registry Port**" (porta do sistema operacional) contém as informações necessárias para estabelecer uma conexão entre Provedor e Registro de serviços.

Após essa conexão, o Provedor deve informar o número identificador do serviço, representado pelo campo "**Service ID**" na **Figura 32**, para o Registro, que deve procurar por uma publicação de serviços com esse identificador.

Esse identificador é o número gerado no processo de publicação. Nesse momento, na instanciação de um serviço, ele tem a função de representar inúmeras instâncias de um único serviço.

Caso não exista um serviço publicado com o identificador "1", valor contido no campo "**Service ID**" (**Figura 32**), o processo de instanciação, que é realizado quando o botão

"**Connect**" da interface gráfica do provedor de serviços é pressionado, deve retornar uma mensagem de erro, informando que não há serviços com este identificador publicado.

Quando o processo de instanciação do serviço é concluído com êxito dentro do Registro, o Provedor de serviços deve então instanciar o **KitchenTimer**, utilizando as informações descritas no campo "**Package.Class**", ilustrado na **Figura 32**.

O Registro, para completar a operação citada, detecta as informações de localização do serviço, o endereço IP e a porta que o identifica na rede, adicionando-as em uma tabela de instâncias de serviço, a **ServiceInstances**, pertencente ao Banco de Dados Registry (**Figura 4**).

O Provedor localiza o serviço **KitchenTimer** através do seu pacote, o "**util.KitchenTimer**", seguido do nome da classe principal, a "**KitchenTimer**", que tem o mesmo nome do sistema especificado em **SDL**.

A localização do serviço em uma máquina é dependente da configuração desse no ambiente (**classpath**), processo descrito na **Seção 4.2.3**.

Dessa forma, todo pedido de conexão ou troca de mensagens, provenientes de um Cliente de serviços, deve utilizar-se do Ambiente Provedor para que se tenha acesso à instância do serviço **KitchenTimer**.

As mensagens codificadas em **XML** são convertidas para uma estrutura de dados dentro do Ambiente Provedor, servindo, dessa maneira, de entrada para o serviço instanciado.

No momento, é importante compreender que o serviço **KitchenTimer**, instanciado pelo provedor está disponível para ser utilizado e, existe um conjunto de objetos, Ambiente Provedor, servindo de interface nas trocas de mensagens e pedidos de conexão entre instância de serviço e Cliente de serviço.

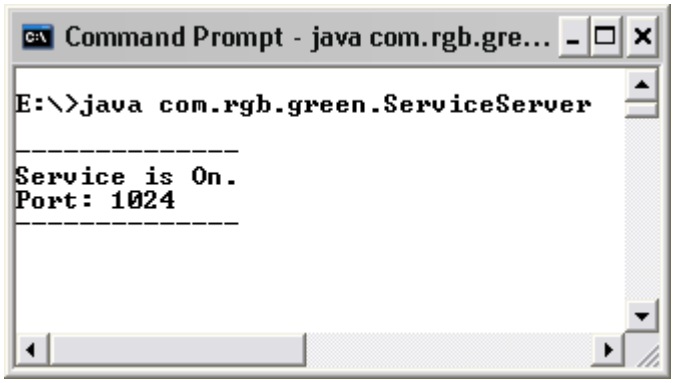

Figura 33: O serviço **KitchenTimer** instanciado.

A **Figura 33**, ilustra o término do processo de instanciação do serviço **KitchenTimer**.

4.2.6. Fazendo uso das funcionalidades do serviço KitchenTimer

A aplicação que utiliza as funcionalidades de um serviço é chamada de "programa cliente". Esse se conecta ao Ambiente RGB Java através da API Blue (Cliente de serviços), que tem as rotinas adequadas para procurar um determinado serviço e requerer uma conexão.

Para exemplificar a utilização das funcionalidades do serviço **KitchenTimer**, através de um programa cliente, será utilizada a aplicação especificada no **Apêndice B**, a classe **KTClient**.

A **KTClient** importa as classes do pacote **com.rgb.blue**, que deverão ser utilizadas para que o cliente conecte-se ao Ambiente. Essa conexão é estabelecida com a instanciação da classe **BlueClient**, pertencente ao pacote acima citado.

Uma das tarefas realizadas pela **BlueClient** é a de abrir uma conexão com o Registro de serviços para que seja informado o serviço que deverá ser utilizado, representado pelo seu identificador.

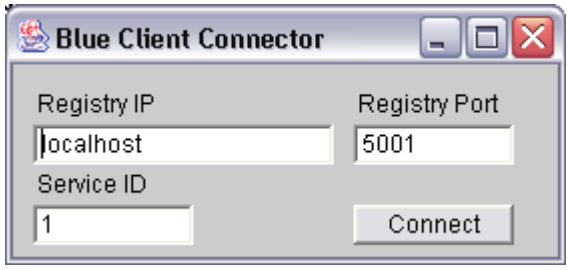

Figura 34: A classe **BlueClient** comunicando-se com o registro de serviços.

A **Figura 34** ilustra a comunicação entre Cliente e Registro de serviços. As três informações dispostas são referentes à localização do Registro ("**Registry IP**" e "**Registry Port**") e a identificação do serviço ("**Service ID**").

Quando a classe **BlueClient** é instanciada sem os parâmetros acima, o Ambiente RGB Java Cliente instancia uma classe para auxiliar a conexão entre Registro de serviços e clientes, essa ilustrada pela **Figura 34**.

O Registro de serviços, ao conectar-se com o Cliente, recebe o identificador do serviço (**serviceID**), procura uma instância e, caso encontre, envia uma mensagem ao Ambiente RGB Java Cliente com informações referentes à localização do serviço, endereço IP e porta do sistema operacional.

Essas informações são recebidas pelo Ambiente RGB Java Cliente através de um servidor instanciado para funcionar concorrentemente à conexão padrão. Este servidor, o **ServiceChange**, existe para que o Ambiente Cliente possa receber mensagens do Registro ao mesmo tempo em que está conectado a uma instância de serviço.

Para que o Registro encontre a instância **ServiceChange** de um determinado Cliente, a porta de mudança **changePort** deve ser informada, no momento em que o cliente envia as informações requerendo a utilização de um serviço.

A **changePort**, junto com o endereço IP do cliente, representam as informações de localização do servidor de mudança, o **ServiceChange**, na rede.

O Registro, após enviar as informações de localização do serviço ao Cliente, inclui na tabela de instância de clientes, a **ClientInstances** (**Figura 4**), as informações de uso do serviço e a localização do Cliente, o endereço IP e a porta de mudança.

Dessa forma, o Ambiente pode notificar os Clientes com novas informações de localização de instâncias de serviços quando a instância do serviço que esses Clientes estão utilizando, perder a conectividade (terminar – ver **Seção 3.2.5**).

A capacidade de re-conectar Clientes a outras instâncias do mesmo serviço é parte da característica de gerenciamento provido pelo Registro de serviços.

Após o recebimento das informações de localização de instância do serviço, o Ambiente Cliente conecta-se com o Provedor de serviços, mudando assim, para o estado de "pronto para a utilização" das funcionalidades do serviço.

O **KTClient** utiliza o Ambiente Cliente através da classe **BlueClient**, invocando o método "**`public void setXml(String message)`**".

Esse método, especificado na **BlueClient**, envia as mensagens codificadas em **XML** ao Ambiente de serviços, o Provedor a processa, identifica uma funcionalidade e executa-a, utilizando a instância do serviço.

A comunicação entre Cliente e serviço, no Ambiente RGB Java, é feita de maneira assíncrona. Uma funcionalidade do serviço pode responder ao Cliente, o resultado do processamento, ou simplesmente acumular um valor em uma variável. Essa mesma funcionalidade, dependendo do seu estado, pode responder inúmeras vezes ao Cliente.

A falta de sincronia faz com que o tráfego na rede seja menos intenso, partindo do princípio de que o serviço não necessita de uma requisição do Cliente para originar uma resposta.

Se a falta de sincronia contribuiu na performance, essa característica tornou a implementação do Ambiente complexa, não na sua utilização e sim no seu desenvolvimento.

A Ambiente faz com que a utilização dos serviços seja facilitada, pois o gerenciamento realiza tarefas que um programador de sistema teria que desenvolver se o Ambiente não tivesse esse grau de complexidade.

Uma mensagem pode ser enviada ao serviço, sem que se necessite desenvolver todo o caminho que essa deve percorrer até atingir o seu destino.

Da mesma maneira que um Cliente envia uma mensagem ao serviço, o serviço, em um determinado momento, pode enviar uma resposta como resultado do seu processamento, ao Cliente.

Essa mensagem, originada do serviço, é capturada pelo Ambiente Cliente, utilizando uma classe chamada **ClientImput**.

A **ClientImput** funciona concorrentemente às outras classes da API Blue, aguardando por mensagens codificadas em **XML** originadas do serviço.

Quando a **ClientImput** recebe uma mensagem, é transformada em sinais, que deverão ativar funcionalidades, pré-definidas pelo **KTClient**, dentro do método "**`public void execute(DataStructure ds)`**".

Este método é invocado pela **ClientImput** toda vez que esse recebe uma mensagem, passando por parâmetro, as informações dos sinais, que foram geradas dentro do serviço.

Se uma instância do serviço **KitchenTimer** envia uma mensagem ao Cliente, utilizando o canal **outRoute3** com o sinal **B** (*Beep*), o **KTClient** deve implementar dentro do método "**`public void execute(DataStructure ds)`**" uma funcionalidade que produza um aviso sonoro.

Dentro do **ds**, variável que contém uma estrutura de dados, são disponibilizadas as informações agrupadas pela **ClientImput**. Essas informações podem ser recuperadas através do método "**`public String get(String s)`**", especificado na classe **DataStructure**.

O método "**`public void execute(DataStructure ds)`**" para identificar uma mensagem de alarme (*Beep*) originada no serviço, pode ser implementado desta maneira:

```
1    public void execute(DataStructure ds){
2       String channel = ds.get("channel");
3       String signal = ds.get("signal");
4       if (channel.equals("outRoute3")){
5          if (signal.equals("B")){
6             java.awt.Toolkit.getDefaultToolkit().beep();
7          }
8       }
9       blue.setFreeForNext();
10   }
```

Figura 35: Implementando a funcionalidade de alarme no programa cliente **KTClient**.

Esse método deve ser especificado dentro do **KTClient**, para que uma mensagem originada da instância do serviço, encontre um destino, ou seja, uma finalidade.

Na **Figura 35**, é possível identificar, na linha **9**, uma chamada de método **`setFreeForNext()`**, especificada pela classe **BlueClient**. Essa chamada faz com que o Ambiente Cliente prepare-se para consumir a próxima mensagem originada pela instância do serviço.

Caso essa linha não seja implementada dentro do método execute, o Ambiente Cliente poderá somente consumir a primeira mensagem proveniente da instância do serviço.

Desta forma, o cliente informa ao Ambiente que só poderá executar outra funcionalidade, após ter terminado a atual. Essa abordagem sugere que as mensagens provenientes do serviço sejam executadas, uma de cada vez, na ordem em que elas aparecerem para a instância da classe **ClientImput**.

Enquanto uma mensagem está sendo decodificada e executada dentro do **KTClient**, a instância da classe **ClientImput** encontra-se no estado "esperando" (*waiting*), evitando o uso de laços infinitos, por consumirem muito do processamento.

A chamada de método, encontrada na linha **9**, da **Figura 35**, muda o estado da instância da classe **ClientImput** para "pronto para receber mensagens".

Caso a instância do serviço envie muitas mensagens ao Cliente, e essas cheguem antes que a primeira funcionalidade seja executada completamente, o **Socket** as armazena em uma fila para serem consumidas quando a classe **ClientImput** estiver no estado de "pronto".

Essa fila de espera é implementada pelo próprio Java, sendo desnecessário o desenvolvimento de uma fila de espera dentro do Ambiente Cliente.

4.2.7. Escrevendo um Cliente de Serviços

Sintetizando a **Seção 4.2.6**, um Cliente de serviços deve utilizar a classe **BlueClient**, pertencente ao pacote **com.rgb.blue**, como interface do Ambiente, que conectará Clientes e instâncias de serviços, através do Registro.

O método "**`public void execute(DataStructure ds)`**" deve ser implementado para que as mensagens provenientes do serviço possam encontrar uma correspondência dentro do cliente, provocando execuções pré-definidas.

O pacote **com.rgb.intersection** deve ser importado para que a classe **DataStructure** seja utilizada, assim como o pacote **com.rgb.blue** para a classe **BlueClient**.

Dentro do método **execute**, a variável **ds** deve ser manipulada, com a invocação do método "**`public String get(String s)`**", para que as mensagens codificadas na classe **ClientImput** sejam descobertas.

Ao final do método **execute**, a chamada de método "**`public void setFreeForNext()`**", especificada na classe **BlueClient** deve ser implementada para que a classe **ClientImput** possa consumir a próxima mensagem proveniente da instância do serviço.

Por último, as mensagens provenientes do Cliente, devem ser enviadas através do método "**`public void setXml(String message)`**", especificado na classe **BlueClient**.

Desta forma, é possível utilizar as funcionalidades de um serviço com um alto grau de coesão, com um baixo acoplamento, em poucas instruções.

A implementação do Cliente de serviços, o **KTClient**, está disponível para consulta no **Apêndice B**.

4.3. Estudo de Caso 2: sdlParserService

Para comprovar que um serviço pode ser utilizado para qualquer propósito, o Gerador de serviços RGB Java foi disponibilizado pelo Ambiente, o **sdlParserService**, tendo como funcionalidade, gerar serviços a partir de uma especificação SDL, ou seja, o serviço que gera serviços.

4.3.1 Especificando o serviço sdlParserService

A **Figura 36**, ilustra o **sdlParserService** em nível de sistema, para que se possa compreender as entradas e saídas presentes no serviço.

A rota de sinal **inRoute**, ligada ao processo **SDLTransformProcess**, é a "porta de entrada" do serviço, essa deverá receber um arquivo de especificação SDL, enquanto que a rota de sinal **outRoute**, ligada ao Ambiente, enviará ao programa cliente, em forma de mensagem codificada em **XML**, os arquivos Java que são os produtos da geração do serviço.

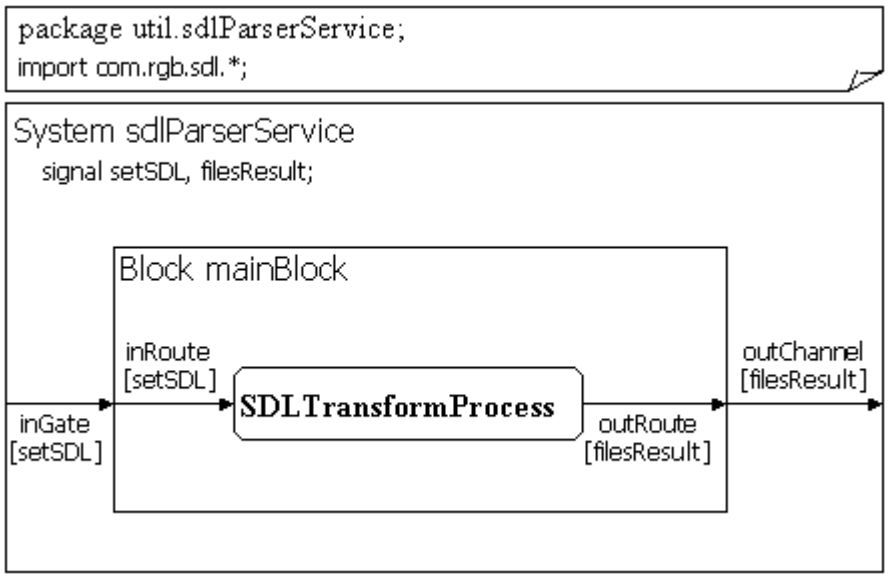

Figura 36: Representação gráfica do sistema **sdlParserService** em SDL.

O processo **SDLTransformProcess** contém as rotinas para que a operação de geração de serviços seja realizada. Essas rotinas devem ser descritas dentro de uma notação *task* **SDL** (tarefa), acessando o pacote do Gerador de serviços, o **com.rgb.sdl** e, utilizando suas funcionalidades.

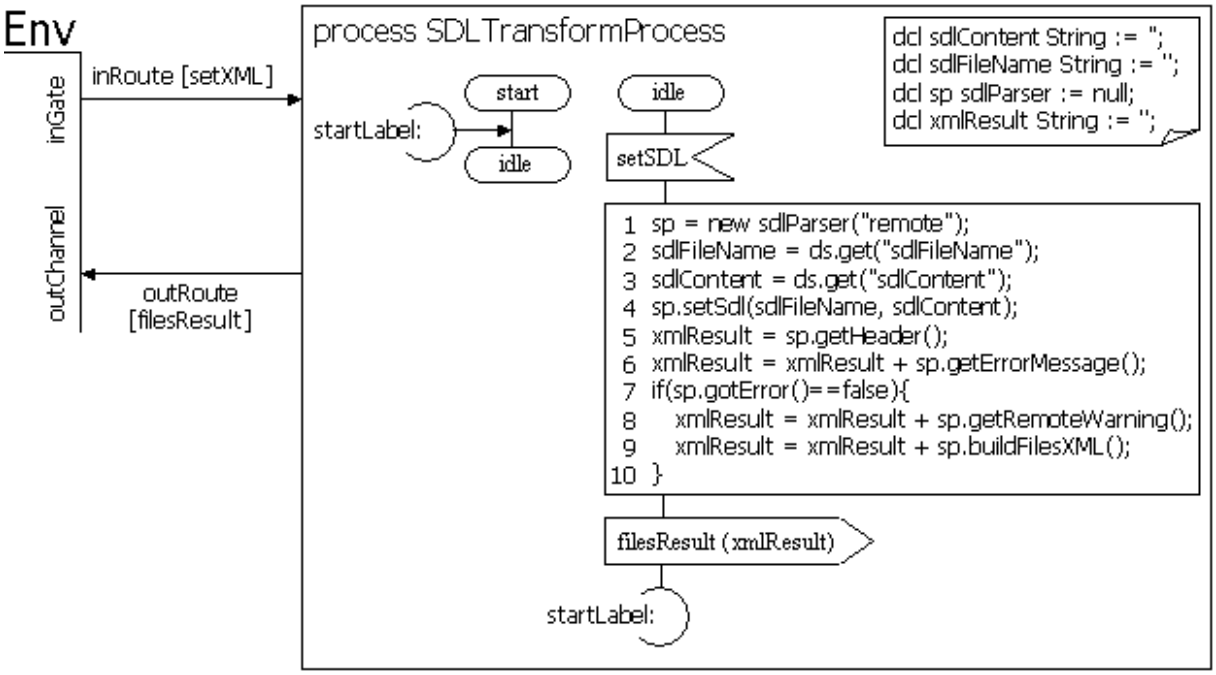

Figura 37: Representação gráfica do Processo **SDLTransformProcess**

Na **Figura 37**, pode-se visualizar, de forma gráfica, a especificação do processo **SDLTransformProcess** e, como as informações provenientes do Cliente são utilizadas na entrada para a Geração de serviços.

Na Linha **1** da *task*, o **sdlParser**, pertencente ao pacote **com.rgb.sdl**, é instanciado com o parâmetro "*remote*", significando que os arquivos de respostas, o conteúdo Java resultante da transformação, será estruturado dentro de uma mensagem codificada em **XML**.

Não é adequado gravar o resultado da transformação em arquivos, no local onde a instância do serviço é executada, pois essas informações devem ser enviadas ao programa cliente.

A mensagem gerada pelo **sdlParser** (o Gerador de serviços RGB Java) deve ser gravada, no Cliente, em forma de arquivos RGB Java, após ser extraída do seu formato XML.

Na instância de serviço, sempre que um comando de entrada, notação *input* em **SDL**, for especificada, o Ambiente de serviços disponibiliza uma variável do tipo **DataStructure**, nomeada como **ds**, para que o conteúdo de uma notação *task* possa manipular a estrutura de dados que representa a mensagem proveniente do Cliente de serviços.

Essa estrutura de dados é montada a partir da mensagem codificada em **XML**, enviada pelo Cliente ao serviço, dentro da classe **inService**.

As Linhas **2** e **3** da *task* (**Figura 37**), representam o nome do arquivo e o conteúdo em **SDL** respectivamente, da entrada. Na Linha **4**, essas informações são utilizadas pelo método "**`public void setSdl (String sdlFileName, String sdlContent)`**", da classe **sdlParser**, para dar início ao processo de geração do serviço.

Ao ser executada a Linha **4**, após a geração do serviço, esse, deverá conter no máximo quatro informações: erro (*error*), avisos (*warnings*), cabeçalho (*header*) e o conteúdo da geração de serviços.

Na Linha **5**, são capturadas as informações de cabeçalho, uma mensagem de apresentação do Gerador contendo o seu nome e a sua versão atualizada; na Linha **6**, as mensagens de erros, caso existam erros, ao contrário, conterá a cadeia de caracteres "*0 errors*"; caso não existam erros, o Gerador deverá produzir alguns avisos, informando ao Cliente como proceder após o processo de geração do serviço, produto da Linha **8** e; na Linha **9**, o conteúdo da geração do serviço.

A variável do tipo String, que acumula as informações de erros, avisos, cabeçalho e conteúdo da geração do serviço, a **xmlResult**, definida dentro do processo, servirá como parâmetro de saída para a próxima instrução *output*.

Essa variável, a **xmlResult**, é parte da especificação do serviço **sdlParserService**, não pertencendo a estrutura do Ambiente do provedor de serviços, diferentemente da variável **ds**, do tipo **DataStructure**, que a existência é definida a cada entrada **SDL**, especificada em Java.

O desenvolvedor de serviços não deve se preocupar com a definição da **ds**, e sim saber que está pronta para ser usada a cada entrada **SDL**.

Da mesma forma que um valor é passado por parâmetro em uma saída **SDL**, a instrução *input* pode recuperar este valor, caso a saída seja direcionada a um outro processo e não ao Ambiente.

Existem três formas de se ler valores de entrada após uma instrução de entrada **SDL**: usando a variável **ds**, a classe **String** ou um conjunto de **String**.

Como já pôde ser visto, usa-se a **ds** quando a entrada é ligada ao Ambiente, permitindo que o programador do serviço, tenha a capacidade de descobrir as mensagens do Cliente.

As outras duas maneiras são indicadas quando a comunicação é feita entre os processos, ou seja, quando a troca de informações está especificada dentro do serviço.

A variável **value**, armazena uma String quando a entrada que antecede o seu uso, está ligada a uma saída que usa a passagem de uma String como parâmetro, da mesma forma ocorre para a variável **values**, que é definida pelo Ambiente como um objeto da classe Vector, do pacote **java.util**.

Sendo assim, as variáveis **ds**, **value** e **values**, são reservadas pelo Ambiente para a passagem de parâmetro. Caso exista uma definição para uma delas dentro de uma *task* **SDL** ou em nível de processo, o compilador Java reportará um erro, apontando a duplicação de definições.

Um exemplo de passagem de parâmetro entre os processos pode ser visualizado no serviço **KitchenTimer**, na rota de sinal **r1**, com a mensagem **incrMsg**.

Utilizando a rota de sinal **r1**, o processo **UIProcess** envia o número de segundos que o contador deverá acumular para o **CounterProcess**. Após a entrada **M** do estado *idle*, a saída **incrMsg** passa um parâmetro da classe **String**.

O sinal de entrada **incrMsg** do processo **CounterProcess** recebe este parâmetro através da variável **value**, transforma-o e armazena-o em uma variável do tipo **int**, a **min**, acumulando o contador do relógio de cozinha.

A passagem de parâmetros entre Clientes e instância do serviço, permite que os processos manipulem as entradas do serviço, utilizando uma estrutura de dados para isto: a classe **DataStructure**.

A especificação do serviço **sdlParserService** está disponível para consulta no **Apêndice C**, em sua forma textual.

4.3.2. Escrevendo um Cliente para utilizar a funcionalidade do sdlParserService

O programa cliente deve enviar ao serviço, ao **sdlParserService**, uma mensagem codificada em **XML**, contendo uma especificação **SDL**, e o nome do arquivo em que ela deverá ser armazenada.

O **sdlParserClient**, disponível no **Apêndice D**, é um exemplo de um programa Cliente para o serviço **sdlParserService**. Este item reserva-se para fazer alguns comentários sobre as funcionalidades do programa Cliente, pois o mesmo processo foi amplamente abordado na **Seção 4.2.7**.

O programa cliente para este propósito, a geração de serviços, deve atender às requisições de entrada e saída do **sdlParserService**. Uma entrada para o **sdlParserService** é uma especificação de serviços **SDL**, a saída são implementações Java referentes a entrada.

O que representa uma entrada para **sdlParserService**, deve ser entendido como saída para o **sdlParserClient**, o que for usado como saída na instância do serviço, será a entrada para o Cliente.

A especificação do **KitchenTimer**, disponível no **Apêndice A**, pode ser utilizada como saída para o programa cliente, no entanto, as entradas deverão ser quatro arquivos: **KitchenTimer.java**; **UIProcess.java**; **CounterProcess.java** e; **RingProcess.java**.

Como essas entradas para o programa cliente são enviadas por um serviço, remotamente ativo, e como a comunicação entre serviços e Clientes de serviços é baseada na troca de mensagens no formado de texto, os arquivos acima citados, as entradas, serão entregues ao cliente, estruturados na forma de mensagem **XML**.

O programa cliente, ao receber este texto no formato **XML**, deverá armazená-lo como arquivos, seguindo algumas regras de estruturação definidas pelo **sdlParser**, o Gerador de serviços.

O **sdlParser** estrutura as saídas, as implementações de serviços, no formato de texto **XML**, enumerando as *tags* de acordo com a sua usabilidade.

De acordo com os índices, é possível reunir o nome do arquivo e o seu conteúdo, para que seja armazenado no Cliente.

No caso do **KitchenTimer**, o primeiro nome do arquivo pode ser capturado, procurando pela tag **fileName+n**, onde **n** é um número intero que começa com algarismo **1** e termina em **4**.

Exemplo: fileName1, fileName2, ... fileName4.

Para encontrar o conteúdo do arquivo correspondente ao arquivo RGB Java, deve-se procurar pela tag **content+n**, onde **n** é um número inteiro que começa em **1** e termina em **4**.

Exemplo: content1, content2, ... content4.

O conteúdo das *tags* fileName1 e content1, são informações referentes ao arquivo **KitchenTimer.java**, que deve ser armazenado na máquina local do Cliente.

Dando seqüência ao **n**, ter-se-ão as representações dos processos **UIProcess.java**, **CounterProcess.java** e **RingProcess.java**.

4.3.3. Gerando o serviço KitchenTimer através do sdlParserService

Para exemplificar o funcionamento do serviço **sdlParserService**, será usada a especificação do sistema **KitchenTimer** como entrada. O sistema **KitchenTimer** foi amplamente explicado na **Seção 4.2**, e encontra-se disponível também no **Apêndice A**, na sua forma textual.

Partindo do ponto em que o Ambiente está configurado; o serviço **sdlParserService** foi especificado, gerado, compilado, publicado no Registro com o identificador número **2**,

adicionado ao Ambiente; que o Registro foi instanciado; que o Provedor de serviços foi executado para o **sdlParserService**; o cliente pode ser executado.

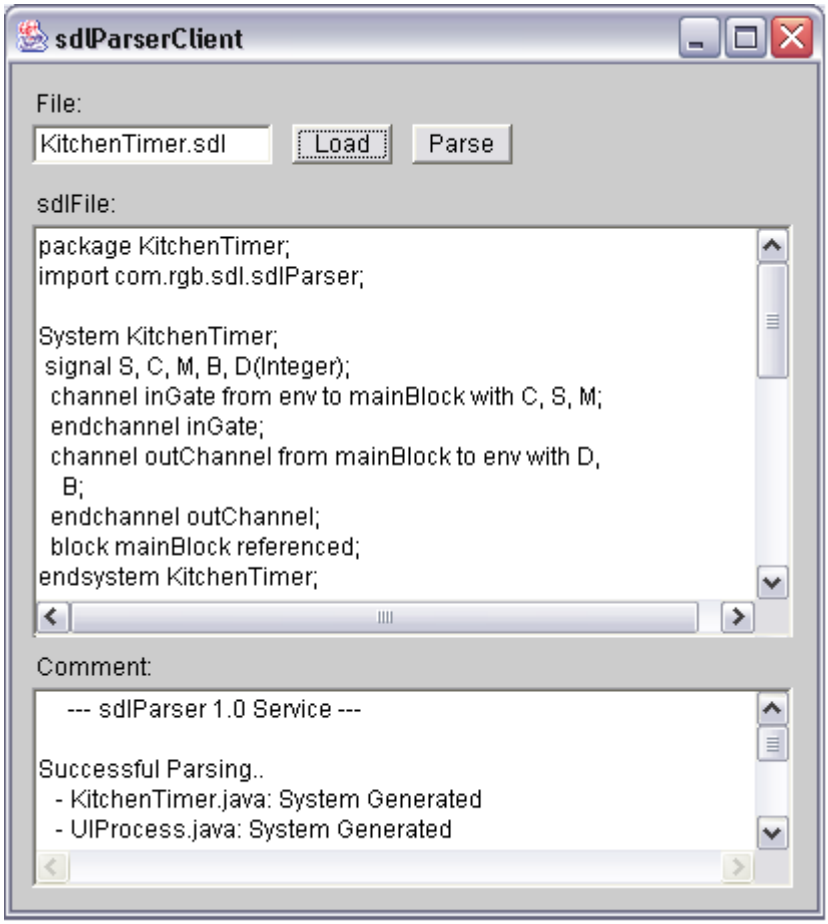

Figura 38: Gerando o serviço **KitchenTimer** através do **sdlParserService**.

Como pode ser visualizada pela **Figura 38**, a especificação do serviço **SDL** não pode conter mais de um nível na definição do pacote, ou seja, o pacote deve ter somente um nome, no caso, "**pacote KitchenTimer;**".

A caixa de texto "**comments**", retorna ao programador, o cabeçalho, os avisos e as mensagens de erro, que através da figura não puderam ser visualizados em sua totalidade.

A implementação do programa cliente, o **sdlParserClient**, está disponível para consulta, no **Apêndice D**.

4.4. Considerações Finais

Os estudos de caso propostos neste Capítulo servem de exemplo para a utilização do Ambiente RGB Java e do Gerador de Serviços RGB Java; a ordem dos itens é importante para a compreensão dos mecanismos de utilização do Ambiente, dessa forma, este Capítulo pode ser usado também como um guia funcional.

Os exemplos demonstram que um serviço pode ser especificado em uma linguagem abstrata (SDL) e convertida em Serviço RGB Java, provando que a utilização do Gerador de Serviços, como uma ferramenta do Ambiente, facilita a especificação dos serviços, por abstrair os aspectos mais técnicos da programação.

Os testes no Ambiente RGB Java foram executados de maneira superficial, sem a utilização de uma metodologia e ferramentas, porém o Ambiente mostrou-se estável na execução das instâncias dos serviços.

Da mesma forma, o construção e a utilização dos serviços por meio do Ambiente correspondeu a expectativas de redução de custos no desenvolvimento; este Capítulo também serve para que se possa comparar o desenvolvimento de software baseado em serviços com outras abordagens; mesmo não sendo realizada essa comparação no trabalho, é possível identificar que os mecanismos disponibilizados pelo Ambiente, facilita a utilização dos serviços.

5. CONCLUSÃO E TRABALHOS FUTUROS

Com o conhecimento das principais idéias sobre Arquitetura Orientada a Serviços, foi possível desenvolver o Ambiente RGB Java, onde suas características básicas foram reproduzidas com o intuito de dar suporte ao desenvolvimento de um gerador automático de gerviços que tem como entrada um sistema especificado na Linguagem SDL.

A revisão bibliográfica foi elaborada a partir das tecnologias e padrões que apóiam ou dão base ao desenvolvimento deste projeto. Sendo elas: XML, Java, Web Service, Componentes, Objetos distribuídos, conceitos e implementação de Arquitetura Orientada a Serviços, a Linguagem SDL e uma introdução à Ferramenta JavaCC.

O objetivo deste projeto foi realizado a partir da construção do Ambiente RGB Java e do Gerador de Serviços RGB Java; o Ambiente de serviços contribui para o desenvolvimento de software de qualidade, pois os serviços são gerados automaticamente por um programa, que tem como entrada uma especificação de sistema na Linguagem SDL.

O Ambiente permite que programas, serviços, comuniquem-se utilizando uma rede de computadores com pouco esforço, baixando os custos do desenvolvimento de software e aumentando a comunicação entre as equipes de desenvolvimento.

Referente à integração de sistemas, o Ambiente de desenvolvimento de serviços RGB Java mostrou-se adequado, pois foi possível conectar programas clientes com serviços de maneira menos custosa, enfocando o conteúdo das mensagens trocadas.

Os trabalhos levantados dão apenas uma direção do que já é consenso entre os grupos de pesquisa, como a estrutura básica de uma Arquitetura Orientada a Serviços e suas possíveis funcionalidades. Dessa maneira, ainda há muito para se desenvolver na área de Arquiteturas Orientadas a Serviços.

Apesar da Linguagem SDL ser uma boa alternativa para a especificação dos serviços, por ter a capacidade de representar sistemas coesos, provou-se que algumas características específicas não puderam ser descritas com o seu uso, como por exemplo, as informações de publicação.

Desta forma, os trabalhos futuros deverão ser referentes a uma proposta de extensão da Linguagem SDL para que se possa representar, de maneira mais adequada, os serviços.

Os outros trabalhos podem ser referentes ao desenvolvimento de uma interface gráfica de administração do Registro de Serviços; o tratamento de erros na especificação SDL para o Gerador de Serviço; o armazenamento dos estados dos Serviços; e contribuições na segurança do Ambiente.

REFERÊNCIAS

AGRAWAL, R; BAYARDO, R. J; GRUHL, D; PAPDIMITRIOU D. **Vinci: A service-oriented architecture for rapid development of web applications**. WWW10 – World Wide Web Conference, Hong Kong, 2001

BECKER, A. K.; CLARO, D. B.; SOBRAL, J. B. M. **Web Services e XML: um novo paradigma da computação distribuída**; OD' 2001 – Objetos Distribuídos, São Paulo – SP, Novembro/2001

BOSAK, J. **XML, Java, and the future of the web** Disponível em: http://www.xml.com/pub/a/w3j/s3.bosak.html Acesso em: 05/06/02

Cilia, M; Buchmann, A. **An Active Functionality Service for E-Business Applications**. ACM SIGMOD Record , 31(1), março 2002. Disponível em: http://acm.org/sigmod/record/issues/0203/SPECIAL/4.cilia-buchmann.pdf Acesso em: 05/11/02

COYLE, F. P. **Ubiquity: XML, Web Services and the changing face of distributed computing**, ACM Ubiquity Magazine, abril 2002. Disponível em: http://www.acm.org/ubiquity/views/f_coyle_1.html Acesso em: 02/10/02

DELAMARO, M. E. Como construir um compilador: utilizando a ferramenta JavaCC. 1.ed. São Paulo: Novatec, 2004.

ELLSBERGER, J. HOGREF, D. SARMA, **A. Formal Object-oriented Language for Communicating Systems**, Prentice-Hall, 1997

GISOLFI, D. **Fee-based web services: terminology**, IBM developerWorks, 2002 Disponível em: http://www-106.ibm.com/developerworks/library/ws-arc4/ Acesso em: 13/03/02

HORSTMANN, C.S.; CORNELL, G. JavaBeans. In:_ _ _ _ _ _.**Core Java 2**. 2.ed. São Paulo: Makron Books, 2001. v.2, cap.8, p.541.623.

MALDONADO, J. C. **Critérios Potenciais Usos: uma contribuição ao teste estrutural de software**. Tese de Doutorado, DCA-FEEC-UNICAMP, Campinas-SP, Brasil, Julho de 1991.

MOORE, J. W. **Software Engineering Standards: a user´s road map**. Los Alamidos: IEEE Computer Society, 2002

RICCIONI, P. R. **Introdução a Objetos distribuídos com CORBA**, Florianópolis: Visual Books, 2000.

SILBERSCHATZ A. GALVIN, P. GAGNE, G. **Sistemas Operacionais: conceitos e aplicações**. Rio de Janeiro: Campus, 2001.

STAL, M. **Web Services: beyond component-based computing**, ACM, 2002.

SUN MICROSYSTEMS. **JavaBeans Short Course:** Disponível em: http://developer.java.sun.com/developer/onlineTraining/Beans/JBShortCourse/beans.html Acesso em: 18/03/02

TANENBAUM, A. Distributed Operating Systems, New Jersey: Pentice-Hall,1995

VILELA, P. R. S. **Parser SDL** (material didático), 2002

W3C - **Extensible Markup Language (XML) 1.0 (Third Edition)**, 2004 Disponível em: http://www.w3.org/TR/2004/REC-xml-20040204/ Acesso em: 25/05/2004

W3C - Web Services Architecture: **Working Draft, 2002** Disponível em: http://www.w3.org/TR/2002/WD-ws-arch-20021114/ Acesso em: 24/01/2002

WIEGERS, K. E. **Read my lips: no new models!** IEEE Software, Setembro de 1998

APÊNDICE A – KitchenTimer.sdl

```
1     package util.KitchenTimer;
2     import com.rgb.sdl.sdlParser;
3
4     System KitchenTimer;
5      signal S, C, M, B, D(Integer);
6       channel inGate from env to mainBlock with C, S, M;
7       endchannel inGate;
8       channel outChannel from mainBlock to env with D,
9         B;
10      endchannel outChannel;
11      block mainBlock referenced;
12    endsystem KitchenTimer;
13
14    Block mainBlock;
15
16       signal incrMsg, clearMsg, startMsg, stopMsg, noringMsg, ringMsg;
17
18       signalroute inRoute from env to UIProcess
19          with C, S, M;
20
21       signalroute r1 from UIProcess to CounterProcess
22          with incrMsg, clearMsg, startMsg, stopMsg;
23
24       signalroute outRoute1 from CounterProcess to env
25          with D;
26
27       signalroute outRoute3 from UIProcess to env
28          with B;
29
30       signalroute r2 from UIProcess to RingProcess
31          with noringMsg;
32          from RingProcess to UIProcess with ringMsg;
33
34       signalroute r3 from CounterProcess to RingProcess
35          with ringMsg;
36
37       signalroute outRoute2 from RingProcess to env
38          with B, noringMsg;
39
40    process UIProcess referenced;
41    process CounterProcess referenced;
42    process RingProcess referenced;
43    connect inGate and inRoute;
44    connect outChannel and outRoute1;
45    connect outChannel and outRoute2;
46    connect outChannel and outRoute3;
47
48    endblock mainBlock;
49
50    Process UIProcess;
51      dcl incrValue String := '';
52      start;
53          output B;
54          output clearMsg;
55      grst0:
56          nextstate idle;
57
58      state idle;
58        input M;
60          task{
61                incrValue = ds.get("initCount");
62          }
63          output incrMsg(incrValue);
64          join grst0;
65        input C;
```

```
66          output clearMsg;
67          join grst0;
68        input S;
69          output startMsg;
70        grst1:
71          nextstate running;
72      endstate;
73
74      state running;
75        input S;
76          output stopMsg;
77          join grst0;
78        input ringMsg;
79          nextstate ringing;
80        input M, C;
81          output B;
82          join grst1;
83      endstate;
84
85      state ringing;
86        input S, C, M;
87          output stopMsg;
88          output noringMsg;
89          join grst0;
90      endstate;
91
92    endprocess UIProcess;
93
94
95
96    Process CounterProcess;
97      timer minute := 60.0;
98      dcl min int := 0;
99      dcl minString String := '';
100
101     start ;
102         task {min = 0;}
103         grst4:
104         nextstate idle;
105
106     state idle;
107        input clearMsg;
108           task {
109                min = 0;
110              minString = String.valueOf(min);
111           }
112           output D(minString);
113           join grst4;
114        input incrMsg;
115           task {
116                min = min + Integer.parseInt(value);
117                minString = String.valueOf(min);
118              }
119           output D(minString);
120           join grst4;
121        input startMsg;
122           set(minute);
123           grst5:
124           nextstate counting;
125     endstate;
126
127     state counting;
128        input stopMsg;
129           reset (minute);
130          task {
131                minString = String.valueOf(min);
132              }
133          output D(minString);
```

```
134          join grst4;
135       input minute;
136          decision min = 0;
137             (true):
138                 output ringMsg;
139             (false):
140                task {
141                      min = min - 1;
142                    minString = String.valueOf(min);
143                      try {Thread.sleep(1000);
144                      }catch(InterruptedException exception){}
145                   }
146                 output D(minString);
146                set(minute);
147           enddecision;
148          join grst5;
149    endstate;
150
151  endprocess CounterProcess;
152
153
154  Process RingProcess;
155    timer second := 1.0;
156    start ;
157       grst2:
158       nextstate idle;
158
160    state idle;
161      input ringMsg;
162         output ringMsg;
163         grst3:
164         output B;
165         set(second);
166         task {
167              try {Thread.sleep(100);
168              }catch(InterruptedException exception){}
169             }
170         nextstate analysing;
171    endstate;
172
173    state analysing;
174       input second;
175          join grst3;
176       input noringMsg;
177          reset(second);
178          join grst2;
179    endstate;
180
     endprocess RingProcess;
```

APÊNDICE B – KTClient.java

```
1   import javax.swing.*;
2   import java.awt.*;
3   import java.awt.event.*;
4   import java.util.Vector;
5   import java.io.*;
6   import java.net.*;
7   import com.rgb.intersection.*;
8   import com.rgb.blue.*;
9
10
11
12  public class KTClient extends JFrame implements ActionListener, ItemListener{
13
14    private JRadioButton jrbAdd = new JRadioButton("Add", true);
15    private JRadioButton jrbClear = new JRadioButton("Clear", false);
16    private JRadioButton jrbStartStop = new JRadioButton("StartStop", false);
17    private Label lblValue1 = new Label("Value1:");
18    private TextField txtValue1 = new TextField("1");
19    private ButtonGroup buttonGroup = new ButtonGroup();
20    private Label lblOut = new Label("XML Output:");
21    private Label lblSeconds = new Label("Seconds:");
22
23    private Button btnProduce = new Button("Produce");
24    private Button btnExit = new Button("Exit");
25
26    private Container c = new Container();
27    private TextArea out;
28
29    private Label lblDisplay = new Label("0");
30
31    private BlueClient blue;
32    private int choice = 1;
33
34
35    public void execute(Object ds2){
36      DataStructure ds = (DataStructure) ds2;
37
38      String channel = ds.get("channel");
39      String signal = ds.get("signal");
40      String content = ds.get("content");
41      Vector multContent = new Vector();
42
43
44      if (!ds.get("multContent").equals("notFound")){
45        int count = 1;
46        String labelValues = "value"+count;
47        while (!ds.get(labelValues).equals("notFound")){
48          multContent.addElement(ds.get(labelValues));
49           count++;
50           labelValues = "value"+count;
51        }
52        for (int q=0; q < multContent.size(); q++){
53           System.out.println(multContent.get(q).toString());
54        }
55      }
56      if (channel.equals("outRoute1")){
57        if (signal.equals("D")){
58          lblDisplay.setText(content);
58        }
60      }
61
62      if (channel.equals("outRoute2")){
63        if (signal.equals("B")){
64          java.awt.Toolkit.getDefaultToolkit().beep();
65        }
```

```
66      }
67
68      if (channel.equals("outRoute3")){
69        if (signal.equals("B")){
70          java.awt.Toolkit.getDefaultToolkit().beep();
71        }
72      }
73
74      blue.setFreeForNext();
75    }
76
77    public KTClient(){
78    super("Kitchen Timer Client");
79    blue = new BlueClient(this);
80    doInterface();
81  }
82
83  public KTClient(String registryIp, int registryPort, String serviceId){
84    super("Kitchen Timer Client");
85    blue = new BlueClient(registryIp, registryPort, serviceId, this);
86    doInterface();
87  }
88
89
90  public void doInterface(){
91      String outMessageIncrement = "<message>\n   <channel>\n      inRoute\n";
92      outMessageIncrement = outMessageIncrement +"   </channel>\n   <signal>\n";
93      outMessageIncrement = outMessageIncrement +"M\n</signal>\n<initCount>\n";
94      outMessageIncrement          =          outMessageIncrement          +"1\n
95  </initCount>\n</message>";
96
97      out = new TextArea(outMessageIncrement);
98
99      setSize(387, 475);
100     setLocation(350,200);
101     c = getContentPane();
102     c.setLayout(null);
103
104     c.add(jrbAdd);
105     c.add(jrbClear);
106     c.add(jrbStartStop);
107     c.add(lblValue1);
108     c.add(txtValue1);
109     c.add(lblOut);
110     c.add(out);
111     c.add(btnProduce);
112     c.add(btnExit);
113     c.add(lblSeconds);
114     c.add(lblDisplay);
115     buttonGroup.add(jrbAdd);
116     buttonGroup.add(jrbClear);
117     buttonGroup.add(jrbStartStop);
118
119     lblOut.setBounds(10, 10, 80, 20);
120     jrbAdd.setBounds(100, 10, 80, 20);
121     jrbClear.setBounds(180, 10, 80, 20);
122     jrbStartStop.setBounds(260, 10, 80, 20);
123     lblValue1.setBounds(10, 35, 80, 20);
124     txtValue1.setBounds(100, 35, 50, 20);
125
126     out.setBounds(10, 60, 350, 340);
127     btnProduce.setBounds(10, 410 , 80, 20);
128     btnExit.setBounds(100, 410, 80, 20);
129     lblSeconds.setBounds(190, 410, 80, 20);
130     lblDisplay.setBounds(270, 410, 80, 20);
131
132     btnProduce.addActionListener(this);
133     jrbAdd.addItemListener(this);
```

```
134     jrbClear.addItemListener(this);
135     jrbStartStop.addItemListener(this);
136     btnExit.addActionListener(this);
137
138     show();
139   }
140
141   public void itemStateChanged(ItemEvent e){
142     Object source = e.getSource();
143     String channel = "inRoute";
144     String value1 = "";
145     String signal = "";
146
146     if (source == jrbAdd){
147       signal = "M";
148       value1 = "<initCount>\n" + txtValue1.getText() + "\n</initCount>\n";
149       choice = 1;
150     }
151     if (source == jrbClear){
152       signal = "C";
153       value1 = "";
154       choice = 2;
155     }
156     if (source == jrbStartStop){
157       signal = "S";
158       value1 = "";
158       choice = 3;
160     }
161
162     String outMessage = "<message>\n   <channel>\n      " + channel + "\n";
163     outMessage = outMessage +"   </channel>\n   <signal>\n";
164     outMessage = outMessage +"      " + signal + "\n   </signal>\n";
165     outMessage = outMessage + value1 + "</message>";
166
167     out.setText(outMessage);
168   }
169
170   public void actionPerformed(ActionEvent ae){
171     Object source = ae.getSource();
172     if(source == btnProduce){
173       blue.setXml(out.getText().trim());
174     }
175     if (source == btnExit){
176       blue.systemExit();
177     }
178   }
179
180   public static void main(String[] args){
181     if (args.length==0){
182       new KTClient();
183     }
184     else{
185       new KTClient(args[0], Integer.parseInt(args[1]), args[2]);
186     }
187   }
188 }
```

APÊNDICE C – sdlParserService.sdl

```
1   package util.sdlParserService;
2   import com.rgb.sdl.*;
3
4   System sdlParserService;
5    signal setSDL, filesResult;
6     channel inGate from env to mainBlock with setSDL;
7     endchannel inGate;
8     channel outChannel from mainBlock to env with filesResult;
9     endchannel outChannel;
10    block mainBlock referenced;
11  endsystem KitchenTimer;
12
13  Block mainBlock;
14
15     signalroute inRoute from env to SDLTransformProcess
16        with setSDL;
17
18     signalroute outRoute from SDLTransformProcess to env
19        with filesResult;
20
21  process SDLTransformProcess referenced;
22  connect inGate and inRoute;
23  connect outChannel and outRoute;
24
25  endblock mainBlock;
26
27  Process SDLTransformProcess;
28     dcl sdlContent String := '';
29     dcl sdlFileName String := '';
30     dcl sp sdlParser := null;
31     dcl xmlResult String := '';
32
33     start;
34         startLabel:
35         nextstate idle;
36
37     state idle;
38       input setSDL;
39         task{
40              sp = new sdlParser("remote");
41             sdlFileName = ds.get("sdlFileName");
42             sdlContent = ds.get("sdlContent");
43             sp.setSdl(sdlFileName, sdlContent);
44             xmlResult = sp.getHeader();
45               xmlResult = xmlResult + sp.getErrorMessage();
46             if(sp.gotError()==false){
47                xmlResult = xmlResult + sp.getRemoteWarning();
48                xmlResult = xmlResult + sp.buildFilesXML();
49             }
50
51         }
52         output filesResult(xmlResult);
53         join startLabel;
54     endstate;
55
56   endprocess UIProcess;
```

APÊNDICE D – sdlParserClient.java

```
1   import com.rgb.blue.*;
2   import com.rgb.intersection.*;
3   import java.io.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6   import java.awt.*;
7
8
9   public class sdlParserClient extends JFrame implements ActionListener{
10    private BlueClient blue;
11    private Label lblFileName = new Label("File:");
12    private Label lblSdlText = new Label("sdlFile:");
13    private TextArea taSdlText = new TextArea();
14    private TextField txtFileName = new TextField();
15    private Button btnGetFile = new Button("Load");
16    private Button btnParse = new Button("Parse");
17    private Label lblComment = new Label("Comment:");
18    private TextArea taComment = new TextArea();
19    private Container c = new Container();
20
21    public sdlParserClient(){
22      super("sdlParserClient");
23      doInterface();
24    }
25
26    public sdlParserClient(String fileName){
27      super("sdlParserClient");
28      txtFileName.setText(fileName);
29      doInterface();
30      getFileContent();
31    }
32
33    public void doInterface(){
34      blue = new BlueClient(this);
35      setSize(410, 690);
36      c = getContentPane();
37      c.setLayout(null);
38      c.add(lblFileName);
39      c.add(lblSdlText);
40      c.add(taSdlText);
41      c.add(txtFileName);
42      c.add(btnGetFile);
43      c.add(btnParse);
44      c.add(taComment);
45      c.add(lblComment);
46
47      lblFileName.setBounds(10, 10, 100, 20);
48      txtFileName.setBounds(10, 30, 120, 20);
49      btnGetFile.setBounds(140, 30, 50, 20);
50      btnParse.setBounds(200, 30, 50, 20);
51      lblSdlText.setBounds(10, 60, 80, 20);
52      taSdlText.setBounds(10, 80, 380, 440);
53      lblComment.setBounds(10, 525, 80, 20);
54      taComment.setBounds(10, 545, 380, 100);
55      btnGetFile.addActionListener(this);
56      btnParse.addActionListener(this);
57      show();
58    }
58
60    public void actionPerformed(ActionEvent ae){
61      Object source = ae.getSource();
62      if(source == btnGetFile){
63        getFileContent();
64      }
65      if (source == btnParse){
```

```
66        String outMessage = "<message><channel>inRoute</channel>";
67        outMessage = outMessage + "<signal>setSDL</signal>";
68        outMessage  =  outMessage  +  "<sdlFileName>"  +  txtFileName.getText()  +
69  "</sdlFileName>";
70        outMessage  =  outMessage  +  "<sdlContent>"  +  taSdlText.getText()  +
71  "</sdlContent>";
72        outMessage = outMessage + "</message>";
73        blue.setXml(outMessage.trim());
74      }
75    }
76
77
78    public void getFileContent(){
79      String sdlContent = "";
80      String text = "";
81      try{
82        FileInputStream fis = new FileInputStream(txtFileName.getText());
83        InputStreamReader isr = new InputStreamReader(fis);
84        BufferedReader in  = new BufferedReader(isr);
85        while ((text = in.readLine())!=null)
86          sdlContent = sdlContent + text + "\n";
87        taSdlText.setText(sdlContent);
88      }catch (Exception e){}
89    }
90
91    public void execute(Object ds2){
92      DataStructure ds = (DataStructure) ds2;
93      String channel = ds.get("channel");
94      String signal = ds.get("signal");
95      String warning = ds.get("warningMessage");
96      String error = ds.get("errorMessage");
97      String header = ds.get("header");
98
99      taComment.append("     " + header + "\n\n");
100     taComment.append(error + "\n");
101     if (channel.equals("outRoute")){
102       if (signal.equals("filesResult")){
103         int n = 1;
104          while (!ds.get("fileName"+n).equals("notFound")){
105           String fileName = ds.get("fileName"+n);
106           String diretorie = ds.get("diretorie"+n);
107           String content = ds.get("content"+n);
108           buildFile(fileName, diretorie, content);
109            taComment.append("   - " + fileName + ".java: System Generated\n");
110           n++;
111         }
112       }
113     }
114
115     if (!warning.equals("notFound")){
116       taComment.append("\n" + warning);
117     }
118     taComment.append("\n-----------------------------------------------\n");
119     blue.setFreeForNext();
120   }
121
122   public void buildFile(String fileName, String diretorie, String content){
123     File dir = new File(diretorie);
124     dir.mkdir();
125     File file = new File(diretorie + "\\" + fileName + ".java");
126     try{
127       RandomAccessFile r = new RandomAccessFile(file, "rw");
128       r.seek(0);
129       r.setLength(0);
130       r.writeBytes(content);
131       r.close();
132     }
133     catch (Exception e){System.out.println(e);}
```

```
134   }
135
136   public static void main(String[] args){
137     if (args.length==0){
138       new sdlParserClient();
139     }
140     else{
141       new sdlParserClient(args[0]);
142     }
143   }
144 }
```

APÊNDICE E – Comandos SQL (MySQL)

Script de criação do Banco de Dados **registry**:

```
1   CREATE DATABASE registry;

2   USE registry;

3   CREATE TABLE ClientInstances (
4          instance         VARCHAR(20),
5          serviceID        VARCHAR(20)
6          changeIP         VARCHAR(20),
7          changePort       VARCHAR(5),
8          serviceInstance  VARCHAR(20),
9   );

10  ALTER TABLE ClientInstances
11         ADD  ( UNIQUE (instance));

12  CREATE TABLE ServiceInstances (
13         serviceIP            VARCHAR(15),
14         servicePort          VARCHAR(5),
15         serviceID            VARCHAR(20),
16         serviceInstance      VARCHAR(20),
17         connectionsCount      VARCHAR(20)
18  );

19  ALTER TABLE ServiceInstances
20         ADD  ( UNIQUE (serviceInstance));

21  CREATE TABLE Services (
22         serviceID            VARCHAR(20),
23         Name                 VARCHAR(40),
24         Descripton           VARCHAR(255)
25  );

26  ALTER TABLE Services
27         ADD  ( UNIQUE (serviceID)) ;

28  ALTER TABLE ClientInstances
29         ADD  ( FOREIGN KEY (serviceID)
                              REFERENCES Services ) ;
30  ALTER TABLE ClientInstances
31         ADD  ( FOREIGN KEY (serviceInstance)
                              REFERENCES ServiceInstances ) ;
32  ALTER TABLE ServiceInstances
33         ADD  ( FOREIGN KEY (serviceID)
34                              REFERENCES Services ) ;
```

Os comandos SQL, especificados abaixo, são utilizados pelo Registro de serviços, para manipular o Banco de Dados **Registry**. Esses são referentes aos diagramas de seqüência, encontrados no **Capítulo 3**. As variáveis sublinhadas, fazem referencia as variáveis especificadas nos diagramas de seqüência.

SQL0
```
select serviceID
from Services
```

SQL1
```
insert into Services
values (serviceID, name, description)
```

SQL2
```
select serviceID
from Services
where serviceID = :serviceID
```

SQL3
```
insert into ServiceInstances
values (:ip, :port, :serviceID, :instância, 0)
```

SQL4
```
select serviceInstance, serviceIP, servicePort
from ServiceInstances
where serviceID = :serviceID
```

SQL5
```
update ServiceInstances
set connectionsCount = connectionsCount + 1
where instance = :serviceInstance
```

SQL6
```
insert into ClientInstances
values(:instância, :serviceID, :changeIP, :changePort, serviceInstance)
```

SQL7
```
select serviceID
from ServiceInstances
where instance = :instância
```

SQL8
```
delete
from ServiceInstances
where instance = :instância
```

SQL9
```
select *
from ServiceInstances
order by connectionsCount
```

SQL10
```
select *
from ClientInstances
where serviceInstances = :instância
```

SQL11
```
update ClientInstances
set serviceInstance = :novaInstância.instância
where serviceInstance = :instância
```

SQL12
```
select serviceInstance
from ClientInstance
where serviceInstance = :instância
```

SQL13
```
update ServiceInstances
set connectionsCount = connectionsCount + 1
where instances = :instância
```

SQL14
```
delete
from ClientInstance
where serviceInstance = :instância
```

APÊNDICE F – Diagramas de Classe UML

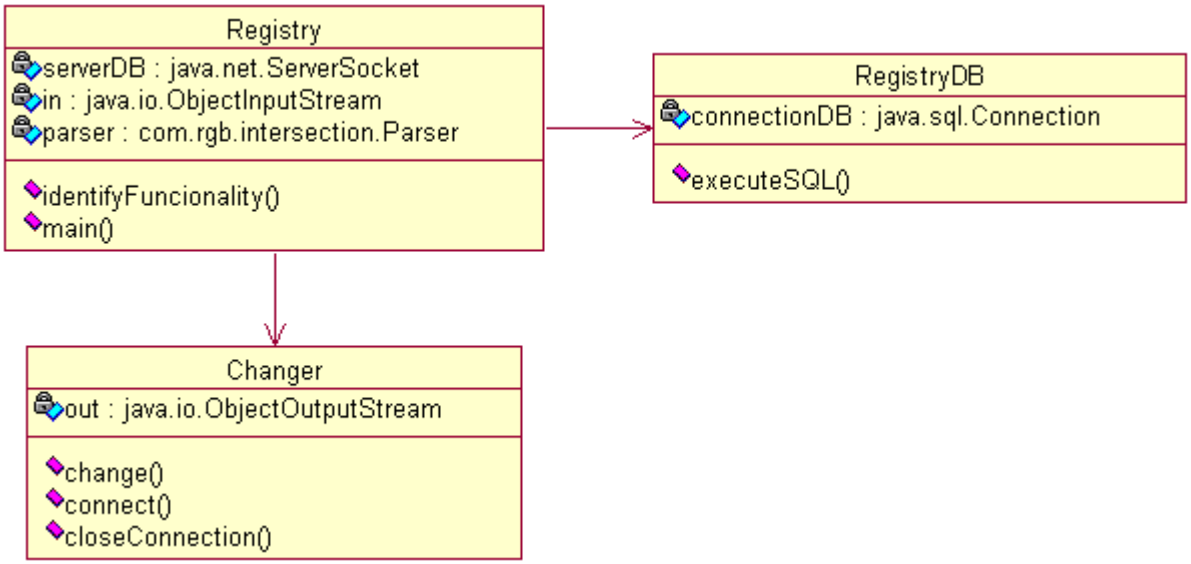

Diagrama de Classes UML – Pacote **com.rgb.red**

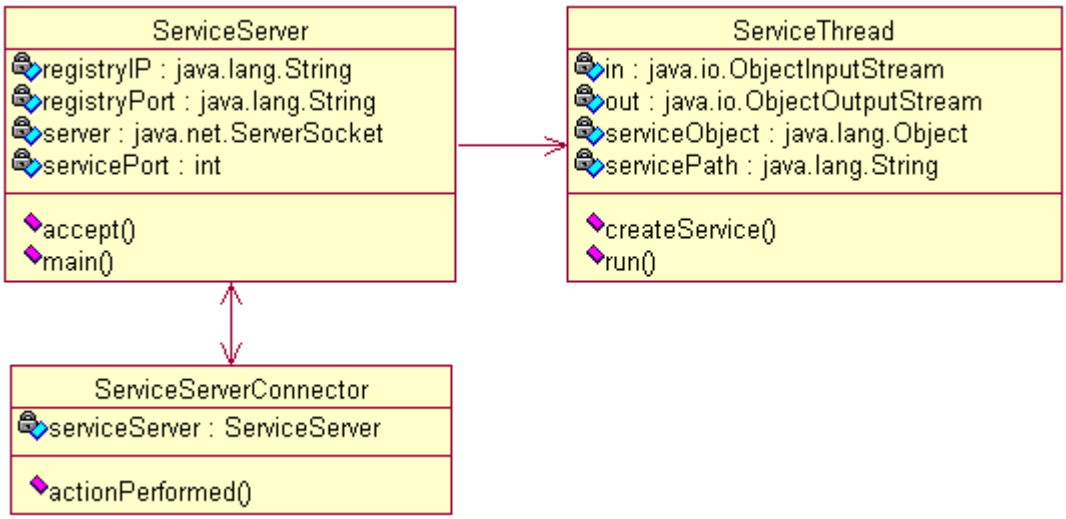

Diagrama de Classes UML – Pacote **com.rgb.green**

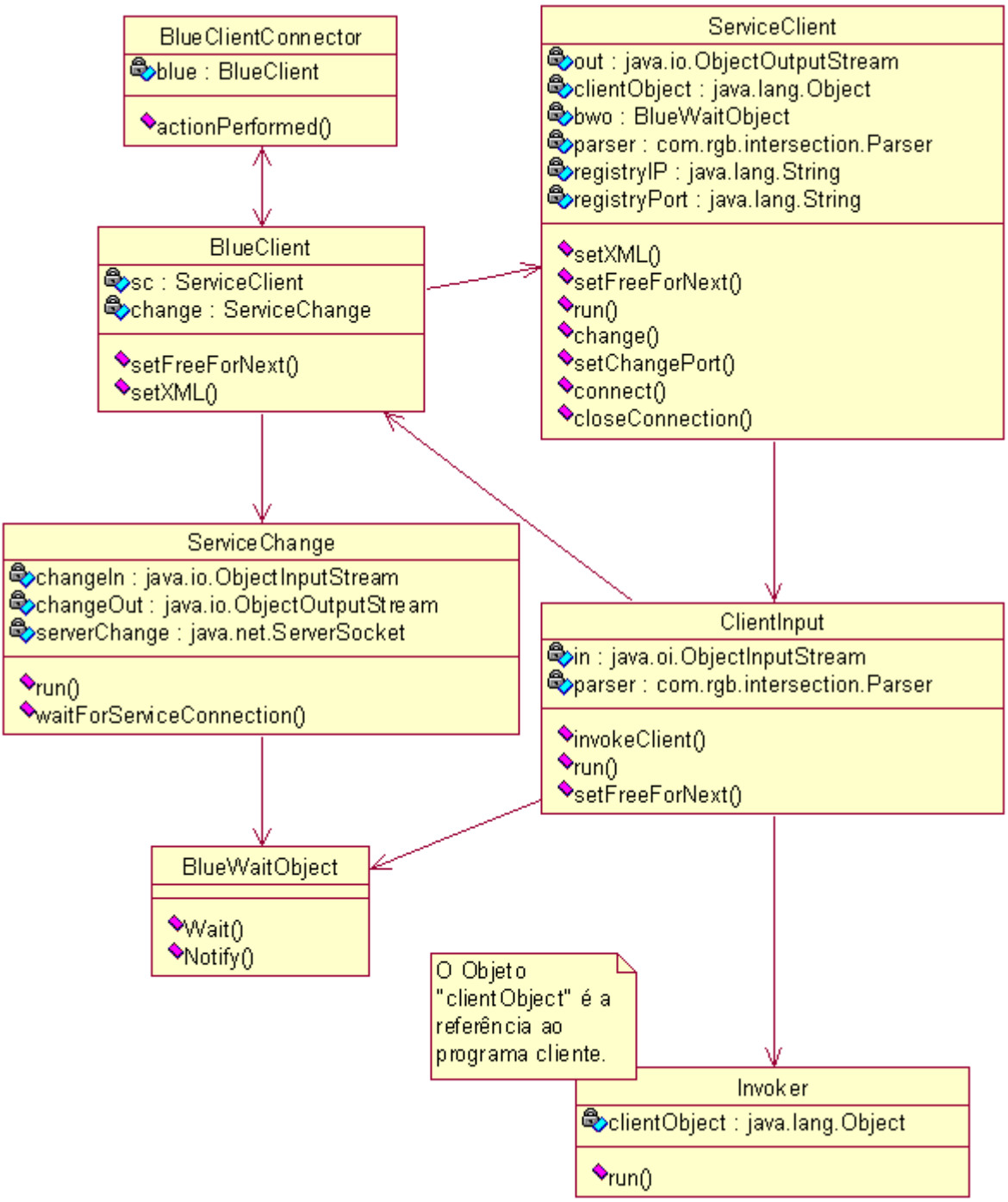

Diagrama de Classes UML – Pacote **com.rgb.blue**

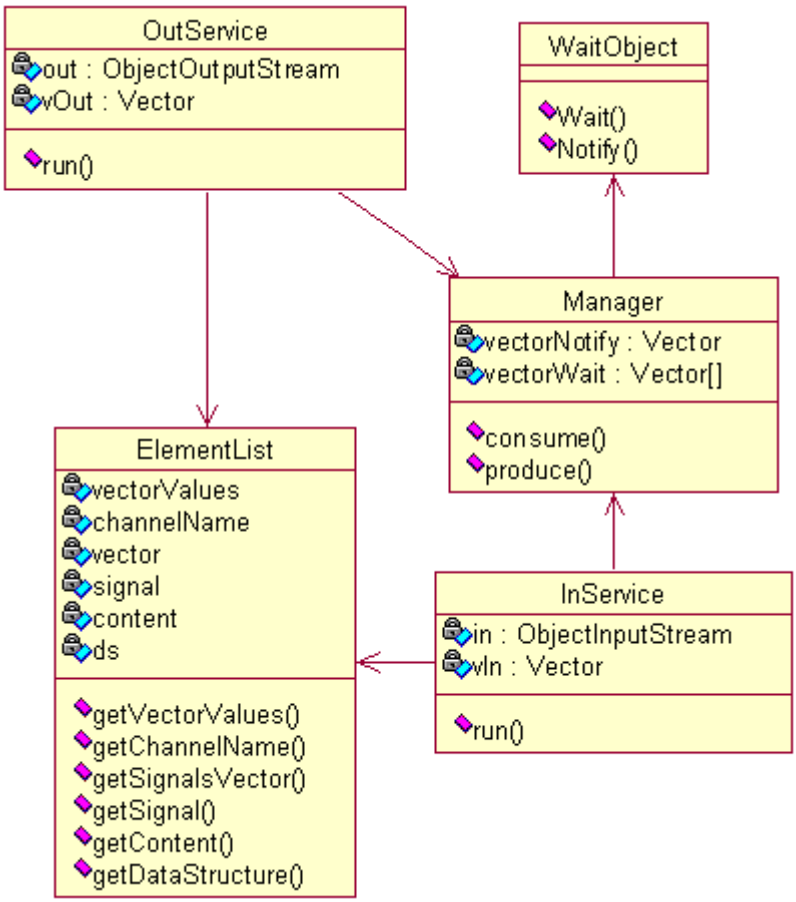

Diagrama de Classes UML – Pacote **com.rgb.green.service**

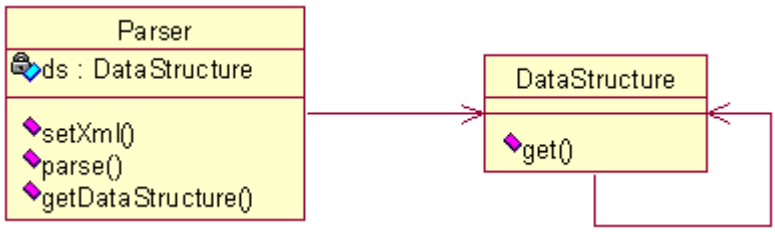

Diagrama de Classes UML – Pacote **com.rgb.intersection**

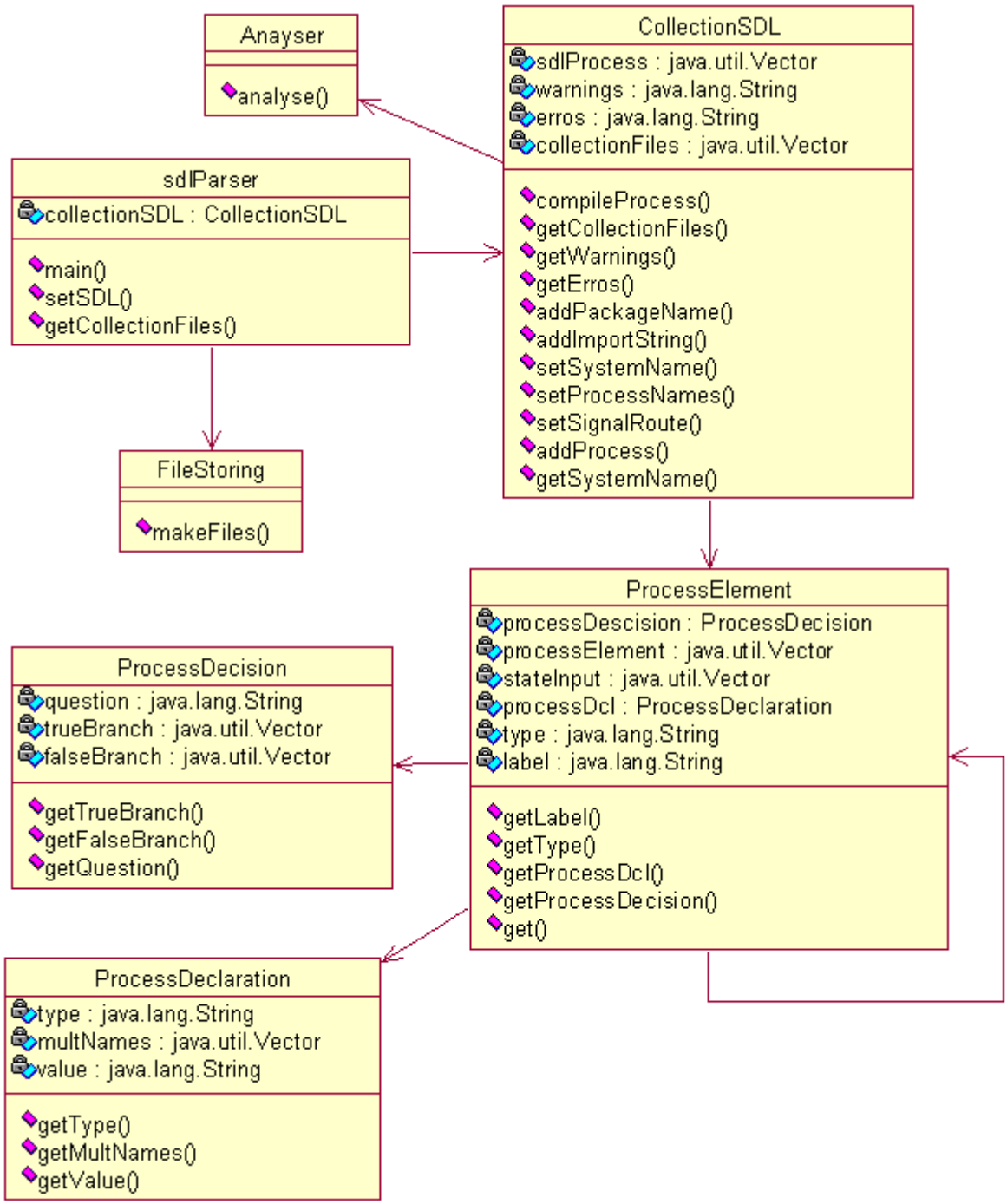

Diagrama de Classes UML – Pacote **com.rgb.sdl**

APÊNDICE G – **Serviço KitchenTimer (arquivos Java).**

KitchenTimer.java

```
1   package util.KitchenTimer;
2
3   import com.rgb.sdl.sdlParser;
4
5   import com.rgb.green.service.*; // Pacote que contém as classes que são
6                                   // utilizadas para a instanciação do serviço.
7   import java.io.*;
8   import java.util.*;
9   import java.net.*;
10
11  public class KitchenTimer extends Thread{
12
13     private Manager manager; // Gerenciador
14     private OutService os; // canal de saída
15     private InService is; // canal de entrada
16     private UIProcess UIProcessInstance; // instância do processo UIProcess
17
18     private CounterProcess CounterProcessInstance; // instância do Processo
19                                                    // CounterProcess.
20
21     private RingProcess RingProcessInstance; // instância do Processo
22                                              // RingProcess
23
24     public KitchenTimer(){
25     }
26
27     public void setStart(Object oosObject, Object oisObject){
28
29        ObjectOutputStream oos = (ObjectOutputStream) oosObject;
30        ObjectInputStream ois = (ObjectInputStream) oisObject;
31
32        manager = new Manager();
33
34        // Cada instância de processo recebe uma referência ao Gerenciador.
35        UIProcessInstance = new UIProcess(manager);
36        CounterProcessInstance = new CounterProcess(manager);
37        RingProcessInstance = new RingProcess(manager);
38
39
40
41        // Crianado o Vetor de entrada para descrever as entradas do Serviço.
42        Vector vInService = new Vector();
43        vInService.addElement(new ElementList("inRoute", "C"));
44        vInService.addElement(new ElementList("inRoute", "S"));
45        vInService.addElement(new ElementList("inRoute", "M"));
46        is = new InService(manager, ois, vInService);
47
48
49        // Criando o Vetor de saída para descrever as saídas do Serviço.
50        Vector vOutService = new Vector();
51        Vector vSignals = new Vector();
52
53        vSignals.removeAllElements();
54        vSignals.addElement("D");
55        vOutService.addElement(new ElementList("outRoute1", vSignals));
56
57        vSignals.removeAllElements();
58        vSignals.addElement("B");
58        vOutService.addElement(new ElementList("outRoute3", vSignals));
60
61        vSignals.removeAllElements();
62        vSignals.addElement("B");
```

```
63          vSignals.addElement("noringMsg");
64          vOutService.addElement(new ElementList("outRoute2", vSignals));
65          os = new OutService(manager, oos, vOutService);
66
67          // inicializando os Processos
68          os.start();
69          is.start();
70          UIProcessInstance.start();
71          CounterProcessInstance.start();
72          RingProcessInstance.start();
73       }
74
75       public void run(){
76       }
77
78       public static void main(String[] args){
79          System.out.println("\n-------------------------------------------");
80          System.out.println("RGB Java Environment Testing The Service:");
81          System.out.println(" - KitchenTimer can be invoked by the Service
82                               Provider...");
83          System.out.println("-------------------------------------------");
84       }
85    }
```

UIProcess.java

```
1    package util.KitchenTimer;
2
3    import com.rgb.sdl.sdlParser;
4    import com.rgb.green.service.*;
5    import com.rgb.intersection.*;
6    import java.io.*;
7    import java.util.*;
8
9    public class UIProcess extends Thread{
10
11      private String incrValue = ""; // dcl incrValue String := ‘’;
12      private Manager manager; // referência ao Gerenciador
13
14      public UIProcess(Manager manager){
15         this.manager = manager;
16      }
17
18      public UIProcess(){
19      }
20      public void run(){
21         do{
22            String label = Start();
23         }while(true);
24      }
25      public String Start(){ // start;
26         String label = "";
27         // output B;
28         manager.produce(new ElementList("outRoute3", "B"), "UIProcess");
29         // output clearMsg
30         manager.produce(new ElementList("r1", "clearMsg"), "UIProcess");
31         label = labelgrst0(); // grst0;
32         return label;
33      }
34
35      // os métodos labelgrst0() e labelgrst0Continue() implementam a instrução “grst0;”
36      public String labelgrst0(){
37         String label = "";
38         do{
39            label = labelgrst0Continue();
```

```
40        }while(label.equals("grst0"));
41        return label;
42     }
43     public String labelgrst0Continue(){
44        String label = "";
45           label = stateidle(); //nextstate idle;
46        return label;
47     }
48     public String stateidle(){ // state idle;
49        String label = "";
50        Vector v = new Vector();
51        Vector vectorLine = new Vector();
52
53        vectorLine.removeAllElements();
54
55        v.removeAllElements();
56        // input M;
57        v.addElement("M");
58        vectorLine.addElement(new ElementList("inRoute", v));
58
60        v.removeAllElements();
61        // input C;
62        v.addElement("C");
63        vectorLine.addElement(new ElementList("inRoute", v));
64
65        v.removeAllElements();
66        // input S;
67        v.addElement("S");
68        vectorLine.addElement(new ElementList("inRoute", v));
69
70        // Intenção de Consumo das entradas acima configuradas.
71        Vector inputs = manager.consume(vectorLine, "UIProcess");
72        String value = ((ElementList) inputs.get(1)).getContent();
73        Vector values = ((ElementList) inputs.get(1)).getVectorValues();
74        DataStructure ds = ((ElementList) inputs.get(1)).getDataStructure();
75        int branch = Integer.parseInt(((String)inputs.get(0)));
76
77        switch(branch){
78        case 0:{ // caso a entrada for M
79           incrValue= ds.get("initCount"); // conteúdo de uma Task
80           // output incrMsg(incrValue);
81     manager.produce(new ElementList("r1", "incrMsg", incrValue), "UIProcess");
82           label = "grst0"; // join grst0;
83        }break;
84        case 1:{ // caso a entrada for C
85           // output clearMsg;
86           manager.produce(new ElementList("r1", "clearMsg"), "UIProcess");
87           label = "grst0"; // join grst0;
88        }break;
89        case 2:{ // caso a entrada S
90           // output startMsg;
91           manager.produce(new ElementList("r1", "startMsg"), "UIProcess");
92           label = labelgrst1(); // grst1;
93        }
94        }
95     return label;
96     }
97
98     // os métodos labelgrst1() e labelgrst1Continue() implementam a instrução “grst1;”
99     public String labelgrst1(){
100       String label = "";
101       do{
102          label = labelgrst1Continue();
103       }while(label.equals("grst1"));
104       return label;
105    }
106    public String labelgrst1Continue(){
107       String label = "";
```

```
108          label = staterunning(); // nextstate running;
109       return label;
110    }
111    public String staterunning(){ // state running;
112       String label = "";
113       Vector v = new Vector();
114       Vector vectorLine = new Vector();
115
116       vectorLine.removeAllElements();
117
118       v.removeAllElements();
119       // input S;
120       v.addElement("S");
121       vectorLine.addElement(new ElementList("inRoute", v));
122
123       v.removeAllElements();
124       // input ringMsg;
125       v.addElement("ringMsg");
126       vectorLine.addElement(new ElementList("r2", v));
127
128       v.removeAllElements();
129       // input M, C;
130       v.addElement("M");
131       v.addElement("C");
132       vectorLine.addElement(new ElementList("inRoute", v));
133
134       // Intenção de Consumo das entradas acima configuradas.
135       Vector inputs = manager.consume(vectorLine, "UIProcess");
136       String value = ((ElementList) inputs.get(1)).getContent();
137       Vector values = ((ElementList) inputs.get(1)).getVectorValues();
138       DataStructure ds = ((ElementList) inputs.get(1)).getDataStructure();
139       int branch = Integer.parseInt(((String)inputs.get(0)));
140
141       switch(branch){
142       case 0:{ // caso a entrada for S
143          // output stopMsg;
144          manager.produce(new ElementList("r1", "stopMsg"), "UIProcess");
145          label = "grst0"; // join grst0;
146
146       }break;
147       case 1:{ // caso a entrada for ringMsg
148          label = stateringing(); // nextstate ringing;
149
150       }break;
151       case 2:{ //caso a entrada for M ou C
152          // output B
153          manager.produce(new ElementList("outRoute3", "B"), "UIProcess");
154          label = "grst1"; // join grst1;
155
156       }
157       }
158    return label;
158    }
160    public String stateringing(){ // state ringing;
161       String label = "";
162       Vector v = new Vector();
163       Vector vectorLine = new Vector();
164
165       vectorLine.removeAllElements();
166
167       v.removeAllElements();
168       // input S, C, M;
169       v.addElement("S");
170       v.addElement("C");
171       v.addElement("M");
172       vectorLine.addElement(new ElementList("inRoute", v));
173
174       // Intenção de Consumo das entradas acima configuradas.
```

```
175       Vector inputs = manager.consume(vectorLine, "UIProcess");
176       String value = ((ElementList) inputs.get(1)).getContent();
177       Vector values = ((ElementList) inputs.get(1)).getVectorValues();
178       DataStructure ds = ((ElementList) inputs.get(1)).getDataStructure();
179       int branch = Integer.parseInt(((String)inputs.get(0)));
180
181       switch(branch){
182       case 0:{ // caso a entrada for C, M ou S
183          // output stopMsg;
184          manager.produce(new ElementList("r1", "stopMsg"), "UIProcess");
185          // output noringMsg;
186          manager.produce(new ElementList("r2", "noringMsg"), "UIProcess");
187          label = "grst0"; // join grst0;
188
189       }
190       }
191    return label;
192    }
193 }
```

CounterProcess.java

```
1    package util.KitchenTimer;
2
3    import com.rgb.sdl.sdlParser;
4    import com.rgb.green.service.*;
5    import com.rgb.intersection.*;
6    import java.io.*;
7    import java.util.*;
8
9    public class CounterProcess extends Thread{
10
11      private double minute = 60.0; // timer minute := 60;
12      private int min = 0; // dcl min int := 0;
13      private String minString = ""; // dcl minString String := ‘’;
14      private Manager manager; // Gerenciador
15
16      public CounterProcess(Manager manager){
17         this.manager = manager;
18      }
19
20      public CounterProcess(){
21      }
22      public void run(){
23         do{
24            String label = Start();
25         }while(true);
26      }
27      public String Start(){ // start;
28         String label = "";
29         min= 0; // task {min = 0};
30         label = labelgrst4(); // grst4;
31         return label;
32      }
33
34      // os métodos labelgrst4() e labelgrst4Continue()
35      // implementam a funcionalidade da instrução “grst4;”
36      public String labelgrst4(){
37         String label = "";
38         do{
39            label = labelgrst4Continue();
40         }while(label.equals("grst4"));
41         return label;
42      }
43      public String labelgrst4Continue(){
```

```
44        String label = "";
45           label = stateidle(); // nextstate idle;
46        return label;
47     }
48     public String stateidle(){ // state idle;
49        String label = "";
50        Vector v = new Vector();
51        Vector vectorLine = new Vector();
52
53        vectorLine.removeAllElements();
54
55        v.removeAllElements();
56        // input clearMsg;
57        v.addElement("clearMsg");
58        vectorLine.addElement(new ElementList("r1", v));
58
60        v.removeAllElements();
61        // input incrMsg;
62        v.addElement("incrMsg");
63        vectorLine.addElement(new ElementList("r1", v));
64
65        v.removeAllElements();
66        // input startMsg;
67        v.addElement("startMsg");
68        vectorLine.addElement(new ElementList("r1", v));
69
70        // Intenção de consumo das entradas acima;
71        Vector inputs = manager.consume(vectorLine, "CounterProcess");
72        String value = ((ElementList) inputs.get(1)).getContent();
73        Vector values = ((ElementList) inputs.get(1)).getVectorValues();
74        DataStructure ds = ((ElementList) inputs.get(1)).getDataStructure();
75        int branch = Integer.parseInt(((String)inputs.get(0)));
76
77        switch(branch){
78        case 0:{ // caso a entrada for clearMsg
79
80           // task {
81           //        min = 0;
82           //        minString = String.valueOf(min);
83           // }
84           min= 0;
85           minString= String.valueOf(min);
86           // output D(minString);
87           manager.produce(new ElementList("outRoute1", "D", minString),
88                                                    "CounterProcess");
89           label = "grst4"; // join grst4;
90
91        }break;
92        case 1:{ // caso a entrada foi incrMsg
93           // task {
94           // min= min+ Integer.parseInt(value);
95           // minString= String.valueOf(min);
96           // }
97           min= min+ Integer.parseInt(value);
98           minString= String.valueOf(min);
99           // output D(minString);
100          manager.produce(new ElementList("outRoute1", "D", minString),
101                                                   "CounterProcess");
102          label = "grst4"; // join grst4;
103       }break;
104       case 2:{ // caso a entrada foi startMsg
105          // set (minute);
106          manager.set(new ElementList("CounterProcessInternalChannel", "minute"),
107                                                              "CounterProcess");
108          label = labelgrst5(); // grst5;
109
110       }
111       }
```

```
112    return label;
113    }
114
115    // os métodos labelgrst5() e labelgrst5Continue()
116    // implementam a funcionalidade da instrução “grst5;”
117    public String labelgrst5(){
118       String label = "";
119       do{
120          label = labelgrst5Continue();
121       }while(label.equals("grst5"));
122       return label;
123    }
124    public String labelgrst5Continue(){
125       String label = "";
126          label = statecounting(); // nextstate counting;
127       return label;
128    }
129    public String statecounting(){
130       String label = "";
131       Vector v = new Vector();
132       Vector vectorLine = new Vector();
133
134       vectorLine.removeAllElements();
135
136       v.removeAllElements();
137       // input stopMsg;
138       v.addElement("stopMsg");
139       vectorLine.addElement(new ElementList("r1", v));
140
141       v.removeAllElements();
142       // input minute;
143       v.addElement("minute");
144       vectorLine.addElement(new ElementList("CounterProcessInternalChannel", v));
145
146       // Intenção de Consumo das entradas acima configuradas.
146       Vector inputs = manager.consume(vectorLine, "CounterProcess");
147       String value = ((ElementList) inputs.get(1)).getContent();
148       Vector values = ((ElementList) inputs.get(1)).getVectorValues();
149       DataStructure ds = ((ElementList) inputs.get(1)).getDataStructure();
150       int branch = Integer.parseInt(((String)inputs.get(0)));
151
152       switch(branch){
153       case 0:{ // caso a entrada for stopMsg
154          // reset (minute);
155          manager.reset(new ElementList("CounterProcessInternalChannel", "minute"),
156                                                                  "CounterProcess");
157          // task {
158          //     minString = String.valueOf(min);
158          // }
160          minString= String.valueOf(min);
161          // output D(minString);
162          manager.produce(new ElementList("outRoute1", "D", minString),
163                                                  "CounterProcess");
164          label = "grst4"; // join grst4;
165
166       }break;
167       case 1:{ // caso a entrada for minute
168          // decision min = 0;
169          if (min==0){
170          // (true):
171          // output ringMsg;
172          manager.produce(new ElementList("r3", "ringMsg"), "CounterProcess");
173          }
174          else{
175          // (false):
176          // task {
177                   min = min - 1;
178                   minString = String.valueOf(min);
```

```
179                     try {Thread.sleep(1000);
180                     }catch(InterruptedException exception){}
181             // }
182             min= min- 1;
183             minString= String.valueOf(min);
184             try{Thread.sleep(1000);
185             }catch(InterruptedException exception){}
186             // output D(minString);
187             manager.produce(new ElementList("outRoute1", "D", minString),
188                                                     "CounterProcess");
189             // set (minute)
190             manager.set(new ElementList("CounterProcessInternalChannel", "minute"),
191                                                             "CounterProcess");
192
193             }
194             label = "grst5"; // join grst5;
195
196          }}
197       return label;
198       }
199    }
```

RingProcess.java

```
1    package util.KitchenTimer;
2
3    import com.rgb.sdl.sdlParser;
4    import com.rgb.green.service.*;
5    import com.rgb.intersection.*;
6    import java.io.*;
7    import java.util.*;
8
9    public class RingProcess extends Thread{
10
11      private double second = 1.0; // timer second := 1.0;
12      private Manager manager; // Gerenciador
13
14      public RingProcess(Manager manager){
15         this.manager = manager;
16      }
17
18      public RingProcess(){
19      }
20      public void run(){
21         do{
22            String label = Start();
23         }while(true);
24      }
25      public String Start(){ // start;
26         String label = "";
27         label = labelgrst2(); // grst2;
28         return label;
29      }
30
31      // os métodos labelgrst2() e labelgrst2Continue() implementam a instrução “grst2;”
32      public String labelgrst2(){
33         String label = "";
34         do{
35            label = labelgrst2Continue();
36         }while(label.equals("grst2"));
37         return label;
38      }
39      public String labelgrst2Continue(){
40         String label = "";
41            label = stateidle(); // nextstate idle;
```

```
42        return label;
43     }
44     public String stateidle(){ // state idle;
45        String label = "";
46        Vector v = new Vector();
47        Vector vectorLine = new Vector();
48
49        vectorLine.removeAllElements();
50
51        v.removeAllElements();
52        // input ringMsg;
53        v.addElement("ringMsg");
54        vectorLine.addElement(new ElementList("r3", v));
55
56        // Intenção de Consumo das entradas acima configuradas.
57        Vector inputs = manager.consume(vectorLine, "RingProcess");
58        String value = ((ElementList) inputs.get(1)).getContent();
58        Vector values = ((ElementList) inputs.get(1)).getVectorValues();
60        DataStructure ds = ((ElementList) inputs.get(1)).getDataStructure();
61        int branch = Integer.parseInt(((String)inputs.get(0)));
62
63        switch(branch){
64        case 0:{ // caso a entrada for ringMsg
65           // output ringMsg;
66           manager.produce(new ElementList("r2", "ringMsg"), "RingProcess");
67           label = labelgrst3(); // grst3;
68        }
69        }
70     return label;
71     }
72
73     // os métodos labelgrst3() e labelgrst3Continue() implementam a instrução “grst3;”
74     public String labelgrst3(){
75        String label = "";
76        do{
77           label = labelgrst3Continue();
78        }while(label.equals("grst3"));
79        return label;
80     }
81     public String labelgrst3Continue(){
82        String label = "";
83        // output B;
84        manager.produce(new ElementList("outRoute2", "B"), "RingProcess");
85        // set (second);
86        manager.set(new ElementList("RingProcessInternalChannel", "second"),
87                                                              "RingProcess");
88        // task {
89        //       try{
90        //           Thread.sleep(100);
91        //       }catch(InterruptedException exception){}
92        // }
93        try{Thread.sleep(100);}catch(InterruptedException exception){}
94
95        // nextstate analysing;
96        label = stateanalysing();
97        return label;
98     }
99     public String stateanalysing(){ // state Analysing;
100       String label = "";
101       Vector v = new Vector();
102       Vector vectorLine = new Vector();
103
104       vectorLine.removeAllElements();
105
106       v.removeAllElements();
107       // input second;
108       v.addElement("second");
109       vectorLine.addElement(new ElementList("RingProcessInternalChannel", v));
```

```
110
111       v.removeAllElements();
112       // input noringMsg
113       v.addElement("noringMsg");
114       vectorLine.addElement(new ElementList("r2", v));
115
116       // Intenção de Consumo das entradas acima configuradas.
117       Vector inputs = manager.consume(vectorLine, "RingProcess");
118       String value = ((ElementList) inputs.get(1)).getContent();
119       Vector values = ((ElementList) inputs.get(1)).getVectorValues();
120       DataStructure ds = ((ElementList) inputs.get(1)).getDataStructure();
121       int branch = Integer.parseInt(((String)inputs.get(0)));
122
123       switch(branch){
124       case 0:{ // caso a entrada for second
125          label = "grst3"; // join grst3;
126       }break;
127       case 1:{
128          manager.reset(new ElementList("RingProcessInternalChannel", "second"),
129                                                                    "RingProcess");
130          label = "grst2"; // join grst2;
131       }}
132    return label;
133    }}
```