UNIVERSIDADE DO VALE DO RIO DOS SINOS
CIÊNCIAS EXATAS E TECNOLÓGICAS,
PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM
COMPUTAÇÃO APLICADA - PIPCA

Um Modelo de Programação Orientado ao Desenvolvimento de Sistemas Ubíquos

por

ALEX SANDRO GARZÃO

Dissertação submetida à avaliação
como requisito parcial para a obtenção do grau de
Mestre em Computação Aplicada

Prof Dr. Jorge Luis Victória Barbosa
Orientador

São Leopoldo, abril de 2010.

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

G245m Garzão, Alex Sandro

Um Modelo de Programação Orientado ao Desenvolvimento de Sistemas Ubíquos / Alex Sandro Garzão. — 2010.

237 f.: il. ; 30cm.

Dissertação (mestrado) — Universidade do Vale do Rio dos Sinos, Ciências Exatas e Tecnológicas, Programa Interdisciplinar de Pós-Graduação em Computação Aplicada - PIPCA, 2010.

"Orientador: Prof. Dr. Jorge Luis Victória Barbosa."

1. Computação Ubíqua. 2. Computação Móvel. 3. Linguagens de Programação. 4. Paradigmas de Programação. 5. Modelos de Programação. I. Título.

CDU 004.75.057.5

Agradecimentos

Aqui termina mais uma etapa da minha vida. Foram tempos onde conciliar o mestrado, família e trabalho, não foi uma tarefa fácil. Incontáveis horas elaborando idéias, arquiteturas, discussões intermináveis que, de tempos em tempos, entravam em pauta novamente. E, como não poderia deixar de ser, muita implementação. Mas, como tudo nesta vida, se vem de graça, não tem graça. E no final das contas, só posso concluir que valeu (e muito) todo este esforço.

Bom, em primeiro lugar, eu agradeço a Deus por tudo. Sem ele, nada seria possível...

Agradeço a minha esposa Graciele por todo o amor, carinho e compreensão que teve comigo, principalmente neste último ano onde a minha ausência foi ainda maior. Sei que nunca lhe agradecerei o suficiente...

Agradeço ao meu filho Gabriel, este anjinho que Deus colocou na minha vida, por alegrar ainda mais os meus dias. Um sorriso neste rostinho lindo faz qualquer problema desaparecer...

Ao meu orientador, professor Jorge Luis Victória Barbosa, por sua infinita paciência e compreensão. Sem seu auxílio e dedicação, certamente eu não teria iniciado (e pouco menos, terminado) este trabalho. Eu gostaria de lhe agradecer também pela oportunidade de, assim como no TCC, realizar um trabalho do qual me orgulho. Meu sincero muito obrigado...

Aos colegas Carlos André Barbara da Silva (vulgo Candrebs), Tiago Rizzetti, Jezer e Solon, pelas inúmeras trocas de idéias.

Por fim, meu muito obrigado a todos que participaram deste período da minha vida e que, direta ou indiretamente, auxiliaram na realização deste trabalho. Apesar de não citados um a um, agradeço a todos, de coração.

Dedico este trabalho à memória de meu pai e avós.

"Programs must be written for people to read, and only incidentally for machines to execute."

(*Sussman*)

"The best way to predict the future is to invent it."

(*Alan Kay*)

"Imaginação é mais importante que inteligência."

(*Albert Einstein - Físico*)

"Há dois tipos de pessoas: as que fazem as coisas, e as que dizem que fizeram as coisas. Tente ficar no primeiro tipo. Há menos competição."

(*Indira Ghandi*)

"O pessimista queixa-se do vento. O otimista espera que ele mude. O realista ajusta as velas."

(*Willian George Ward (1812-1992) - Teólogo inglês.*)

"É impossível para um homem aprender aquilo que ele acha que já sabe."

(*Epíteto, filósofo grego, 55 - 135.*)

"Se você tem uma maça e eu tenho uma maça, e nós trocarmos maças, então você e eu teremos cada um uma maça. Mas se você tem uma idéia e eu tenho uma idéia, e nós trocarmos estas idéias, então cada um de nós terá duas idéias."

(*George Bernard Shaw*)

"É melhor tentar e falhar que ocupar-se em ver a vida passar."

(*Martin Luther King*)

Resumo

O presente trabalho propõe o *Ubiquitous Oriented Programming* (abreviadamente UOP), um modelo de programação orientado ao desenvolvimento de sistemas ubíquos. UOP utiliza os conceitos de Serviços e Orientação a Objetos, integrando-os com os requisitos de aplicações ubíquas como contexto, sensibilidade ao contexto, adaptação ao contexto, mobilidade de código e concorrência. Uma linguagem de programação (UbiLanguage) implementa os conceitos deste novo modelo de programação, provendo assim suporte ao desenvolvimento de sistemas ubíquos. O ambiente de desenvolvimento é composto por um compilador (UbiCompiler) que traduz código escrito em UbiLanguage para *bytecode*. A plataforma de execução é composta por uma máquina virtual ubíqua (UbiVM) que suporta a execução deste *bytecode*.

Palavras-chave: Computação Ubíqua, Computação Móvel, Linguagens de Programação, Paradigmas de Programação, Modelos de Programação.

TITLE: "A Programming Model oriented to Development of Ubiquitous Systems"

Abstract

This work presents the Ubiquitous Oriented Programming (UOP in short), a programming model oriented to development of ubiquitous systems. UOP uses Services concepts and Object Oriented Programming, integrating them with the requirements of ubiquitous applications as context, context awareness, context adaptation, code mobility and concurrent. A programming language (UbiLanguage) implements the concepts of this new programming model, thus providing the development of ubiquitous systems. The development environment is composed by a compiler (UbiCompiler) which translates code written in UbiLanguage to bytecode. The execution platform is composed by a virtual machine (UbiVM) which supports the execution of this bytecode.

Keywords: Ubiquitous Computing, Pervasive Computing, Programming Paradigms, Programming Models, Programming Languages, Virtual Machines, Compilers.

Sumário

Lista de Figuras

Lista de Abreviaturas

API	Application Programming Interface
BNF	Backus Naur Form
CP	Constant Pool
FDL	GNU Free Documentation License
GCC	GNU Compiler Collection
GPL	GNU General public license
JVM	Java Virtual Machine
LBS	Location Based Services
MV	Máquina Virtual
OOP	Object Oriented Programming
PDA	Personal Digital Assistant
SMP	Simetric Multiprocessing
TTL	Time to Live
UOP	Ubiquitous Oriented Programming
UbiL	UbiLanguage
UbiC	UbiCompiler
Ubicomp	Ubiquitous Computing
UbiVM	Ubiquitous Virtual Machine
WAM	Warren Abstract Machine

Lista de Tabelas

Capítulo 1

Introdução

A computação ubíqua (*Ubiquitous Computing*, abreviadamente *ubicomp*) [Weiser 1991, Satyanarayanan 2001, Costa, Yamin e Geyer 2008], também conhecida como computação pervasiva (*Pervasive Computing*), integra diversas tecnologias e idéias existentes. Enquanto a computação baseada em *desktop* possui uma abordagem estática, a computação ubíqua vale-se da mobilidade do usuário através de dispositivos móveis e serviços providos pelas redes sem fio. A possibilidade de utilizar a computação de forma integrada às atividades do dia-a-dia faz com que ela desapareça, do ponto de vista do usuário. O usuário acessa o seu ambiente computacional, em qualquer lugar, há qualquer tempo, e em qualquer dispositivo, sendo que sempre está inserido em um ambiente que dispõe de um poder computacional.

Com a maior popularização dos dispositivos móveis, já é possível um uso computacional praticamente ubíquo. As redes sem fio propiciam acesso à Internet nas áreas urbanas e a melhoria nas interfaces dos dispositivos permite uma experiência cada vez mais próxima e integrada ao cotidiano. O próximo passo, já presente em algumas aplicações, é a percepção dos atributos do usuário e do ambiente, tais como a localização física, estado emocional e história pessoal.

A popularização dos dispositivos móveis, aliado ao uso das informações de contexto, permitem o desenvolvimento de aplicações que propiciem uma interação ubíqua. A sensibilidade ao contexto é a característica que as aplicações devem incorporar para fazer uso das informações que caracterizam a situação

das pessoas, lugares ou objetos [Dey 2001]. A sensibilidade ao contexto pode ser implementada combinando informações como localização do usuário e suas preferências [Naismith e Smith 2004]. Este modelo computacional, denominado de Sensível ao Contexto, se beneficia do uso de informações contextuais para adaptar o comportamento dos sistemas ao seu contexto, aprimorando a interação com seus usuários. Saha e Mukherjee [Saha e Mukherjee 2003] defendem que a adaptação do ambiente é uma das características chaves que diferencia a computação ubíqua da computação tradicional.

Um dos principais objetivos de uma aplicação sensível ao contexto é atender a diversos usuários, diferenciando-os através de perfis, fornecendo informações pertinentes levando em consideração o ambiente físico (localização) em que eles se encontram no momento [Syvanen et al. 2005]. Entretanto, muitas aplicações sensíveis ao contexto ainda estão sendo executadas em laboratórios, ao invés de estarem presentes em ambientes reais do dia-a-dia [Henricksen e Indulska 2006].

Relacionar os interesses do usuário com a sua localização possibilita o que tem sido denominado de *Location Based Services* (Serviços Baseados em Localização, abreviadamente LBS) onde não somente a localização é informada, mas um conjunto de serviços contextualizados são oferecidos [Vaughan-Nichols 2009].

Desenvolvimentos recentes [Ubisense site, U-BLOX site] demonstram que a precisão atual da localização permite a implementação de aplicações comerciais. A precisão de localização estimulará a computação ubíqua, gerando novas oportunidades em áreas como educação, comércio, redes sociais, medicina, jogos, entre outros.

1.1 Motivação

A dificuldade ainda reside em como desenvolver aplicações que adaptam-se continuamente ao ambiente, sem parar sua execução, mesmo com o usuário movendo-se ou alternando de dispositivo [Grimm 2004]. Aplicações móveis sensíveis ao contexto são difíceis de projetar e implementar devido a necessidade de um comportamente dinamicamente adaptativo às condições ambientais em que executam [Dey 2000]. Aplicações ubíquas necessitam de uma abstração entre

os diferentes dispositivos e as aplicações dos usuários para que possam abstrair a complexidade do ambiente e isolar as aplicações da necessidade de gerenciar protocolos, memória distribuída e falhas de comunicação [Saha e Mukherjee 2003].

O desenvolvimento de aplicações ubíquas exige que os projetistas de sistemas computacionais adaptem-se à nova realidade. Para tanto, devem ter consciência de que a computação móvel e ubíqua constitui um novo paradigma computacional centrado no usuário e suas tarefas, com a computação inserida no ambiente [Yamin e Augustin 2006]. Precisam, então, desenvolver aplicações que tirem proveito das novas tecnologias, ao mesmo tempo em que satisfazem as necessidades cada vez mais específicas de um número cada vez maior de usuários.

As aplicações móveis e ubíquas, que visam suprir parte dessas necessidades, são desenvolvidas por linguagens de programação que apresentam suporte à mobilidade. Porém, a falta de modelos conceituais e métodos, com ênfase na computação ubíqua, dificulta o uso das novas tecnologias existentes por estas aplicações.

Esforços estão sendo dedicados atualmente ao desenvolvimento de sistemas distribuídos. Linguagens e *frameworks* estão sendo utilizados para implementar estes sistemas. Alguns autores afirmam que o paradigma orientado a objetos (OOP), apesar de ser o modelo de programação mais utilizado, não é suficiente para a computação ubíqua [Costa 2008].

No modelo de programação tradicionalmente em uso para sistemas distribuídos existem limitações que afetam a computação ubíqua [Grimm 2004]. Na distribuição transparente ambos os recursos locais e remotos podem ser utilizados praticamente da mesma forma, simplificando assim a programação de sistemas distribuídos. Porém isto dificulta a gerência da sensibilidade ao contexto. A abstração encapsula código e dados, mantendo-os dentro dos objetos, o que dificulta o compartilhamento destes dados.

Além destas limitações, os modelos tradicionais (paradigma imperativo, lógico, funcional e orientado a objetos) geralmente são baseados em premissas estáticas de arquitetura, aplicação, dados e sistema operacional. Dados são usualmente armazenados sem uma definição de formato próprio. Além disso, os recursos que serão utilizados devem ser conhecidos *a priori*. Como resultado, não é fácil criar aplicativos ubíquos utilizando apenas modelos tradicionais e OOP.

Este cenário motiva a criação de um modelo de programação focado na computação ubíqua que facilite o desenvolvimento das aplicações previstas por Weiser [Weiser 1991] e Satyanarayanan [Satyanarayanan 2001].

1.2 Definição do Problema

Apesar das várias propostas para facilitar o desenvolvimento de aplicações ubíquas, não foi encontrado um modelo de programação que contemple todos os conceitos necessários nestas aplicações. A maioria das abordagens para facilitar o desenvolvimento destas aplicações tem sido embutida em outros domínios de pesquisa como *object-oriented programming, aspect-oriented programming, feature-oriented programming* e *dynamic languages*.

Com esta visão, surgem as seguintes questões de pesquisa: É possível criar um modelo de programação específico para o desenvolvimento de aplicações ubíquas, integrando os conceitos previstos na computação ubíqua ? É possível, baseado neste modelo, criar um ambiente que suporte a implementação e execução destas aplicações ?

1.3 Objetivos

O presente trabalho propõe o *Ubiquitous Oriented Programming* (abreviadamente UOP), um modelo de programação focado no desenvolvimento de sistemas ubíquos. UOP integra os conceitos da OOP e serviços com os requisitos de aplicativos ubíquos [Costa, Yamin e Geyer 2008] como contextos, sensibilidade ao contexto, adaptação ao contexto, mobilidade de código e concorrência.

Diferentemente de outras propostas, que focam apenas no ambiente e serviços, UOP foca também no desenvolvimento de aplicações. Para tal, além do modelo, também define uma linguagem e o suporte para construir e executar aplicações ubíquas.

O ambiente para desenvolvimento e execução de aplicativos proposto para o UOP pode ser visto na figura 1.1. O **Ambiente de Desenvolvimento** é composto pela *UbiLanguage* (abreviadamente UbiL) e pelo *UbiCompiler* (abreviadamente

UbiC). UbiL é a linguagem de programação que concretiza os conceitos propostos no UOP enquanto que o UbiC é o compilador que traduz código escrito em UbiL para *bytecode*. A **Plataforma de Execução** é composta pelo Sistema Computacional (*Hardware* + Sistema Operacional) e pela *Ubiquitous Virtual Machine* (abreviadamente UbiVM). A UbiVM é a máquina virtual ubíqua responsável por executar o *bytecode* gerado pelo UbiC. As UbiVMs interagem entre si para gerenciar a distribuição dos conteúdos e serviços existentes nos contextos, os quais são criados e gerenciados dinamicamente pelos aplicativos em execução.

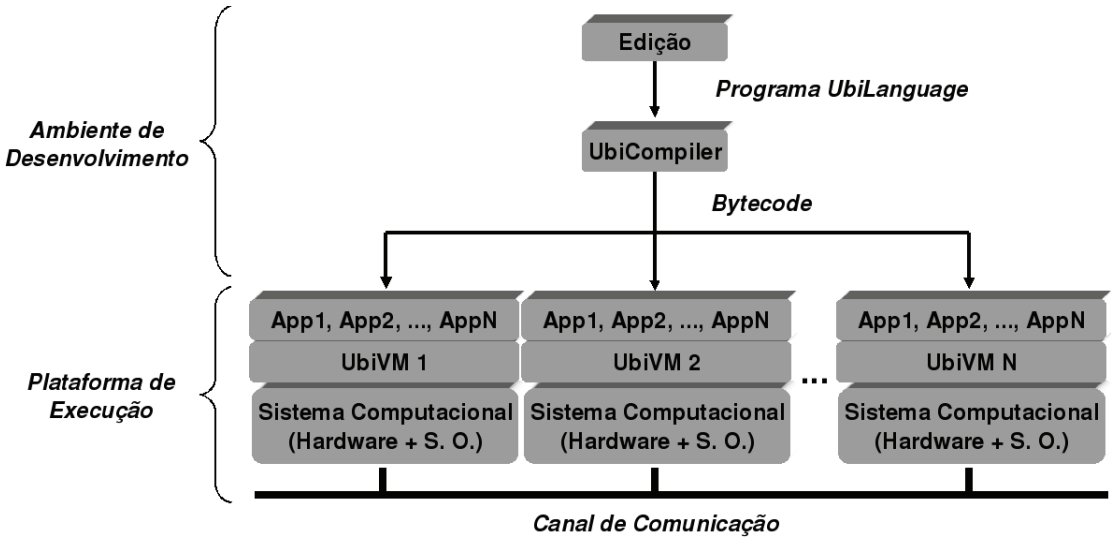


Figura 1.1: Ambiente de desenvolvimento e execução de aplicativos

Os objetivos específicos deste trabalho são:

- definir o UOP, um modelo de programação que contemple os conceitos envolvidos em aplicações ubíquas, provendo assim suporte à modelagem destas aplicações;
- definir a UbiL, uma linguagem de programação que concretize os conceitos propostos no UOP, permitindo assim o desenvolvimento destas aplicações;
- definir e implementar o UbiC, o compilador que traduz código em UbiL para *bytecode*, podendo ser executado pela UbiVM;

- definir e implementar a UbiVM, a máquina virtual responsável pela execução deste *bytecode*, possibilitando assim a execução destas aplicações;
- avaliar este trabalho através do desenvolvimento de aplicações ubíquas nas áreas da educação, comércio e redes sociais.

1.4 Organização da Dissertação

Esta dissertação está organizada em nove capítulos. São eles:

- *Capítulo 2*: descreve conceitos relacionados ao presente trabalho, ou seja, computação ubíqua, paradigmas de programação, máquinas virtuais e serviços;
- *Capítulo 3*: descreve os trabalhos relacionados, agrupados em modelos de programação e ambientes de computação ubíqua;
- *Capítulo 4*: descreve o UOP, abordando os conceitos herdados da OOP e dos serviços, bem como os novos conceitos introduzidos neste modelo;
- *Capítulo 5*: descreve a UbiL, a linguagem proposta que concretiza os conceitos do UOP;
- *Capítulo 6*: apresenta a UbiVM, abordando sua arquitetura e como seus recursos podem ser utilizados para executar os aplicativos escritos em UbiL;
- *Capítulo 7*: descreve os protótipos do UbiC e da UbiVM;
- *Capítulo 8*: contém os experimentos realizados para avaliar o presente trabalho;
- *Capítulo 9*: contém as considerações finais, onde são abordadas as principais contribuições, conclusões e trabalhos futuros.

Além destes capítulos, foram introduzidos quatro apêndices, os quais são citados quando necessário. São eles:

- *Apêndice A*: apresenta a gramática da UbiL;

- *Apêndice B*: contém o código fonte dos experimentos apresentados no capítulo 8;
- *Apêndice C*: apresenta a descrição detalhada das instruções presentes na UbiVM, com exemplos de uso;
- por fim, *Apêndice D*: apresenta o formato do arquivo UVM, o arquivo que contém o *bytecode* gerado pelo UbiC.

Capítulo 2

Conceitos básicos

Este capítulo aborda os conceitos utilizados no desenvolvimento deste trabalho: computação ubíqua, paradigmas de programação, máquinas virtuais e serviços.

2.1 Computação Ubíqua

Mark Weiser, em 1991, afirmava que as tecnologias mais profundas são aquelas que desaparecem [Weiser 1991]. Em outras palavras, as tecnologias mais profundas são aquelas que o homem não percebe que está utilizando.

As principais características da computação ubíqua incluem:

1. Heterogeneidade: consiste no suporte a variados e diferentes dispositivos, sistemas operacionais, redes, linguagens de programação e aplicações [Costa, Yamin e Geyer 2008];

2. Escalabilidade: propriedade que faz com que o sistema continue efetivo mesmo quando existe um aumento significativo no número de usuários [Costa, Yamin e Geyer 2008];

3. Dependabilidade (*Dependability*): é a capacidade de evitar defeitos nos serviços, mais freqüentes ou mais severos do que o aceitável [Costa, Yamin e Geyer 2008, Ucla et al. 2001];

4. Segurança: refere-se à existência concomitante de disponibilidade somente de ações autorizadas, confidencialidade e integridade [Costa, Yamin e Geyer 2008, Ucla et al. 2001];

5. Privacidade: diz respeito ao risco de muita exposição de informações pessoais, que ocorre muitas vezes sem o conhecimento do usuário. Além disso, envolve discussões sobre o conjunto e a precisão de dados coletados [Costa, Yamin e Geyer 2008];

6. Confiança: está relacionada ao estabelecimento de confiança entre os componentes que estão interagindo [Costa, Yamin e Geyer 2008];

7. Interoperação espontânea: interações de um conjunto de componentes que se comunicam e que podem alterar tanto a identidade quanto a funcionalidade no decorrer do tempo quando as circunstâncias mudam [Costa, Yamin e Geyer 2008, Kindberg e Fox 2002]. Nota-se que componentes espontâneos suportam mudança freqüente nos parceiros de comunicação e fácil interação;

8. Mobilidade: propriedade que permite ao usuário ter acesso a dados e aplicativos independente da sua localização ou do seu deslocamento. Para tanto, o ambiente acompanha o usuário. A mobilidade pode ser lógica (de aplicativos, dados e serviços) ou física (de usuários e equipamentos). A mobilidade lógica, também conhecida como mobilidade de código, é a capacidade de alterar dinamicamente as ligações entre fragmentos de código e a localização na qual eles executam. Existem dois tipos: mobilidade fraca e mobilidade forte [Fuggetta, Picco e Vigna 1998, Ghezi e Vigna 1997, Thorn 1997, Naseen 2004]. Na **mobilidade fraca**, somente o código é movido e, em alguns casos, dados necessários para inicialização da execução no ambiente remoto também são movidos. Na **mobilidade forte**, o código de uma unidade de execução é movido juntamente com seu estado de execução para um ambiente computacional diferente. O suporte à mobilidade forte pode ocorrer através de duas abordagens: migração e clonagem remota. Na **migração** a execução em uma unidade de execução é interrompida, seu estado de execução é armazenado juntamente com o código e então migrado

para outro ambiente computacional. Na abordagem que utiliza **clonagem remota**, a unidade de execução continua executando em seu ambiente computacional enquanto ela é clonada para outro ambiente computacional;

9. Sensibilidade ao contexto: todo elemento usado para caracterizar a situação de entidades (pessoas, lugares e objetos) que for considerado relevante para a interação entre o usuário e a aplicação representa uma informação de contexto [Dey 2000]. Assim, sensibilidade ao contexto refere-se à obtenção de informações relevantes através de sensores (combinação de hardware e software) usados para medir valores contextuais, que permitem a adaptação de comportamento das aplicações ao seu contexto (localização física, por exemplo);

10. Adaptação ao contexto: permite que o sistema haja de forma inteligente (*smartness*) e necessita de elementos abstratos para suportar as diferentes mudanças. Alguns autores consideram como parte da sensibilidade ao contexto, pois está associada à tomada de decisões baseada nas informações de contexto. Satyanarayanan [Satyanarayanan 1996] caracteriza diferentes formas de adaptação: **adaptação consciente da aplicação** (colaboração entre sistema e aplicação), **adaptação sem suporte do sistema** (aplicação é totalmente responsável por adapta-se) e **adaptação transparente à aplicação** (sem alterações na aplicação);

11. Interação transparente com o usuário: está relacionada com a geração de interfaces adaptadas aos dispositivos e integradas ao mundo real. Neste sentido, idealmente, os sistemas computacionais devem considerar a forma mais natural de interação com o usuário. Para tanto, poderiam utilizar informações contextuais e comportamentos do usuário [Costa, Yamin e Geyer 2008];

12. Invisibilidade: é a característica que permite a computação ubíqua como foi concebida por Mark Weiser [Weiser 1991]. Refere-se à integração total dos computadores com o ambiente de forma a desaparecerem no entorno. Esta característica é tópico de pesquisas atuais, pois é a mais difícil de ser obtida, uma vez que requer todas as características

apresentadas integradas numa única infra-estrutura, e exige um alto nível de automatização [Costa, Yamin e Geyer 2008].

As características 1, 2, 3 e 4, são também relacionadas com a área de sistemas distribuídos e, em função disso, algumas tecnologias já existem. As características 5, 6, 7, 8, 9 e 10 são tópicos de pesquisas da computação móvel. As características 11 e 12, por sua vez, referem-se a novos desafios vinculados com a área de computação ubíqua.

A localização do usuário é um fator importante para determinar a sua mobilidade. Pesquisadores estão trabalhando para desenvolver sistemas e tecnologias que automaticamente localizam pessoas e equipamentos. Um sistema de localização pode prover dois tipos de informação: física e simbólica [Hightower, Lamarca e Smith 2006, Hightower e Borriello 2001]. Um sistema de GPS provê posição física. Em contraste, localização simbólica engloba idéias abstratas de onde alguma coisa está, por exemplo, "na cozinha", "próximo a caixa de correio", "no trem chegando ao seu destino".

Existem dois tipos de sistemas de localização: absoluta e relativa [Hightower e Borriello 2001]. **Sistemas de localização absoluta** utilizam uma referência compartilhada para todos os objetos localizados. Por exemplo, todos os receptores GPS utilizam latitude, longitude e altitude. Dois receptores GPS colocados na mesma posição irão reportar a mesma posição. Em **sistemas de localização relativos**, cada objeto tem seu próprio plano de referência. Com isso o dispositivo reporta a posição relativa ao objeto em relação a si mesmo.

A figura 2.1 mostra como os problemas de pesquisa da computação ubíqua relacionam-se com a computação móvel e sistemas distribuídos [Satyanarayanan 2001]. Novos problemas são encontrados quando move-se da esquerda para a direita nesta figura. Adicionalmente, a solução para problemas encontrados anteriormente tornam-se mais complexas. Como o símbolo de modulação sugere, a complexidade é multiplicada, ao invés de somada; é muito mais difícil desenvolver e implementar um sistema de computação ubíqua do que um sistema distribuído de comparável robustez e maturidade.

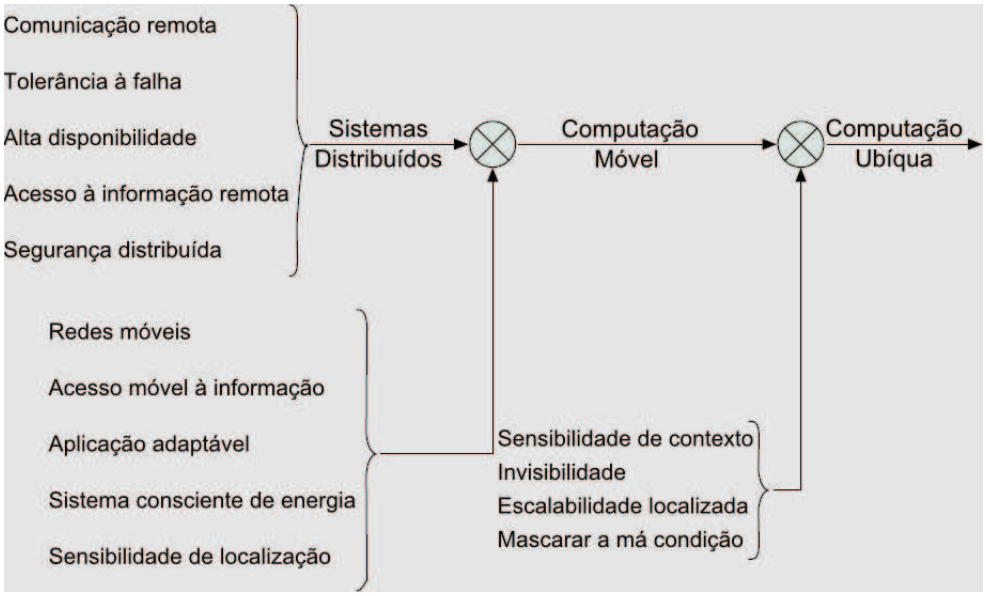


Figura 2.1: Taxonomia da computação ubíqua [Satyanarayanan 2001]

2.2 Paradigmas de Programação

Um Paradigma (ou modelo) de Programação é um padrão ou estilo de programação suportado por linguagens que agrupam certas características comuns. Um paradigma de programação fornece (e determina) a visão que o programador possui sobre a estruturação e execução do programa.

Um paradigma suporta a modelagem de sistemas computacionais. Com este intuito, estabelece um conjunto de abstrações que serão utilizadas na criação dos modelos. A eficiência do paradigma está relacionada com a distância entre o significado destas abstrações e os conceitos existentes no domínio modelado. No âmbito da engenharia de software, esta distância é denominada *gap* semântico (figura 2.2). Um pequeno *gap* semântico significa que as abstrações são adequadas. A semântica está relacionada com o significado. Os paradigmas buscam diminuir este *gap* semântico. No caso dos paradigmas, a semântica estabelece o significado de suas abstrações. Por exemplo, o paradigma orientado a objetos possui como principal abstração o objeto. No âmbito deste paradigma, objeto é a unidade de modelagem e significa qualquer realidade que possa ser organizada em uma unidade que encapsule atributos e métodos [Ghezzi e Jazayeri 1998, Sebesta 1999].

Algumas linguagens foram desenvolvidas para suportar um paradigma

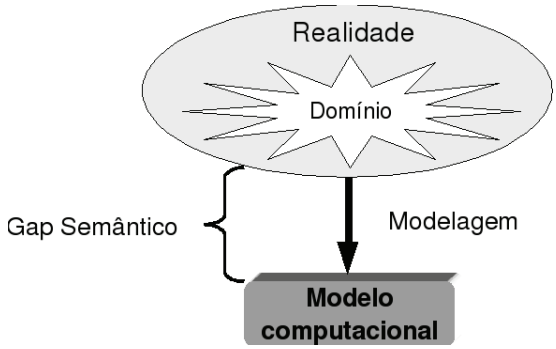


Figura 2.2: *Gap* semântico

específico (Smalltalk e Java suportam o paradigma de orientação a objetos enquanto Haskell e Scheme suportam o paradigma funcional), enquanto outras linguagens suportam múltiplos paradigmas (como Perl, Python, C++ e Oz).

2.2.1 Paradigma Imperativo

A Programação Imperativa (*Imperative Programming*), mais popular na década de 70, concentra-se em subprogramas e em bibliotecas de subprogramas. Dados são enviados a subprogramas para computações [Sebesta 1999]. É um paradigma de programação que descreve a computação como ações (comandos) que mudam o estado (variáveis) de um programa. Programas imperativos são uma sequência de comandos para o computador executar. Procedimentos (também conhecidos como rotinas, sub-rotinas, métodos ou funções) contém um conjunto de passos computacionais a serem executados. Um dado procedimento pode ser chamado a qualquer hora durante a execução de um programa, inclusive por outros procedimentos ou por si mesmo. As entradas costumam ser especificadas sintaticamente na forma de argumentos e as saídas na forma de valores de retorno.

C, Pascal e BASIC são exemplos de linguagens imperativas.

2.2.2 Paradigma Funcional

O paradigma de programação funcional, baseado em funções matemáticas, é a base de projeto dos estilos das linguagens funcionais mais importantes [Sebesta 1999]. Uma função matemática é uma correspondência biunívoca de membros de um conjunto (conjunto domínio) com outro (conjunto

imagem). Uma função produz ou retorna um elemento do conjunto imagem. Uma função, neste sentido, pode ter ou não parâmetros e um simples valor de retorno. Os parâmetros são os valores de entrada da função, e o valor de retorno é o resultado da função.

Uma das características fundamentais das funções matemáticas é que a ordem de avaliação de suas expressões de correspondência é controlada por recursão e por expressões condicionais, não pela seqüência ou pela repetição iterativa comuns nas linguagens de programação imperativas. Outra característica é que, dado o mesmo conjunto de argumentos, sempre definem o mesmo valor.

Na programação funcional normalmente as funções são puras, ou seja, não dependem de nenhum "efeito colateral", como variáveis globais, passagem por referência, enfim, seu resultado depende apenas dos parâmetros de entrada.

Lisp e Haskell são exemplos de linguagens funcionais.

2.2.3 Paradigma em Lógica

No paradigma em lógica a abordagem é expressar programas na forma de lógica simbólica e usar um processo de inferência lógica para produzir resultados [Sebesta 1999]. Os programas lógicos são declarativos em vez de baseados em procedimentos, o que significa que somente as especificações dos resultados desejados são declarados em vez de procedimentos para gerá-los.

A programação que usa uma forma de lógica simbólica como linguagem freqüentemente é chamada programação lógica e as linguagens baseadas na lógica simbólica são chamadas linguagens de programação lógica ou declarativas.

Prolog é um exemplo de linguagem de programação lógica.

2.2.4 Paradigma Orientado a Objetos

O conceito de Programação Orientado a Objetos (*Object-Oriented Programming*, abreviadamente OOP) tem suas raízes na SIMULA 67, mas não foi amplamente desenvolvido até que a evolução da Smalltalk resultasse na produção de sua versão 80. Os tipos de dados abstratos em linguagens orientadas a objetos são classes e as instâncias de classes são chamadas objetos [Sebesta 1999]. Uma classe definida pela herança de outra é uma classe derivada ou subclasse. Uma

classe da qual a nova é derivada é sua classe-pai ou superclasse. Os subprogramas que definem as operações em objetos de uma classe são chamados métodos. As chamadas a métodos são denominadas de mensagens. Assim, as computações em um programa orientado a objetos são especificadas pelas mensagens enviadas de objetos para outros objetos.

Um Objeto é a principal abstração deste paradigma, é a unidade de modelagem e significa qualquer realidade que possa ser organizada em uma unidade que encapsule atributos e métodos [Ghezzi e Jazayeri 1998]. Em OOP as classes definem os objetos presentes no sistema de software. Cada classe determina o comportamento e possíveis estados de seus objetos, assim como o relacionamento com outros objetos.

Smalltalk, Eiffel, C++, C# e Java são exemplos de linguagens orientadas a objetos.

2.3 Máquinas Virtuais

Sebesta [Sebesta 1999] define três métodos de implementação de linguagens: compilação, interpretação pura e sistemas de implementação híbridos.

Na **Compilação** (figura 2.3) as linguagens podem ser traduzidas para linguagem de máquina, a qual pode ser executada diretamente no computador. Assim que o processo de compilação é concluído, este método tem a vantagem de uma execução de programa rápida. A desvantagem desta abordagem é que o processo de compilação é mais lento do que na interpretação pura [Sebesta 1999].

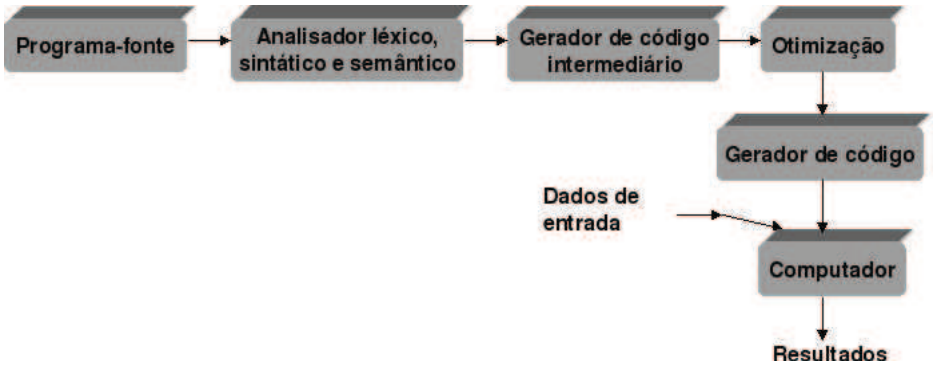


Figura 2.3: Etapas do processo de compilação

Na **Interpretação pura** (figura 2.4) os programas são executados (interpretados) por outro programa chamado interpretador, sem nenhuma conversão. O programa interpretador age como uma simulação de software de uma máquina, lidando com instruções de alto nível em vez de instruções de máquina, fornecendo assim uma máquina virtual para a linguagem [Sebesta 1999]. Esta técnica tem a vantagem de permitir uma fácil implementação de muitas operações de depuração do código-fonte. Por outro lado, este método tem a desvantagem de que a execução é de 10 a 100 vezes mais lenta do que em sistemas compilados. Outra desvantagem é que ela freqüentemente exige um maior espaço em memória.

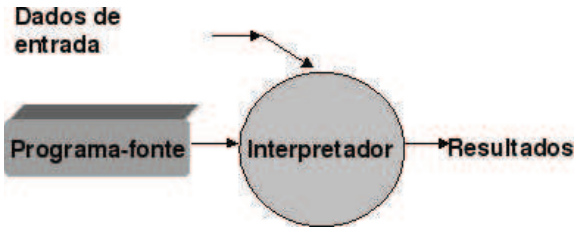


Figura 2.4: Etapas do processo de interpretação pura

Sistemas de implementação híbridos (figura 2.5) são considerados uma abordagem de implementação híbrida porque são um meio-termo entre os compiladores e os interpretadores puros. Nesta abordagem programas em linguagem de alto nível são traduzidos para uma linguagem intermediária projetada para permitir fácil interpretação [Sebesta 1999]. Este método é mais rápido do que a interpretação pura porque as instruções da linguagem fonte são decodificadas somente uma vez. Nesta abordagem, em vez de traduzir código em linguagem intermediária para código de máquina (método da compilação), ele simplesmente interpreta o código intermediário.

A **Máquina Virtual** é o componente identificado como interpretador no sistema de implementação de linguagens híbrido, sendo responsável por executar (interpretar) o *bytecode* (código intermediário), criando assim uma abstração de hardware sobre o sistema computacional (*Hardware* + Sistema Operacional).

Um ambiente típico de uma máquina virtual (figura 2.6) é dividido em:

- *Ambiente de desenvolvimento*: Composto de um editor de textos capaz de gerar um arquivo em uma linguagem aceita pelo compilador. Este

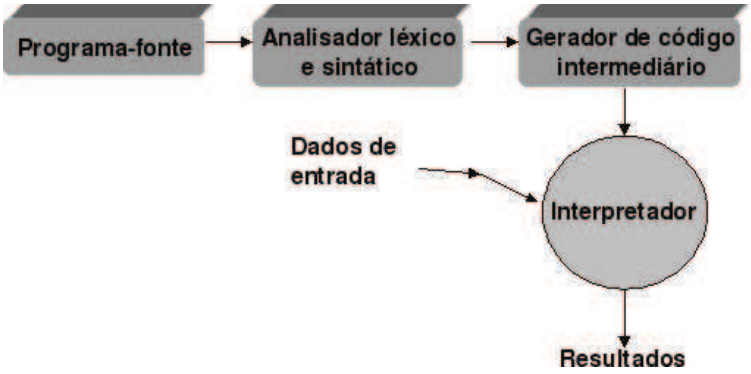


Figura 2.5: Etapas do processo de implementação híbrida

compilador, por sua vez, traduz este arquivo para um código virtual aceito pela máquina virtual;

- *Plataforma de execução*: Composto pela máquina virtual que executa este código virtual. Esta máquina cria a abstração de hardware necessária para que este código virtual seja executado em qualquer plataforma onde exista uma implementação desta máquina virtual, tornando assim este código independente de hardware e sistema operacional.

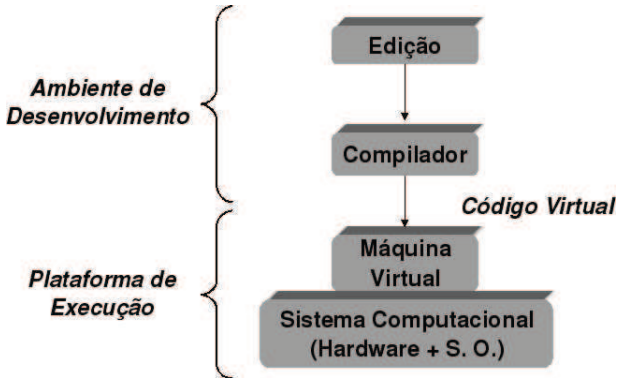


Figura 2.6: Ambiente típico de uma máquina virtual

As instruções existentes em um *bytecode* (código virtual) são a representação binária de um programa escrito na linguagem de máquina compatível com a máquina virtual. Nesta representação as instruções são traduzidas para um código binário compreensível para esta máquina virtual.

O laço principal do interpretador de uma máquina virtual pode ser visto na figura 2.7. Em cada ciclo do laço, a máquina virtual busca e executa uma instrução. Uma instrução é composta de um *opcode* e seus argumentos.

```
faça {
   busca um opcode;
   se (tem operandos) então busca operandos;
   executa o opcode;
} enquanto (houver mais opcodes);
```

Figura 2.7: Laço principal do interpretador

Alguns exemplos de máquinas virtuais são:

- WAM (*Warren Abstract Machine*) [Warren 1983, Aït-kaci 1991];
- JVM (*Java Virtual Machine*) [Lindholm e Yellin 1999, Tucker 1998, Moss 2000, Yourst 1998];
- CLR (*Common Language Runtime*) [CLI: Common Language Infrastructure];
- LuaVM (*Lua Virtual Machine*) [Ierusalimschy, Figueiredo e Celes];
- HoloVM (*Holo Virtual Machine*) [Garzão e Barbosa 2002, Garzão e Barbosa 2003].

2.4 Serviços

Nos *Web Services* aplicativos e rotinas são disponibilizadas em uma rede de computadores como serviços independentes, comunicando-se através de padrões abertos [Austin Abbie Barbir e Garg 2002]. Um *Web Service* é um sistema de software designado para suportar interoperabilidade na interação máquina-a-máquina através da rede.

Um *Web service* é uma noção abstrata que deve ser implementada por um agente [Booth et al. 2004]. O **agente** é um pedaço de software ou hardware que envia e recebe mensagens, enquanto que o **serviço** é o recurso caracterizado pelas funcionalidades que ele disponibiliza. Um serviço é uma função de um sistema computacional que é disponibilizado para outro sistema. Serviços permitem que os aplicativos se comuniquem entre si de modo independente da plataforma.

A arquitetura de *Web Services* [Erl 2005] é baseada na interação entre três componentes principais (figura 2.8): **Provedor de serviços**, **Registro de serviços** e **Consumidor de serviços**. Esses componentes interagem usando operações de publicação, busca e ligação. Na etapa 1 da figura 2.8 o provedor é a entidade que provê acesso aos seus serviços ao publicar a descrição dos serviços em um registro, informando características e aspectos relevantes às tomadas de decisões. A publicação de um serviço é realizada através de sua descrição seguindo o padrão WSDL (*Web Services Description Language*). Na etapa 2 o consumidor localiza no registro, através da descrição, o serviço que tem interesse, e utiliza esta informação para realizar a ligação com os provedores escolhidos. Nesta etapa, o consumidor recebe, além da informação de como acessar o provedor, informações de como a requisição ao serviço deve ser realizada. As informações de requisição são o nome da operação a ser acessada e os parâmetros que devem ser informados (bem como os seus tipos). De posse destas informações, na etapa 3 o serviço pode ser diretamente acessado pelo consumidor. Como o consumidor tem acesso as características de um serviço, pode utilizar estas informações para determinar se o serviço atende suas necessidades.

Figura 2.8: Arquitetura orientada a serviços

Serviços funcionam de forma independente do estado de outros serviços (*stateless*), e possuem uma interface bem definida. Isso possibilita que o consumidor do serviço possa sequenciá-lo, ou seja, orquestrá-lo em vários fluxos (*pipelines*) para executar a lógica de uma aplicação.

2.5 Considerações sobre o capítulo

Este capítulo abordou os conceitos utilizados no desenvolvimento desta dissertação. São eles: computação ubíqua, paradigmas de programação, máquinas virtuais e serviços.

O próximo capítulo aborda trabalhos que suportam o desenvolvimento de software com características que interessam a ubiquidade: contexto, sensibilidade ao contexto, adaptação ao contexto, mobilidade de código e concorrência.

Capítulo 3

Trabalhos relacionados

Este capítulo contém trabalhos que suportam o desenvolvimento de software com características que interessam a ubiquidade: contextos, serviços, sensibilidade ao contexto, adaptação ao contexto, mobilidade de código e concorrência. A seção 3.1 aborda modelos de programação enquanto que a seção 3.2 aborda ambientes de computação ubíqua.

3.1 Modelos de programação

Esta seção aborda modelos de programação com características que interessam a ubiquidade.

3.1.1 *Subject-Oriented Programming*

Programação Orientada a Assuntos (*Subject-Oriented Programming*, ou SOP) estende os objetos e mensagens da programação orientada a objetos incorporando perspectivas, também conhecidas como *subjects* (assuntos) [Harrison e Ossher 1993]. A programação é dirigida pelos estados do objeto e as especificações do seu comportamento relacionados a uma expectativa ou objetivo em particular. Um objeto pode ser visto de múltiplas perspectivas, isto é, um objeto pode responder diferentemente para mensagens enviadas de diferentes perspectivas. Métodos são selecionados não apenas baseando-se no nome da mensagem e de seu receptor, mas também em quem está enviando.

A implementação mais conhecida de SOP é o Hyper/J, uma ferramenta que permite a composição de classes Java de acordo com regras de composição definidas em um arquivo.

3.1.2 *Context-Oriented Programming*

Programação Orientada a Contextos (*Context-Oriented programming*, ou COP) é uma forma de pensar sobre sistemas de software [Hirschfeld, Costanza e Nierstrasz 2008]. Contextos expressam a idéia de um sistema de relações e assemelham-se as classes e objetos da OOP. Contextos possibilitam um meio de implementar variações customizadas da OOP.

COP possibilita a expressão de variações comportamentais dependendo do contexto. Linguagens que implementem COP devem endereçar os seguintes requisitos:

- *Behavioral variations*: variações comportamentais tipicamente consistem de novos ou modificados comportamentos, mas podem também indicar um comportamento removido. Podem ser expressados como definições parciais de módulos no modelo de programação, como procedimentos ou classes;
- *Layers*: são relacionados a comportamento, dependentes de contexto;
- *Activation*: *layers* agregando comportamentos dependentes de contexto podem ser ativados e desativados dinamicamente durante a execução. O código pode habilitar ou desabilitar *layers* baseado no contexto atual;
- *Context*: qualquer informação que é computacionalmente acessível pode formar parte do contexto que as variações de comportamento dependem;
- *Scoping*: o escopo de quais *layers* serão ativados ou desativados podem ser controlados explicitamente. Variações podem ser simultaneamente ativadas ou não dentro de diferentes escopos da mesma aplicação em execução.

Os modelos de programação podem ser interpretados como um sistema multidimensional. São eles:

- Sistemas com uma dimensão: *procedural programming* provê apenas uma dimensão para associar uma unidade computacional com um nome. Chamadas de procedimentos, ou nomes, são diretamente mapeados para implementações de procedimentos;

- Sistemas com duas dimensões: *object-oriented programming* adiciona outra dimensão para a resolução de nomes. Ao buscar por um método, além do nome do método ou procedimento, o envio de mensagens considera também o receptor da mensagem;

- Sistemas com três dimensões: *subjective programming* estende o método de envio de mensagens da *objetc-oriented programming*, adicionando outra dimensão. Métodos são selecionados não apenas baseando-se no nome da mensagem e de seu receptor, mas também em quem está enviando.

Context-oriented programming avança um passo sobre *subjective programming* ao utilizar um sistema com quatro dimensões. Mensagens são enviadas não apenas baseando-se no nome da mensagem, quem está enviando e quem recebe, mas também baseando-se no contexto atual da mensagem enviada. Baseado nas informações do contexto, métodos ou suas definições parciais são selecionados ou excluídos no envio da mensagem.

Existem bibliotecas de extensão para COP nas linguagens Java (ContextJ), Squeak/Smalltalk (ContextS), Lisp (ContextL), Groovy (ContextG), Python (ContextPy) e Ruby (ContextR).

3.1.3 *Aspect-Oriented Programming*

Na programação Orientada a Aspectos (*Aspect-Oriented Programming*, ou AOP), uma aplicação é estruturada em módulos (aspectos) que agrupam pontos de interceptação de código (*pointcuts*) que afetam outros módulos (classes) ou outros aspectos, definindo um novo comportamento (*advice*) [Kiczales et al.].

AOP é uma técnica de programação de computadores que permite aos desenvolvedores de software separar e organizar o código de acordo com a sua importância para a aplicação (*separation of concerns*). Todo o programa escrito na AOP possui código que é alheio a implementação do comportamento do objeto.

Este código é todo aquele utilizado para implementar funcionalidades secundárias e que encontra-se espalhado por toda a aplicação (*crosscutting concern*). A AOP permite que este código seja encapsulado e modularizado.

O conceito foi criado por Gregor Kiczales e a sua equipe na Xerox PARC, a divisão de pesquisa da Xerox. Eles desenvolveram o AspectJ [AspectJ 2009], a primeira e mais popular linguagem AOP.

Os paradigmas de programação mais antigos, como a programação imperativa e a programação orientada a objetos, implementam a separação do código, através de entidades únicas. Por exemplo, a funcionalidade de *log* de dados, numa linguagem orientada a objetos, é implementada em uma única classe, que é referenciada em todos os pontos onde é necessário fazer *log* de dados. Como praticamente todo método necessita que dados sejam registrados em *log*, as chamadas a esta classe são espalhadas por toda a aplicação. Tipicamente uma implementação da AOP busca encapsular essas chamadas através de uma nova construção chamada de "aspecto". Um aspecto pode alterar o comportamento de um código (a parte do programa não orientada a aspectos) pela aplicação de um comportamento adicional (*advice*), sobre um ponto de execução (*join point*). A descrição lógica de um conjunto de *join points* é um *pointcut*.

Em muitas linguagens AOP, a execução de um método e referências a atributos são exemplos de *join points*. Um *pointcut* consiste, por exemplo, de todas as referências a um conjunto de atributos.

A AOP tem como objetivo a separação do código segundo a sua importância para a aplicação, permitindo que o programador encapsule o código secundário em módulos separados do restante da aplicação.

3.1.4 Holoparadigma

O Holoparadigma (abreviadamente Holo) foi proposto como um modelo de desenvolvimento para sistemas distribuídos tradicionais [Barbosa 2002, Barbosa 2005]. Um *blackboard* (chamado história) implementa o mecanismo de coordenação. Uma nova entidade de programação (chamada ente) organiza níveis encapsulados de entes e histórias, e cria uma abstração de contexto. Há 2 tipos de entes: ente elementar, que é um ente atômico sem níveis de composição e ente

composto, que é um ente composto por outros entes.

Um ente elementar é organizado em três partes: interface, comportamento e história. A interface descreve as possíveis interações entre entes; o comportamento contém ações, que implementam a funcionalidade dos entes enquanto que a história é um espaço de armazenamento compartilhado de um ente.

Um ente composto (figura 3.1a) tem a mesma organização, mas também pode ser composto de outros entes (que são chamados de entes componentes). Cada ente tem sua história encapsulada. A história de um ente composto é também compartilhada com seus entes componentes. Desta forma, vários níveis de história encapsulada podem existir.

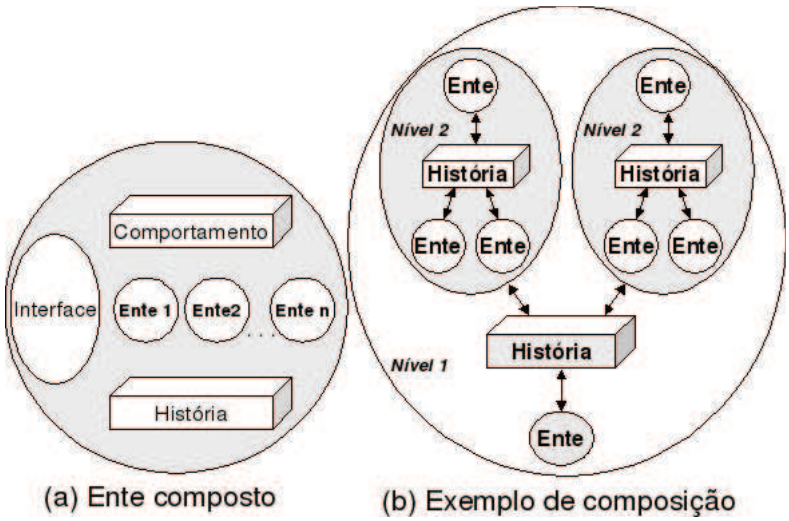


Figura 3.1: Organização dos Entes

Entes usam histórias em níveis de composição específicas. Por exemplo, a figura 3.1b mostra 2 níveis de história encapsulada em um ente com três níveis de composição. Ambos comportamento e interface foram omitidas para simplificar.

A HoloLinguagem [Barbosa e Geyer 2001] é uma linguagem baseada nos conceitos do Holoparadigma. A linguagem suporta mobilidade lógica, concorrência e adaptação dinâmica.

3.1.5 Comparação entre os modelos

A tabela 3.1 contém as características dos modelos de programação pesquisados. São elas:

- **Entidade elementar:** identifica o conceito que expressa a principal abstração do modelo;
- **Contextos:** identifica se o modelo suporta a definição e uso de contextos;
- **Sensibilidade ao contexto:** identifica se o modelo tem mecanismos que auxiliam na detecção das alterações ocorridas nos contextos;
- **Adaptação ao contexto:** identifica se o modelo tem suporte à adaptação ao contexto;
- **Mobilidade de código:** identifica se o modelo tem suporte à mobilidade de código, indicando o seu tipo (fraca ou forte);
- **Concorrência:** identifica se o modelo suporta à exploração de fluxos de execução concorrentes.

Tabela 3.1: Comparativo entre os modelos

	COP	**SOP**	**AOP**	**Holo**
Entidade elementar	Contexto	Assunto	Aspecto	Ente
Contextos	X	X	X	X
Sensibilidade ao contexto	X			
Adaptação ao contexto	X	X	X	
Mobilidade de código				X
Concorrência				X

Com base nas características dos modelos pesquisados, foi possível identificar algumas particularidades. São elas:

- a adaptação provida pela AOP é estática, ou seja, são realizadas durante a compilação dos programas;
- a adaptação provida por COP e SOP são dinâmicas, ou seja, são realizadas durante a execução;

- apenas COP implementa sensibilidade ao contexto;
- apenas Holo implementa mobilidade de código, suportando mobilidade de código forte.

3.2 Ambientes de computação ubíqua

Diversos ambientes para o desenvolvimento de aplicações móveis e ubíquas têm sido propostos. Estes ambientes exploram em maior ou menor grau as características da computação ubíqua. As próximas subseções tratam de sete ambientes: *Context Toolkit*, UbiHolo, ISAM, Continuum, One.World, Gaia e Aura. Estes ambientes foram selecionados por apresentarem idéias inovadoras relacionadas ao usuário, à representação de contexto, a dispositivos heterogêneos e à mobilidade.

3.2.1 *Context Toolkit*

O *Context Toolkit* [Dey 2001] é um *framework* para facilitar o desenvolvimento e a implantação de aplicações sensíveis ao contexto, provendo uma solução reutilizável para manipulação de contextos. É implementado como uma biblioteca de código para ser utilizada em outras linguagens.

No *Context Toolkit* contexto são informações do ambiente que fazem parte do ambiente operacional de uma aplicação e que podem ser detectadas pela aplicação. O *Context Toolkit* consiste de *context widgets* e de uma infra-estrutura distribuída que hospeda estes *widgets*. *Context widgets* são componentes de software que possibilitam que aplicações acessem informações de contexto, enquanto abstraem detalhes sobre como estes dados de contexto são obtidos.

Os serviços fornecidos pelo *Context Toolkit* são:

- encapsulamento de sensores;
- acesso a dados de contextos através de uma API de rede;
- abstração de dados de contexto através de interpretadores;

- compartilhamento de dados de contexto através de uma infra-estrutura distribuída;
- armazenamento de dados de contexto, incluindo o histórico;
- controle de acesso (privacidade).

3.2.2 UbiHolo

Ubiquitous Holo (abreviadamente UbiHolo) é uma plataforma que suporta a execução distribuída de programas sensíveis a contexto no Holo. O ambiente do UbiHolo [BARBOSA et al. 2007] é baseado no Holoparadigma (citado na seção 3.1.4).

O ambiente de execução do UbiHolo é composto pela HoloVM, HNS e HoloGo. A **HoloVM** é a máquina virtual responsável por executar os programas Holo. Ela cria uma camada de abstração entre os programas e o hardware, possibilitando que programas Holo executem em qualquer plataforma que a HoloVM tenha sido portada. **HNS** implementa uma visão distribuída dos entes, possibilitando acesso distribuído a história. Ele informa as HoloVMs sobre a localização de um ente em particular. A meta principal do HNS é a execução distribuída dos programas Holo através de várias HoloVMs. Por sua vez, o **HoloGo** suporta mobilidade de código entre *hosts*.

3.2.3 ISAM

ISAM [Augustin 2004] integra os conceitos de sensibilidade ao contexto, *grid* e computação móvel. A idéia do ISAM é construir uma infra-estrutura para computação pervasiva, integrando uma linguagem de programação e um *middleware* que suporte sua execução. Diferentemente de outras propostas, ISAM foca no desenvolvimento de aplicações ao invés de focar no ambiente e serviços. Em função disso, o projeto define um modelo, uma linguagem e o suporte em tempo de execução para construir e executar aplicações pervasivas. Programadores podem desenvolver aplicações utilizando Java. Para futuramente facilitar o desenvolvimento de aplicações pervasivas, ISAMadapt foi definido como um *framework* para uma linguagem de programação. Ele provê formas de expressar

adaptação dinâmica e sensibilidade ao contexto durante o desenvolvimento. ISAMadapt utiliza conceitos do Holoparadigma, citado na seção 3.1.4.

3.2.4 Continuum

O Continuum [Costa 2009] é uma infra-estrutura de software para a computação ubíqua, sensível ao contexto, baseada na arquitetura orientada a serviços (SOA), que facilita o desenvolvimento de aplicações ubíquas. A proposta integra *middleware* e *frameworks* e foca nos principais desafios apresentados pela computação ubíqua [Costa, Yamin e Geyer 2008].

O foco particular é a sensibilidade ao contexto, ou seja, a percepção de características relacionadas aos usuários e ao entorno. São considerados os recursos disponíveis no ambiente e é mantida a história dos dados de contexto. Além disso, é proposta a representação do contexto para promover raciocínio e compartilhamento de conhecimento, empregando uma ontologia.

3.2.5 One.World

One.World [Grimm 2004] é um projeto desenvolvido por pesquisadores da Universidade de Washington e da Universidade de Nova Iorque. A arquitetura proposta provê um *framework* integrado e geral para o desenvolvimento de aplicações ubíquas. A arquitetura inclui um conjunto de serviços, entre os quais serviços de descoberta, *checkpointing*, migração e replicação. Estes serviços simplificam a tarefa de lidar com mudanças constantes. As aplicações que executam nos dispositivos móveis utilizam tais serviços. Além disso, é definido um modelo de programação que deve ser adotado pelos desenvolvedores de aplicações One.World. Neste modelo, a distribuição deve ser explícita. O projeto foi implementado na linguagem Java, assim as aplicações também devem ser desenvolvidas nesta linguagem. Neste caso, o desenvolvedor precisa estender as classes do One.World de maneira a utilizar as funcionalidades oferecidas pelo *framework* em sua aplicação. Quando a plataforma da aplicação for um PDA (*Personal Digital Assistant*), a implementação deve, ainda, observar limitações da API Java e da própria API do *framework*, uma vez que essas são mais enxutas em PDAs do que em *desktops*.

3.2.6 Gaia

O ambiente Gaia [Román et al. 2002] foi desenvolvido na Universidade de Illinois. O Gaia tem como objetivo criar uma infra-estrutura de *middleware* distribuído que coordena serviços de software e dispositivos, que se utilizam de uma rede heterogênea, contidos em um espaço físico. Em outras palavras, o Gaia pode ser visto como um meta sistema operacional que coordena entidades de software e dispositivos heterogêneos presentes em um espaço físico. O Gaia possibilita a consulta por serviços, utiliza-se dos recursos existentes para acessar e manipular o contexto atual e provê um *framework* para desenvolver aplicações centradas no usuário, consciente do espaço físico e para múltiplos dispositivos. Estas aplicações podem ser implementadas em Java, C++ ou Lua. O projeto define, ainda, um novo conceito chamado de *Active Space*. Enquanto um espaço físico tem uma série de características bem definidas (possui fronteiras, possui objetos, dispositivos conectados a redes e usuários desempenhando diferentes atividades), um *Active Space* é um espaço físico coordenado por uma infra-estrutura de software. Porém são restritos a espaços pré-definidos. Baseado em contexto, este software potencializa a habilidade do usuário para interagir e configurar seu ambiente virtual e físico de forma consistente.

3.2.7 Aura

Aura [Garlan et al. 2002] é um projeto que vem sendo desenvolvido por pesquisadores da Universidade de Carnegie Mellon. Ele é caracterizado por ser um ambiente para a computação ubíqua que envolve comunicação sem-fio, computadores portáteis que podem ser vestidos (*wearable*) e espaços inteligentes. Seu foco é reduzir a distração do usuário com fatores externos à aplicação, uma vez que considera a atenção do usuário como recurso escasso. O conceito que o projeto introduz diz que o usuário passa a ter uma "aura pessoal". Neste sentido, tem-se que quando o usuário entra em um novo ambiente, sua aura encarrega-se de conseguir recursos necessários para que o usuário execute a sua tarefa. Percebe-se, assim, que o sistema busca a pró-atividade (antecipa requisições) e o auto-ajuste (adaptação de uso de recursos e de desempenho). Neste contexto, são exemplos de tarefas: escrever um artigo, uma apresentação ou mesmo comprar uma casa,

onde cada uma destas tarefas pode envolver uma grande quantidade de fontes de informação e aplicações.

Aura possui uma API em Java e C para uso dos seus serviços. Porém os serviços existentes não podem ser ampliados e a adaptação é apenas do ambiente em relação aos recursos disponíveis.

3.2.8 Comparação entre os ambientes

A tabela 3.2 contém as características dos ambientes pesquisados. São elas:

- **Serviços:** indica que a solução suporta a publicação, busca e execução de serviços;
- **Contextos:** identifica que a solução suporta a definição e uso de contextos;
- **Sensibilidade ao contexto:** indica que a solução obtém informações relevantes através de sensores (hardware ou software) que permitem a adaptação do comportamento dos aplicativos ao seu contexto;
- **Adaptação ao contexto:** indica que a solução modifica o seu comportamento baseada nas informações de contexto percebidas, adaptando-se para a situação atual;
- **Mobilidade de código:** indica que a solução suporta a mobilidade de código;
- **Concorrência:** indica que a solução suporta a exploração de fluxos de execução concorrentes;
- **Suporta um modelo:** indica que a solução envolve um novo modelo de programação;
- **Linguagem de programação:** indica que a solução possui uma linguagem de programação própria;
- **Máquina virtual:** indica que a solução possui uma máquina virtual própria para a execução dos aplicativos.

Tabela 3.2: Características dos ambientes de computação ubíqua

	Context Toolkit	**UbiHolo**	**ISAM**	**Continuum**	**One.World**	**Gaia**	**Aura**
Serviços				X			
Contextos	X	X	X	X	X	X	
Sensibilidade ao contexto	X			X	X	X	X
Adaptação ao contexto	X			X	X	X	X
Mobilidade de código		X	X	X	X		
Concorrência		X	X	X	X	X	X
Suporta um paradigma		X	X				
Linguagem de programação		X	X				
Máquina virtual		X					

Com base nas características dos modelos pesquisados, foi possível identificar algumas particularidades. São elas:

- Apenas Continuum tem suporte a serviços e a contextos que compartilham informações entre várias aplicações;
- UbiHolo e ISAM possuem suporte parcial ao item "Contextos" porque o ambiente suporta vários contextos, mas um Ente somente pode estar em um contexto em determinado momento;
- Apenas UbiHolo utiliza uma máquina virtual própria para a execução dos seus aplicativos;
- Apenas UbiHolo e ISAM definem uma nova linguagem de programação;
- Aura tem suporte parcial no item "Adaptação do contexto" porque a adaptação é apenas do ambiente em relação aos recursos disponíveis.

3.3 Considerações sobre o capítulo

Este capítulo abordou trabalhos relacionados à computação ubíqua. Foram identificadas as características que, apesar de relevantes, estão presentes em alguns destes trabalhos apenas: compartilhamento de informações através dos contextos, serviços, sensibilidade ao contexto, adaptação ao contexto, mobilidade de código e concorrência. Estas características fazem parte das contribuições propostas pelo UOP, descrito no próximo capítulo.

Capítulo 4

Ubiquitous Oriented Programming

Este capítulo apresenta o *Ubiquitous Oriented Programming* (abreviadamente UOP), um novo modelo de programação orientado ao desenvolvimento de aplicativos ubíquos. A seção 4.1 apresenta os conceitos básicos do UOP; a seção 4.2 apresenta os aspectos herdados e estendidos da OOP e serviços; a seção 4.3 apresenta os aspectos relevantes aos contextos que armazenam informações privadas; a seção 4.4 apresenta os aspectos relevantes aos contextos que armazenam informações contextuais compartilhadas; a seção 4.5 apresenta como os contextos hierárquicos podem ser utilizados na descoberta de informações; a seção 4.6 apresenta como a sensibilidade ao contexto é utilizada para auxiliar na adaptação das aplicações; a seção 4.7 apresenta os conceitos de mobilidade enquanto que a seção 4.8 apresenta os conceitos de concorrência; por fim, a seção 4.9 contém as considerações finais deste capítulo.

Salienta-se que as seções 4.3, 4.4, 4.6, 4.7 e 4.8 apresentam os diferencias deste modelo em relação a OOP.

4.1 Conceitos básicos

UOP é um modelo de programação que introduz um conjunto de diretrizes para facilitar o desenvolvimento de aplicativos ubíquos. Para tal, utiliza o conceito de serviços, herdado dos *Web Services* [Austin Abbie Barbir e Garg 2002], e os seguintes conceitos da OOP: classes, objetos, propriedades, métodos, mensagens,

sobrecarga de métodos, herança e polimorfismo.

A principal contribuição é a integração, em um único modelo orientado para computação ubíqua, dos conceitos herdados da OOP e dos *Web Services*, com os seguintes conceitos: contextos [Dey 2001], sensibilidade ao contexto [Dey 2001, Satyanarayanan 2001], adaptação ao contexto [Costa, Yamin e Geyer 2008, Satyanarayanan 2001, Satyanarayanan 1996], mobilidade forte de código [Fuggetta, Picco e Vigna 1998, Ghezi e Vigna 1997, Thorn 1997, Naseen 2004] e concorrência.

Contexto é qualquer informação que possa ser utilizada para caracterizar a situação das entidades (pessoas, lugares ou objetos) que são consideradas relevantes para a interação entre o usuário e uma aplicação, incluindo o usuário e a aplicação em si. Contexto é tipicamente a localização, identidade e estado das pessoas, grupos e objetos físicos e computacionais [Dey 2001].

Alguns trabalhos associam contextos apenas com a localização. No UOP, os contextos podem estar associados com determinadas localizações, mas o modelo não obriga isso. Desta forma, **contexto físico** é um contexto que está associado a uma localização, enquanto que **contexto simbólico** é um contexto onde a localização é irrelevante. Um exemplo de contexto físico seria "usuários próximos", sendo "próximos" medido como raio, em metros. Um exemplo de contexto simbólico seria "Grupo de pesquisa sobre compiladores e linguagens de programação".

Os contextos permitem a colaboração e a interação entre os elementos através das informações que compartilham. Quanto a visibilidade das informações, existem dois tipos de contextos: privados e públicos, que serão explicados, respectivamente, nas seções 4.3 e 4.4.

4.2 Extensões da Orientação a Objetos e Serviços

Vários aspectos da OOP e serviços foram utilizados, e esta seção aborda como foram contemplados no UOP.

4.2.1 Entidades

A entidade é a principal abstração do UOP, sendo similar ao conceito de classes. Uma entidade (figura 4.1) define quais estados (propriedades) ela é capaz de manter e o seu comportamento (métodos e serviços). Estes conceitos serão abordadas no decorrer deste capítulo.

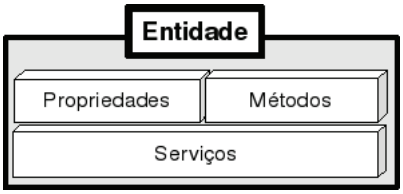


Figura 4.1: Representação de uma Entidade

Os programas são compostos pela descrição de suas entidades. Uma entidade é criada no nível de modelagem e programação, e define o padrão que é utilizado durante a criação de elementos através da instanciação.

O **elemento** é a instância de uma entidade, sendo similar ao conceito de objeto. Um programa em execução é composto de elementos que armazenam estados através de suas propriedades, relacionando-se com outros elementos através do envio de mensagens e reagindo as mensagens recebidas.

4.2.2 Métodos

Os métodos definem as habilidades das entidades, possuindo entradas (parâmetros) e saídas (valores de retorno). Possuem dois níveis de visibilidade: ***private*** (acessíveis somente na entidade que definiu o método) e ***public*** (acessíveis nas outras entidades).

O ato de invocar um método de um elemento é conhecido como mensagem. Uma entidade pode ter vários métodos e a utilização de um método afeta apenas um elemento em particular.

Através dos métodos uma entidade **encapsula** seus dados impedindo o acesso direto as suas propriedades. As entidades disponibilizam externamente apenas os métodos que alteram estes estados.

Os métodos podem ser **sobrecarregados**, ou seja, utiliza-se o mesmo nome para métodos com operações ou funcionalidades distintas. Quando um elemento

recebe uma mensagem, a assinatura distingue qual o método que deve ser invocado. A assinatura é composta pelo nome do método e o tipo de seus argumentos.

4.2.3 Serviços

Os serviços estendem o conceito de métodos da OOP, integrando-o ao conceito de serviços existentes nos *Web Services*. Assim como os métodos, os serviços possuem entradas, saídas e podem ser sobrecarregados.

Porém, os serviços possuem um diferencial em relação aos métodos: serviços podem ser compartilhados nos contextos. Os elementos, durante sua execução, decidem quais serviços irão compartilhar, e em quais contextos. Após serem compartilhados, os serviços podem ser encontrados e invocados pelos outros elementos em execução nos aplicativos.

4.2.4 Propriedades

As propriedades representam as características de uma entidade. Os possíveis valores que cada propriedade recebe denomina-se estados.

A visibilidade das propriedades sempre é ***private***, ou seja, somente são acessíveis nas sub-rotinas [1] da entidade. A única forma de acessar as propriedades, fora do escopo da entidade, é através das suas sub-rotinas.

4.2.5 Herança entre entidades

Herança é o mecanismo pelo qual uma entidade (sub-entidade) pode estender outra entidade (super-entidade), herdando seu comportamento (métodos e serviços). A herança ocorre durante o desenvolvimento e permite que a sub-entidade defina e/ou redefina o seu comportamento.

4.2.6 Métodos polimórficos

Polimorfismo é o princípio pelo qual duas ou mais sub-entidades, derivadas de uma mesma super-entidade, podem invocar métodos que têm a mesma assinatura, mas comportamentos distintos, especializados para cada sub-entidade. A decisão

[1] Métodos e serviços

sobre qual método deve ser executado, de acordo com o tipo da sub-entidade, é tomada em tempo de execução.

4.2.7 Clonagem de elementos

A clonagem de elementos corresponde à criação de clones suportada por algumas linguagens orientadas a objetos [Flanagan 2000]. A clonagem ocorre durante a execução e cria um clone de determinado elemento, copiando deste o estado das suas propriedades e herdando seus métodos e serviços. A partir da clonagem, o clone não mantém qualquer ligação com o elemento que foi clonado, e alterações posteriores no elemento clonado não são refletidas no clone.

4.3 Contextos privados

Os contextos privados armazenam informações privadas de uma aplicação. São implicitamente criados na inicialização do aplicativo, e nunca são destruídos. Por conterem informações privadas, apenas a aplicação que os criou acessa estas informações.

As quatro categorias de informações contextuais definidas por Dey [Dey 2001] são implicitamente definidas como contextos privados. São elas:

- **Localização** (*location*): armazena informações sobre a localização simbólica e física;
- **Identidade** (*identity*): armazena informações de identificação como nome, usuário, CPF, RG, números de cartão de crédito, número de série de equipamentos, senhas, entre outros;
- **Tempo** (*time*): armazena informações relativas a data e hora;
- **Atividade** (*activity*): armazena informações sobre o que está ocorrendo no momento como "Indo fazer compras na livraria".

Cabe ressaltar que os contextos privados mantém as informações consideradas relevantes a aplicação. Se a aplicação representa uma pessoa, mantém

informações desta pessoa; se a aplicação representa um lugar, então mantém informações sobre este lugar; da mesma forma, se representa um objeto, mantém informações sobre este objeto.

4.4 Contextos públicos

Os contextos públicos são formados dinamicamente durante a execução das aplicações e armazenam as informações contextuais compartilhadas pelos elementos destas aplicações. Por conterem informações públicas, qualquer elemento de qualquer aplicação pode acessar estas informações.

Um contexto público (figura 4.2) permite a colaboração e a interação entre elementos através do compartilhamento de três informações contextuais: conteúdos, serviços e membros, explicados nas próximas subseções.

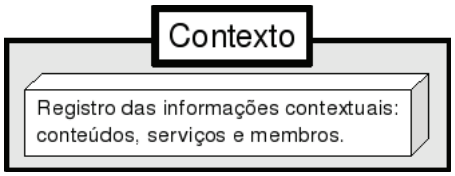


Figura 4.2: Representação de um Contexto Público

4.4.1 Membros

Membros são os elementos que fazem parte de um contexto público. Durante sua execução, um elemento pode ser membro de vários contextos simultaneamente. Existem duas operações que atuam sobre os membros: associação e desassociação de membros em um contexto. A **associação** faz com que determinado elemento faça parte da lista de membros de um contexto enquanto que a **desassociação** faz com que determinado elemento saia da lista de membros de um contexto.

A partir de um contexto, os elementos conseguem identificar as informações contextuais existentes. Quando um elemento necessita de um conteúdo ou serviço, ele tenta obtê-lo dos contextos. Da mesma forma, quando necessita saber a lista dos membros, ele a obtém do contexto.

A figura 4.3 contém um exemplo com a relação entre contextos e seus

membros. Neste exemplo existem dois contextos (*Unisinos* e *UFRGS*) e quatro elementos (*E1*, *E2*, *E3* *E4*). Nas etapas 1 e 2, os elementos *E1* e *E2* associam-se ao contexto *UNISINOS* e compartilham os conteúdos e serviços que acharem pertinentes ao contexto; nas etapas 3 e 4, o elemento *E3* associa-se aos contextos *UNISINOS* e *UFRGS*, e compartilha suas informações nestes contextos; por fim, na etapa 5, o elemento *E4* associa-se e compartilha suas informações no contexto *UFRGS*. Os elementos *E1* e *E2* são membros do contexto *Unisinos*; o elemento *E3* é membro dos contextos *Unisinos* e *UFRGS*; por sua vez, o elemento *E4* é membro apenas do contexto *UFRGS*.

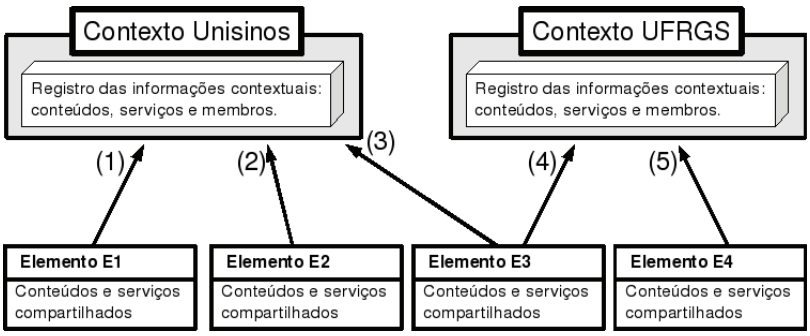


Figura 4.3: Relação entre contextos e seus membros

Salienta-se que os elementos não necessitam serem membros do contexto para terem acesso às informações contextuais. Na figura 4.3, apesar de serem membros apenas do contexto Unisinos, os elementos E1 e E2 acessam as informações existentes no contexto UFRGS. Da mesma forma, apesar de *E4* ser membro apenas do contexto UFRGS, acessa as informações existentes no contexto UNISINOS.

4.4.2 Conteúdos compartilhados

Os conteúdos são compartilhados nos contextos pelos elementos. Um elemento pode disponibilizar vários conteúdos em vários contextos. A decisão de que conteúdos serão publicados ou removidos, e em quais contextos, é tomada dinamicamente durante a execução dos elementos. A visibilidade dos conteúdos sempre é *public*, ou seja, são acessíveis por qualquer elemento.

As operações existentes sobre os conteúdos podem ser bloqueantes ou não bloqueantes. Uma **operação bloqueante** estabelece um ponto de sincronismo,

isto é, a execução da aplicação continua somente após a execução da operação. Uma **operação não bloqueante** obtém sucesso (retorna o resultado) apenas se for possível executar a operação; caso não obtenha sucesso, a operação indica que não foi possível executar a operação, mas não bloqueia a execução da aplicação.

Existem oito operações que manipulam os conteúdos dos contextos. São elas:

- *Publica*: sempre não bloqueante, publica determinado conteúdo no contexto;
- *Lista*: sempre não bloqueante, lista os conteúdos do contexto;
- *Busca bloqueante*: busca determinado conteúdo no contexto. Caso não exista, bloqueia a aplicação até que o conteúdo esteja disponível;
- *Busca não bloqueante*: busca determinado conteúdo no contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução;
- *Obtém bloqueante*: busca e remove determinado conteúdo do contexto. Caso não exista, bloqueia a aplicação até que o conteúdo esteja disponível;
- *Obtém não bloqueante*: busca e remove determinado conteúdo do contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução;
- *Remove bloqueante*: remove determinado conteúdo do contexto. Caso não exista, bloqueia a aplicação até que o conteúdo esteja disponível;
- *Remove não bloqueante*: remove determinado conteúdo do contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução.

A operação "obtém", além de buscar determinado conteúdo, também o remove do contexto.

A arquitetura para compartilhamento de conteúdos (figura 4.4) é baseada na interação entre três componentes principais: Elemento Provedor de Conteúdos, Contexto e Elemento Consumidor de Conteúdos. **Provedor** é o elemento que publica conteúdos no contexto enquanto que **Consumidor** é o elemento que utiliza conteúdos do contexto. O provedor e o consumidor interagem utilizando as operações que manipulam os conteúdos. Na etapa 1, o provedor publica conteúdos no contexto; na etapa 2, o consumidor busca conteúdos no contexto, solicitando-os

diretamente ao provedor na etapa 3. Esta arquitetura facilita a tarefa de publicar e encontrar os conteúdos compartilhados nos contextos, possibilitando que os conteúdos de um contexto estejam distribuídos entre N provedores.

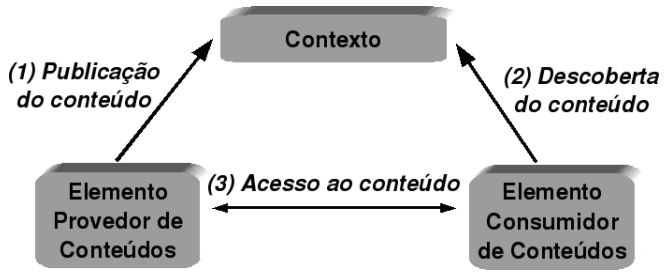


Figura 4.4: Arquitetura para compartilhamento de conteúdos

Um conteúdo, após ser publicado, pode ser lido, alterado e removido por qualquer elemento. Este conteúdo fica disponível no contexto até que algum elemento decida removê-lo ou até que o elemento provedor finalize sua execução.

4.4.3 Serviços compartilhados

Os serviços são compartilhados nos contextos pelos elementos. Um elemento pode disponibilizar vários serviços em vários contextos. A decisão de que serviços serão publicados ou removidos, e em quais contextos, é tomada dinamicamente durante a execução dos elementos. A visibilidade dos serviços sempre é *public*, ou seja, são acessíveis por qualquer elemento.

Assim como ocorre com os conteúdos, as operações que manipulam os serviços podem ser bloqueantes ou não bloqueantes. Existem oito operações:

- *Publica*: sempre não bloqueante, publica determinado serviço no contexto;
- *Lista*: sempre não bloqueante, lista os serviços do contexto;
- *Busca bloqueante*: busca determinado serviço no contexto. Caso não exista, bloqueia a aplicação até que o serviço esteja disponível;
- *Busca não bloqueante*: busca determinado serviço no contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução;

- *Executa bloqueante*: busca e executa determinado serviço do contexto. Caso não exista, bloqueia a aplicação até que o serviço esteja disponível;
- *Executa não bloqueante*: busca e executa determinado serviço do contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução;
- *Remove bloqueante*: remove determinado serviço do contexto. Caso não exista, bloqueia a aplicação até que o serviço esteja disponível;
- *Remove não bloqueante*: remove determinado serviço do contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução.

Assim como em *Web Services* [Austin Abbie Barbir e Garg 2002], a arquitetura para compartilhamento de serviços (figura 4.5) é baseada na interação entre três componentes principais: Elemento Provedor de Serviços, Contexto e Elemento Consumidor de Serviços. **Provedor** é o elemento que publica serviços no contexto enquanto que **Consumidor** é o elemento que busca serviços no contexto. O provedor e o consumidor interagem utilizando as operações que manipulam os serviços. Na etapa 1, o provedor publica serviços no contexto; na etapa 2, o consumidor busca serviços no contexto, solicitando sua execução diretamente ao provedor na etapa 3. Esta arquitetura facilita a tarefa de publicar, encontrar e executar os serviços compartilhados nos contextos, possibilitando que os serviços de um contexto estejam distribuídos entre N provedores.

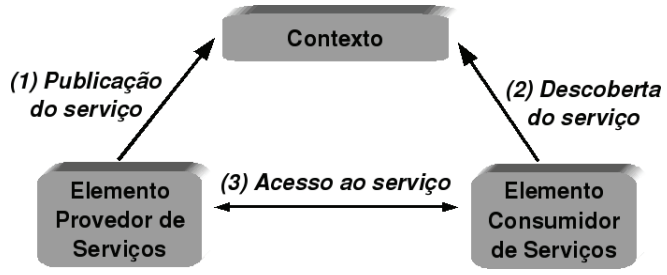


Figura 4.5: Arquitetura para compartilhamento de serviços

A figura 4.6 contém um exemplo onde os elementos "Elemento 1" e "Elemento 2" compartilham serviços no contexto "Contexto 1". Neste exemplo "Elemento 1" mantém os serviços *S1_1* e *S1_2*, mas decide compartilhar neste contexto (etapa 1) apenas o serviço *S1_1*; por sua vez, "Elemento 2" compartilha

neste contexto (etapa 2) seu único servico (*S2_1*). "Elemento 1", durante a execução do seu método *M1_1*, busca no contexto (etapa 3) quem disponibiliza o serviço *S2_1*, solicitando sua execução diretamente ao "Elemento 2" (etapa 4).

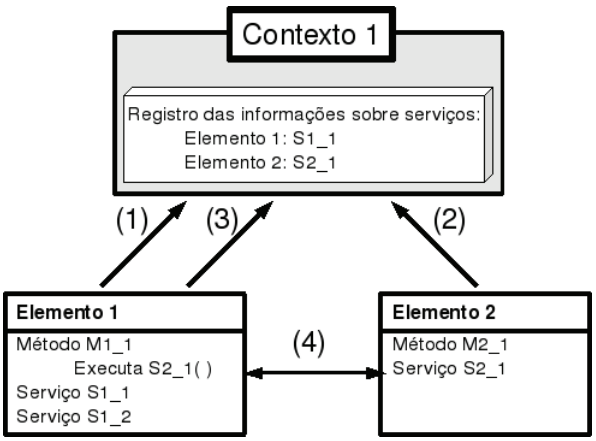


Figura 4.6: Elementos compartilhando serviços em um contexto

Um serviço, após ser publicado, fica disponível no contexto até que o elemento provedor decida removê-lo ou finalize sua execução. Salienta-se que um serviço somente pode ser removido pelo elemento provedor.

Ao publicar um serviço, o provedor publica também as características deste serviço, possibilitando que o consumidor determine o serviço que melhor atende suas necessidades.

Assim como os métodos, os serviços possuem entradas (parâmetros) e saídas (valores de retorno). Conforme visto na seção 2.4, um serviço deve funcionar de forma independente do estado de outros serviços (*stateless*), possibilitando assim que sejam executados de forma independente. Através da coreografia, os consumidores podem seqüenciar os serviços, ou seja, executá-los em vários fluxos (*pipelines*) para executar a lógica da aplicação. Com isto em mente, os resultados de um serviço devem ser obtidos através dos seus valores de retorno.

Assim como nos métodos, os serviços também podem ser sobrecarregados. Um elemento provedor, ao publicar um serviço, utiliza a assinatura completa deste serviço. A assinatura é composta pelo nome do serviço e o tipo de seus argumentos. Da mesma forma, ao solicitar a execução de um serviço, o elemento consumidor informa a assinatura completa do serviço.

A figura 4.7 contém um exemplo de sobrecarga de serviços. Na etapa 1, "Elemento 1" disponibiliza, no contexto "Contexto 1", o serviço *S1* sobrecarregado em duas versões: *S1:int*, com um argumento do tipo *int*, e *S1:string*, com um argumento do tipo *string*. "Elemento 2", durante a execução do seu método *M2*, verifica no contexto quem disponibiliza os serviços *S1:int* e *S1:string* (etapa 2), solicitando sua execução diretamente ao "Elemento 1" (etapa 3).

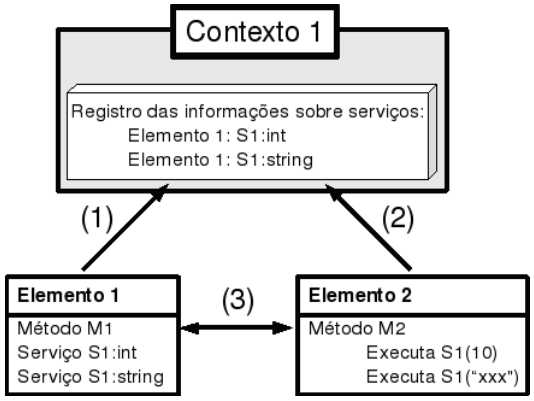


Figura 4.7: Sobrecarga de serviços

4.5 Contextos hierárquicos

A convenção de nomes da *Internet* é baseada no conceito de domínio. O nome de um domínio consiste de uma concatenação de um ou mais "nomes comuns" [RFC 819]. O conjunto de domínios forma uma hierarquia.

Um sistema de domínios é uma estrutura em árvore global que contém domínios de alto nível. Os domínios de alto nível são subdivididos em domínios de segundo nível (subdomínios). Os domínios de segundo nível podem ser subdivididos em um terceiro nível, e assim sucessivamente [RFC 920].

Os contextos hierárquicos são organizados como um sistema de domínios; porém, para refletir as idéias propostas no UOP, os domínios e subdomínios são tratados como contextos. Os contextos hierárquicos possibilitam que, a partir de um determinado contexto, contextos e informações contextuais mais especializadas possam ser descobertas. Como os contextos e as informações mais especializadas encontram-se nos níveis mais baixos da hierarquia, a busca realizada a partir de

um contexto percorre todos os níveis abaixo deste, e só termina quando é atingido o último nível.

Como as informações existentes nesta hierarquia são acessíveis por todos os elementos, os contextos existentes, bem como as informações contextuais destes contextos, são acessíveis a qualquer elemento.

A figura 4.8 contém um exemplo que reflete esta estrutura em árvore. O contexto inicial sempre é o "/", e fica localizado no nível um da árvore. Neste exemplo, no nível dois, tem-se o contexto *Universidades*; no nível três, existem os contextos *Unisinos* e *UFRGS*; no nível quatro, a partir do contexto *Unisinos*, é possível identificar os contextos *Laboratórios* e *Disciplinas*; por sua vez, no mesmo nível, a partir do contexto *UFRGS*, é possível identificar o contexto *Disciplinas*; e assim sucessivamente.

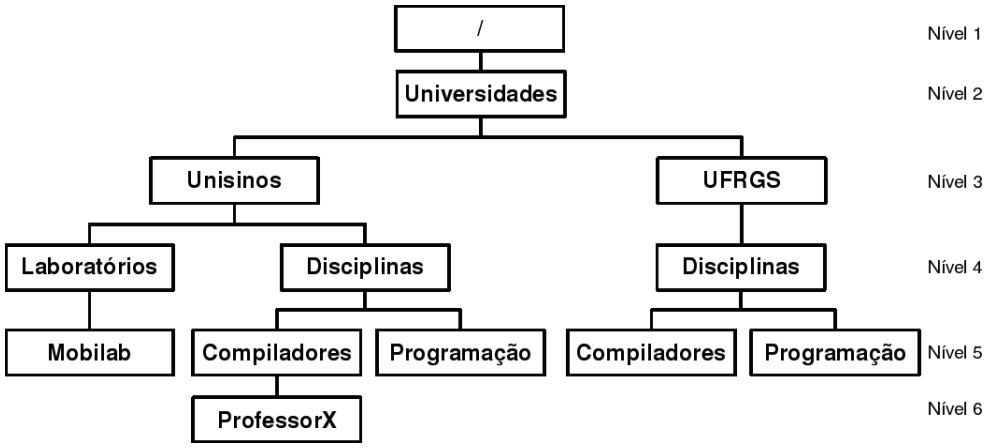


Figura 4.8: Organização dos contextos hierárquicos

O caminho de um contexto é a informação contendo os contextos que devem ser percorridos, a partir do contexto inicial, para acessar o contexto requerido. Neste exemplo, o caminho do contexto *Mobilab* é "/Universidades/Unisinos/Laboratórios/Mobilab".

Esta estrutura em árvore facilita a organização dos contextos em uma hierarquia. Neste exemplo, o contexto *Mobilab* está abaixo do contexto *Laboratórios*, que por sua vez está abaixo do contexto *Unisinos*, e assim sucessivamente até o contexto inicial. O contexto *Unisinos* faz parte do contexto *Universidades*, mas também é o contexto pai do contexto *Laboratórios*, onde reside

o contexto *Mobilab*.

Nesta hierarquia em contextos, quanto mais baixo o nível do contexto, mais especializada são suas informações. Por exemplo, o contexto *Compiladores*, disponível no nível 5, contém informações oficiais disponibilizadas pela Unisinos sobre a disciplina de compiladores, tais como ementa, livros recomendados, entre outros. Já o contexto *ProfessorX*, disponível no nível 6, contém material adicional disponibilizado pelo professor como notas de aula, trabalhos solicitados, resultado das avaliações, serviços para execução dos trabalhos de aula, entre outros.

A partir do contexto "/Universidades", é possível descobrir os contextos *Unisinos* e *UFRGS*, bem como acessar suas informações contextuais; a partir do contexto "/Universidades/Unisinos/Disciplinas/Compiladores", é possível identificar as informações contextuais neste contexto, bem como buscar as informações mais especializadas existentes em "/Universidades/Unisinos/Disciplinas/Compiladores/ProfessorX".

4.6 Adaptação das aplicações

A adaptação é necessária quando existe uma incompatibilidade significante entre o que é fornecido e a demanda real de um recurso. O recurso em questão pode ser largura de banda em conexões sem fio, energia, ciclos computacionais, memória, entre outros [Satyanarayanan 2001].

Através da sensibilidade ao contexto é possível obter as alterações ocorridas nos contextos. A adaptação ao contexto reage à estas alterações, criando um balanceamento dinâmico entre os recursos disponíveis e as necessidades das aplicações, ajustando aspectos das aplicações em virtude de alterações nos ambientes operacionais [Satyanarayanan 2001].

A seção 2.1 esboçou as três estratégias existentes para adaptação. A adaptação consciente da aplicação, utilizada no UOP, une programação com adaptação automática do sistema.

As alterações ocorridas nos contextos são utilizadas para adaptar o comportamento das aplicações (figura 4.9). Uma interface, por exemplo, necessita adaptar-se não somente em função do dispositivo que o usuário está utilizando, mas

também por preferências pessoais do usuário ou fatores externos como iluminação e temperatura.

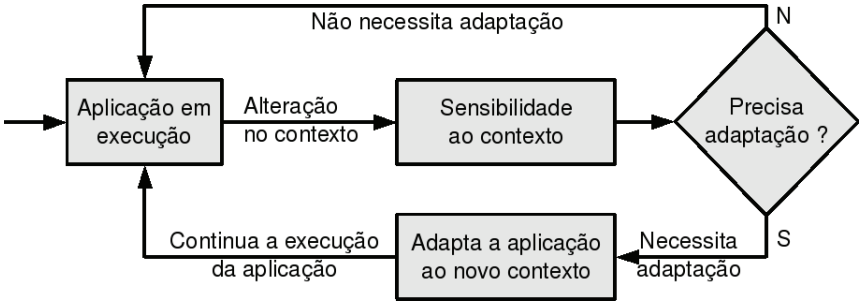


Figura 4.9: Adaptação ao contexto

A adaptação das aplicações ocorre de duas formas: através dos eventos e durante a resolução de nomes.

4.6.1 Adaptação através dos eventos

Na programação dirigida a eventos existem dois componentes: o ***publisher***, que é quem gera eventos, e o ***subscriber***, que é quem quer reagir aos eventos. O *subscriber* registra, a qualquer momento, as ações (métodos) que ele quer executar quando determinados eventos ocorrerem. Durante a execução, o *publisher* pode gerar eventos há qualquer momento, causando a execução dos métodos registrados pelo *subscriber* a este evento.

Eventos podem ser definidos para determinadas situações que podem ocorrer com os elementos bem como com os contextos privados e públicos. Por exemplo, alterações na localização do usuário, na temperatura do corpo, na velocidade da rede e no nível da bateria geram eventos que, caso tenham métodos associados, serão executados.

4.6.2 Adaptação durante a resolução de nomes

Além dos eventos, a adaptação também pode ocorrer durante a resolução de nomes de entidades ou sub-rotinas.

Uma entidade pode ter múltiplas definições, onde cada definição se adapta a contextos específicos. Por este motivo, durante a instanciação de um elemento,

é necessário resolver o nome da entidade, identificando a que melhor se adapta ao contexto atual.

Da mesma forma, uma sub-rotina pode ter múltiplas definições, onde cada definição se adapta a contextos específicos. Por este motivo, durante o recebimento de uma mensagem, é necessário resolver o nome da sub-rotina, identificando a que melhor se adapta ao contexto atual.

4.7 Mobilidade forte de código

A mobilidade possibilita o acesso a dados e aplicativos, independente da localização. O UOP propõe a mobilidade forte de código através da migração [Fuggetta, Picco e Vigna 1998, Ghezi e Vigna 1997, Thorn 1997, Naseen 2004] (seção 2.1). A mobilidade de código possibilita que um aplicativo mova-se para outro dispositivo durante sua execução.

A figura 4.10 contém um exemplo onde uma aplicação move-se para outro dispositivo. Nesta figura existem três aplicativos em execução (*Aplicativo 1*, *Aplicativo 2* e *Aplicativo 3*) e dois dispositivos (*Dispositivo 1* e *Dispositivo 2*). *Aplicativo 1* e *Aplicativo 3* estão em execução no *Dispositivo 1* enquanto que *Aplicativo 2* está em execução no *Dispositivo 2*. Durante sua execução, *Aplicativo 3* move-se, juntamente com os seus elementos, do *Dispositivo 1* para o *Dispositivo 2*. Após a mobilidade, os elementos do *Aplicativo 3* permanecem nos mesmos contextos em que se encontravam anteriormente (neste exemplo, apenas o contexto *Contexto 1*), compartilhando as mesmas informações neste contexto. Para simplificar a figura, os elementos existentes nos aplicativos foram omitidos.

4.8 Concorrência

Um aplicativo, ao ser iniciado, é associado a um fluxo de execução (fluxo de execução principal). Este fluxo inicia e termina sua execução junto com a aplicação.

O UOP suporta o desenvolvimento de aplicativos concorrentes. Para facilitar a implementação destes aplicativos, os métodos podem ser executados como um

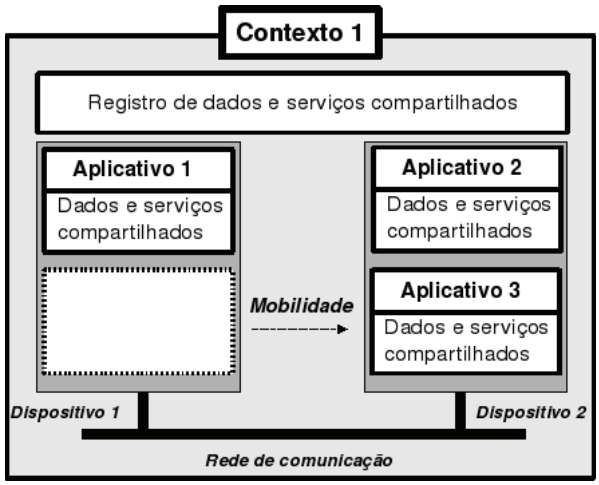


Figura 4.10: Mobilidade de código

fluxo de execução independente. **Métodos concorrentes**, quando executados, criam o seu próprio fluxo de execução, concorrendo assim com o fluxo de execução principal bem como com os fluxos dos outros métodos concorrentes (se existirem).

A figura 4.11 contém um exemplo de métodos concorrentes. Logo após o início da execução, existe apenas o fluxo de execução principal da aplicação. Ao serem invocados pelo fluxo principal, os métodos concorrentes *Metodo1* e *Metodo2* criam seu próprio fluxo de execução. A partir deste momento, existem três fluxos de execução concorrentes: fluxo principal, fluxo do Método1 e fluxo do Método2. Neste exemplo, o fluxo de execução principal aguarda que os métodos concorrentes terminem. A partir deste momento, os fluxos de *Metodo1* e *Metodo2* são extintos, e apenas o fluxo de execução principal continua em execução.

Conforme visto na figura 4.5, os serviços são solicitados por um elemento consumidor, mas executados pelo elemento provedor. Para possibilitar que o elemento provedor execute um serviço, sem interromper suas outras tarefas em execução, os serviços criam o seu próprio fluxo de execução ao serem invocados, concorrendo assim com os outros fluxos existentes.

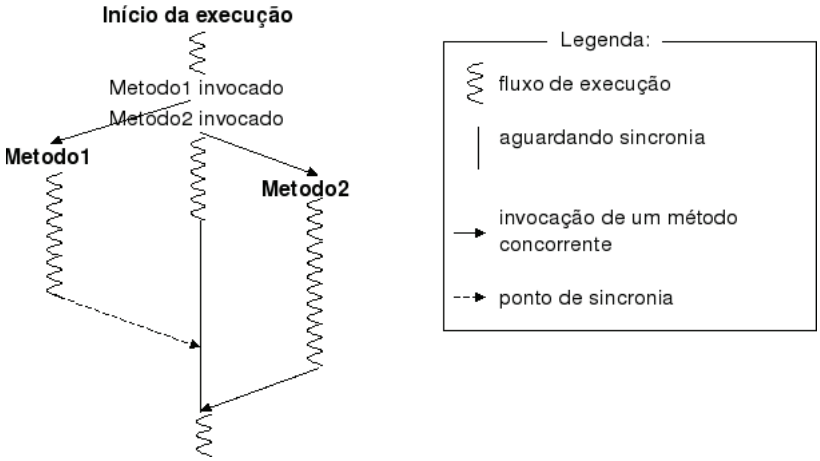


Figura 4.11: Métodos concorrentes

4.9 Considerações sobre o capítulo

Este capítulo abordou o UOP, um modelo orientado ao desenvolvimento de aplicativos ubíquos. UOP herda conceitos da OOP e *Web Services*, integrando-os com os conceitos de contextos, sensibilidade ao contexto, adaptação ao contexto, mobilidade de código e concorrência.

Para que seja possível a exploração dos conceitos do UOP, necessita-se de uma linguagem que suporte o modelo. O próximo capítulo aborda a UbiL, a linguagem que implementa os conceitos propostos no UOP.

Capítulo 5

UbiLanguage

Este capítulo aborda a UbiLanguage (abreviadamente UbiL), uma linguagem de programação que implementa os conceitos propostos no UOP.

UbiL integra conceitos da OOP aos conceitos de contextos, serviços, sensibilidade ao contexto, adaptação ao contexto, mobilidade forte de código e concorrência. Em linguagens de programação tradicionais como C, C++, Java, C#, Python e Ruby, estes conceitos podem ser implementados através de bibliotecas. Em uma linguagem *Ubiquitous Oriented* como a UbiL, a linguagem provê estas abstrações, reduzindo assim o *gap* semântico (figura 2.2) existente entre o domínio ubíquo e a implementação das aplicações. Estas abstrações auxiliam no desenvolvimento de aplicativos ubíquos, proporcionando maior portabilidade e clareza do código.

O capítulo está estruturado da seguinte forma. A seção 5.1 apresenta as características básicas da linguagem; a seção 5.2 apresenta como os recursos existentes nas entidades podem ser explorados; as seções 5.3 e 5.4 apresentam, respectivamente, como os contextos privados e públicos podem ser utilizados; a seção 5.5 apresenta como os contextos hierárquicos podem ser utilizados na descoberta de contextos, serviços e conteúdos; a seção 5.6 apresenta como a adaptação pode ser inserida nas aplicações; as seções 5.7 e seção 5.8 apresentam, respectivamente, como a mobilidade de código e a concorrência podem ser exploradas; a seção 5.9 contém a gramática básica da linguagem; por fim, a seção 5.10 apresenta as considerações finais deste capítulo.

5.1 Características básicas

As principais características da UbiL são:

- é uma linguagem estaticamente tipada, ou seja, existe a definição explícita de entidades, propriedades e variáveis;
- é *case insensitive*, ou seja, a variável "Um"é igual a variável "UM"que é igual a variável "um";
- herdou características sintáticas das linguagens C++, C# e Ruby.

Um programa é formado pela descrição de suas entidades. Uma entidade contém a definição de suas **propriedades** (responsáveis por armazenar o seu estado interno), seus **métodos** (responsáveis por implementar a funcionalidade de uma entidade) e seus **serviços** (funcionalidades compartilhadas com outros elementos).

A figura 5.1 contém a estrutura básica de um programa. A cláusula *import* é utilizada para indicar as bibliotecas de código necessárias para a correta execução do aplicativo. No exemplo, duas bibliotecas são requeridas na linha 1: *xxx* e *yyy*. Além disso, duas entidades são definidas: *Entidade1* (linha 3) e *Entidade2* (linha 17). A *Entidade1* possui duas propriedades (definidas nas linhas 4 e 5), dois métodos (definidos nas linhas 6 e 9) e um serviço (definido linha 12).

As bibliotecas de código disponíveis atualmente são:

- *io*: gerencia a entrada e saída de dados;
- *ncurses*: gerencia a definição e manipulação de interfaces texto;
- *ncurses_menu*: gerencia a criação e uso de menus em conjunto com a biblioteca *ncurses*;
- *datetime*: funções relacionadas a data e hora;
- *tools*: rotinas de uso comum como manipulação de strings, conversão entre texto e numérico, entre outras.

```
1  import xxx,yyy;
2
3  entity Entidade1
4    prop property_type propriedade1;
5    prop property_type propriedade2;
6    method metodo1()
7      // ...
8    end
9    method metodo2()
10     // ...
11   end
12   service service1()
13     // ...
14   end
15 end
16
17 entity Entidade2
18   // ...
19 end
```

Figura 5.1: Estrutura básica de um programa

A figura 5.2 contém um exemplo que define uma entidade. A entidade *exemplo* (linha 1) possui duas propriedades: a propriedade *propriedade1*, do tipo *string*, definida na linha 2, e a propriedade *propriedade2*, do tipo *int*, definida na linha 3. Além disso possui três métodos: *metodo1*, na linha 5, que não recebe argumentos e não tem valores de retorno; *metodo2*, na linha 8, que recebe os argumentos *v1* (*int*) e *v2* (*string*), e não possui valores de retorno; e *metodo3*, na linha 11, que não recebe argumentos e possui um valor de retorno do tipo *int* e outro do tipo *string*. Os colchetes, após a definição de uma sub-rotina, indicam o tipo dos valores retornados pela sub-rotina. Na linha 14 foi definido o serviço *servico1*, que recebe um argumento do tipo *int* e tem um valor de retorno do tipo *int*.

A figura 5.3 contém o exemplo "Olá mundo !!!". Ao iniciar a execução da aplicação, implicitamente é criado o elemento *Start* contendo a instância da entidade *Start*. É no construtor deste elemento (método *constructor*) que inicia a execução dos aplicativos escritos em UbiL. Neste exemplo, o ponto de início da execução é o construtor definido na linha 2. A sentença da linha 3 invoca a função *writeln* para exibir a mensagem "Olá mundo !!!". No término da aplicação, o elemento *Start* é destruído e, se houvesse um destrutor nesta entidade (método *destructor*), ele seria executado.

```
entity exemplo
  prop string propriedade1;
  prop int    propriedade2;

  method metodo1()
    // ...
  end
  method metodo2(int v1, string v2)
    // ...
  end
  method metodo3() [int, string]
    // ...
  end
  service servico1(int v1) [int]
    // ...
  end
end
```

Figura 5.2: Definição de uma entidade

```
entity start
  method constructor()
    io.writeln( "Olá mundo !!!" );
  end
end
```

Figura 5.3: "Olá mundo !!!" em UbiL

A função *writeln* faz parte da biblioteca de *io*, responsável pela entrada e saída de dados. Salienta-se que *io* é a única biblioteca implicitamente requerida nas aplicações, e por este motivo não é necessário explicitar o seu uso através da cláusula *import*.

As estruturas que controlam os laços de repetição podem ser vistas na tabela 5.1 enquanto que as estruturas que controlam os fluxos de execução podem ser vistas na tabela 5.2. Nos exemplos, *value* é uma variável numérica inteira.

Os operadores são utilizados para manipular variáveis, propriedades, parâmetros e constantes. As tabelas 5.3, 5.4, 5.5 e 5.6 apresentam, respectivamente, os operadores aritméticos, de atribuição, relacionais e lógicos. Cada operador contém sua descrição e em quais tipos de dados se aplica.

5.2 Entidades

Esta seção apresenta os recursos existentes nas entidades que foram herdados da OOP, e como podem ser explorados na linguagem.

Tabela 5.1: Laços de repetição

Sintaxe	Exemplo

while (<CONDICAO>)
    <sentencas>
end
	value = 1;
	while(value <= 10)
	    io.writeln(value);
	    value++;
	end

do
    <sentencas>
while (<CONDICAO>);
	value = 1;
	do
	    io.writeln(value);
	    value++;
	while(value <= 10);

for (<init>;<cond>;<increment>)
    <sentencas>
end
	for(value = 0; value <= 10; value++)
	    io.writeln(value);
	end

foreach (<var> in <conjunto>) <sentencas> end	foreach(value in [1..10]) io.writeln(value); end

Tabela 5.2: Fluxos de execução

Sintaxe	Exemplo

if (<condicao>)
    <sentencas>
else
    <sentencas>
end
	value = 10;
	if (value < 100)
	    value = 100;
	else
	    value = value - 1;
	end

switch (<condicao>)
    case test_1:
        <sentencas>
    end
    case test_2:
        <sentencas>
    end
    othersize:
        <sentencas>
    end
end
	value = 1;
	switch(value)
	    case 1:
	        value = 100;
	    end
	    case 2:
	        value = 200;
	    end
	    othersize:
	        value = 999;
	    end
	end

5.2.1 Elementos

Os elementos são criados e destruídos dinamicamente durante a execução da aplicação. A figura 5.4 contém um exemplo onde o elemento *ex*, que é uma instância da entidade *exemplo*, recebe a mensagem *print*. O método *new* cria

Tabela 5.3: Operadores aritméticos

Nome	Sintaxe	Descrição	Tipos aplicáveis
Soma	Termo1+Termo2	Retorna a soma de Termo1 com Termo2	int, real, string
Subtração	Termo1-Termo2	Retorna a subtração de Termo1 com Termo2	int, real, string
Multiplicação	Termo1*Termo2	Retorna a multiplicação de Termo1 com Termo2	int, real
Divisão	Termo1/Termo2	Retorna a divisão de Termo1 com Termo2	int, real
Módulo	Termo1%Termo2	Retorna o módulo de Termo1 com Termo2	int
Incremento	Termo1++	Incrementa o valor de Termo1	int
Decremento	Termo1–	Decrementa o valor de Termo1	int

Tabela 5.4: Operadores de atribuição

Nome	Sintaxe	Descrição	Tipos aplicáveis
Atribuição	Variavel=Expressao	Variável recebe o valor da expressão	todos
Soma	Termo1+=Termo2	Soma o valor de Termo2 em Termo1	int, real, string
Subtração	Termo1-=Termo2	Subtrai o valor de Termo2 em Termo1	int, real, string
Multiplicação	Termo1*=Termo2	Multiplica o valor de Termo2 em Termo1	int, real
Divisão	Termo1/=Termo2	Divide o valor de Termo2 por Termo1	int, real

Tabela 5.5: Operadores relacionais

Nome	Sintaxe	Descrição	Tipos aplicáveis
Idêntico	Termo1==Termo2	Sucesso se os termos forem idênticos	todos
Diferente	Termo1<>Termo2	Negação do operador "=="	todos
Maior	Termo1>Termo2	Verdadeiro se Termo1 maior que Termo2	int, real, string
Menor	Termo1<Termo1	Verdadeiro se Termo1 menor que Termo2	int, real, string
Maior ou Igual	Termo1>=Termo2	Verdadeiro se Termo1 maior ou igual a Termo2	int, real, string
Menor ou Igual	Termo1<=Termo2	Verdadeiro se Termo1 menor ou igual a Termo2	int, real, string

Tabela 5.6: Operadores lógicos

Nome	Sintaxe	Descrição	Tipos aplicáveis
E relacional	Termo1 and Termo2	Verdadeiro se Termo1 e Termo2 são verdadeiros	bool
Ou relacional	Termo1 or Termo2	Verdadeiro se Termo1 ou Termo2 são verdadeiros	bool
Not relacional	not Termo1	Retorna a negação de Termo1	bool

novos elementos a partir de uma entidade. A cláusula *var* define o tipo e o nome de cada variável da sub-rotina. Neste exemplo *Start* cria um elemento (instância) da entidade *exemplo* (linha 9) e o armazena na variável *ex*. Na linha 10 é enviada a mensagem *print* para este elemento. Ao receber esta mensagem, o elemento *ex* encontra e executa o método correspondente (*print*, linha 2).

A tabela 5.7 contém os tipos de dados existentes em UbiL, com sua descrição e os limites impostos para cada tipo.

Um elemento sempre é criado ou destruído por outro elemento; a exceção é o elemento *Start* que é criado implicitamente no início da execução da aplicação, e destruído ao seu final.

```
1  entity exemplo
2    method print()
3      io.writeln("Executando print...");
4    end
5  end
6
7  entity start
8    method constructor()
9      var element ex = exemplo.new();
10     ex.print();
11   end
12 end
```

Figura 5.4: Criação de elementos

Tabela 5.7: Tipos de dados existentes na UbiL

Tipo	**Descrição**	**Tamanho máximo**
bool	Armazenam os valores booleanos *true* (verdadeiro) ou *false* (falso)	1 byte
int	Armazenam valores numéricos inteiros	8 bytes (2^{64} bits)
real	Armazenam valores numéricos com casas decimais	8 bytes (2^{64} bits)
string	Armazenam valores alfanuméricos	65536 bytes (2^{16} bits)
table	Armazenam dados onde cada informação possui um índice de acesso	(2^{64}) entradas
tuple	Armazenam tuplas obtidas dos contextos	65536 bytes (2^{16} bits)
element	Armazenam instâncias das entidades	65536 bytes (2^{16} bits)
thread	Armazenam a identificação dos fluxos de execução dos métodos concorrentes	8 bytes (2^{64} bits)
event	Armazenam informações sobre um evento	8 bytes (2^{64} bits)

A destruição dos elementos ocorre em dois momentos:

1. automaticamente, quando não são mais necessários;
2. explicitamente destruídos, com o uso do método *delete*.

Um elemento, ao ser destruído, invoca o seu destrutor, caso exista.

A figura 5.5 contém um exemplo de uso do método *delete*. Enquanto o método *new* cria novos elementos, o *delete* destrói elementos existentes. Na linha 12, o elemento *ex* é explicitamente destruído, mesmo antes do final da execução do método.

As variáveis tem como escopo a sub-rotina onde foram definidas, e existem somente durante a execução desta.

5.2.2 Métodos

Os métodos tem dois níveis de visibilidade: **privados** (acessíveis somente no escopo da entidade) e **públicos** (acessíveis por qualquer entidade). Quando não informada a visibilidade, assume-se que o método é público.

```
entity exemplo
  method print()
    io.writeln("Executando print...");
  end
end

entity start
  method constructor()
    var int loop;
    var element ex = exemplo.new();
    ex.print();
    ex.delete();
    for(loop=1; loop <= 10000; loop++)
      // ...
    end
  end
end
```

Figura 5.5: Destruição de elementos

A figura 5.6 contém um exemplo onde é definida a entidade *test* contendo o método *m1* (público) e o método *m2* (privado). *Start* cria uma instância da entidade *test* e utiliza seus métodos. As cláusulas *public* e *private* definem, respectivamente, um método público ou privado. Neste exemplo, as linhas 2 e 5 definem, respectivamente, o método *m1* (público) e *m2* (privado). A linha 13 invoca o método público *m1*. A linha 14 tenta invocar o método *m2*; porém, *m2* é um método privado da entidade *test* e, por este motivo, é gerado um erro durante a compilação.

```
entity test
  method public m1()
    io.writeln("m1");
  end
  method private m2()
    io.writeln("m2");
  end
end

entity start
  method constructor()
    var element t = test.new();
    t.m1(); // Ok
    t.m2(); // Erro
  end
end
```

Figura 5.6: Visibilidade dos métodos

A figura 5.7 contém um exemplo com métodos contendo um ou mais valores de retorno. Os tipos definidos entre colchetes, após a definição de um método,

indicam o tipo dos valores de retorno que o método disponibiliza ao final da sua execução. A cláusula *return* encerra a execução do método e define seus valores de retorno, caso existam. Neste exemplo, a linha 2 define o método *multiplica*, que recebe 2 argumentos (x e y) e retorna o resultado numérico determinado pela cláusula *return*, na linha 3. Por sua vez, a linha 5 define o método *calcula*, que recebe 2 argumentos (x e y) e retorna os dois resultados numéricos determinado pela cláusula *return*, na linha 6. Logo após o início da execução do elemento *Start*, a linha 10 invoca o método *multiplica* com os argumentos *10* e *20*, e armazena o resultado (*200*, neste exempÍo) na variável m. Por sua vez, a linha 11 invoca o método *calcula* com os argumentos *10* e *20*, e armazena os dois resultados (*30* e *-10*) nas variáveis a e s. As linhas 12, 13 e 14 exibem, respectivamente, a multiplicação, a adição e a subtração destes valores.

```
1  entity start
2    method multiplica(int x, int y) [int]
3      return(x * y);
4    end
5    method calcula(int x, int y) [int, int]
6      return(x + y, x - y);
7    end
8    method constructor()
9      var int m, a, s;
10     m = multiplica(10,20);
11     a, s = calcula(10,20);
12     io.writeln("Multiplicacao dos valores: ", m);
13     io.writeln("Adicao dos valores       : ", a);
14     io.writeln("Subtracao dos valores    : ", s);
15   end
16 end
```

Figura 5.7: Métodos com um ou mais valores de retorno

Os métodos podem ser sobrecarregados. A figura 5.8 contém um exemplo onde o método *print* é sobrecarregado em duas versões distintas: linha 2, com um argumento do tipo *string*, e linha 5, com um argumento do tipo *int*. Quando o elemento recebe uma mensagem, a assinatura desta mensagem define qual versão de *print* será invocada. Na linha 9, como o argumento é do tipo *string*, o método *print* que tem uma *string* como argumento (linha 2) será executado. Na linha 10, como o argumento é do tipo *int*, o método *print* que tem um *int* como argumento (linha 5) será executado.

Como visto no exemplo, cada método sobrecarregado possui uma assinatura distinta. A assinatura do método definido na linha 2 é “print:string” enquanto que

```
1  entity start
2    method print(string x)
3      io.writeln("String: ", x);
4    end
5    method print(int x)
6      io.writeln("Numero: ", x);
7    end
8    method constructor()
9      print("Oi");
10     print(10);
11   end
12 end
```

Figura 5.8: Sobrecarga de métodos

a assinatura do método definido na linha 5 é "print:int". Não podem existir dois métodos, com a mesma assinatura, em uma mesma entidade.

5.2.3 Propriedades

As propriedades tem como escopo a entidade onde foram definidas. Somente as sub-rotinas de um elemento conseguem acessar as suas propriedades.

A figura 5.9 contém um exemplo que define e utiliza uma propriedade. A cláusula *prop* define as propriedades de uma entidade. A propriedade *p1*, definida na entidade *test* (linha 2), é acessada no método *m1* na entidade (linha 4). Através dos métodos da entidade *test*, *Start* consegue acessar a propriedade *p1* (linha 11). Na linha 12, *Start* tenta acessar esta propriedade diretamente e, por este motivo, o compilador reporta um erro durante a compilação.

```
1  entity test
2    prop string p1;
3    method m1()
4      p1 = "Oi";   // Ok
5    end
6  end
7
8  entity start
9    method constructor()
10     var element t1 = test.new();
11     t1.m1();      // Ok
12     t1.p1 = "Oi"; // Erro
13   end
14 end
```

Figura 5.9: Visibilidade das propriedades

Para utilizar as propriedades fora do seu escopo, métodos de acesso devem ser definidos. As cláusulas *get* e *set* servem facilitar a definição destes métodos. A

cláusula *get* define o método que obtém o estado de uma propriedade, enquanto que a cláusula *set* define o método que altera o estado de uma propriedade.

Existem quatro possíveis situações durante a definição de uma propriedade:

- sem cláusula de *get* e sem cláusula de *set*: esta propriedade somente é acessível no seu escopo;
- apenas cláusula de *get*: esta propriedade é *read-only*, ou seja, ela somente pode ser lida fora do seu escopo;
- apenas cláusula de *set*: esta propriedade é *write-only*, ou seja, ela somente pode ser alterada fora do seu escopo;
- cláusula de *get* e *set*: esta propriedade é *read-write*, ou seja, pode ser lida e alterada fora do seu escopo.

A figura 5.10a contém exemplos com a sintaxe para definição dos métodos de acesso. A entidade *sample* define quatro propriedades: *p1*, na linha 2, somente acessível no seu escopo; *p2*, na linha 3, como *read-write*; *p3*, na linha 4, como *read-only* e *p4*, na linha 5, como *read-write*. Ao definir *p2*, como as cláusulas de *get* e *set* não foram implementadas, um método de *get* (obter o estado da propriedade) e um método de *set* (alterar o estado da propriedade) são implicitamente definidos. Ao definir *p4*, é indicado no método de *get* que o valor da propriedade é retornado enquanto que no método de *set* um processamento adicional será executado. *value* (linha 8) é um *token* especial da linguagem e contém o valor que está sendo atribuído a propriedade. Neste método de *set*, definido entre as linhas 7 e 13, caso *value* seja menor ou igual a *15*, *p4* recebe o valor de *value*; caso contrário recebe *0*.

A figura 5.10b contém exemplos que utilizam as propriedades definidas por *sample*. A linha 3 cria uma instância da entidade *sample*. Na linha 4, *s.p1* gera um erro durante a compilação porque *p1* não é acessível fora do seu escopo (entidade *sample*). Na linha 5, *s.p2* implicitamente executa o método de *get* da propriedade *p2*. Na linha 6, *s.p3* implicitamente executa o método de *get* da propriedade *p3*. Porém, na linha 7, o compilador tenta executar o método de *set* da propriedade *p3*; como esta propriedade é *read-only*, o compilador reporta um erro durante a

```
1  entity sample
2    prop int p1;
3    prop int p2 get set;
4    prop int p3 get;
5    prop int p4
6      get return p4;
7      set
8        if (value <= 15)
9          p4 = value;
10       else
11         p4 = 0;
12       end
13     end
14   end
15 end
```

(a)

```
entity start                                    1
  method constructor()                          2
    var element s = sample.new();               3
    io.writeln("x = " + s.p1); // Erro          4
    io.writeln("x = " + s.p2);                  5
    io.writeln("x = " + s.p3);                  6
    s.p3 = 20;                 // Erro          7
    s.p4 = 30;                                  8
  end                                           9
end                                             10
```

(b)

Figura 5.10: Métodos de acesso as propriedades

compilação. Na linha 8 o método de *set* é executado normalmente, e *p4* recebe o valor *30*.

Como o escopo das propriedades é a entidade em que foram definidas, as propriedades são destruídas junto com o elemento.

5.2.4 Herança entre entidades

A figura 5.11 contém um exemplo de herança entre entidades. Na UbiL somente existe herança simples. A cláusula *inherits* indica de qual entidade será herdado seus métodos e serviços. Neste exemplo, a entidade a (definida na linha 1) possui apenas o método x (linha 2). A entidade b (definida na linha 7) herda o método x da entidade a e implementa o método y (linha 8). *Start* cria um elemento a partir da entidade a (linha 15) e invoca o seu método x (linha 17). *Start* também cria um elemento a partir da entidade b (linha 16) e invoca o seu método x (herdado da entidade a) na linha 18 e o seu método y na linha 19. As linhas 17 e 18 exibem a mensagem "Executando a::x", enquanto que a linha 19 exibe a mensagem "Executando b::y".

5.2.5 Métodos polimórficos

A figura 5.12a contém um exemplo de polimorfismo. A cláusula *virtual* indica que um método pode ser sobrescrito; por sua vez, a cláusula *override* indica que o método está sobrescrevendo um método virtual. Neste exemplo, além de *Start*, são definidas outras três entidades: a super-entidade a e as sub-entidades que estendem

```
1  entity a
2    method x()
3      io.writeln("Executando a::x");
4    end
5  end
6
7  entity b inherits a
8    method y()
9      io.writeln("Executando b::y");
10   end
11 end
12
13 entity start
14   method constructor()
15     var element ea = a.new();
16     var element eb = b.new();
17     ea.x();
18     eb.x();
19     eb.y();
20   end
21 end
```

Figura 5.11: Herança entre entidades

a super-entidade (*b1* e *b2*). A entidade *a* define o método *x* (linha 2) e o método virtual *y* (linha 5). A entidade *b1* sobrescreve o método *y* da super-entidade (linha 10) e define o método *z* (linha 13). Da mesma forma a entidade *b2* sobrescreve o método *y* (linha 18) e define o método *z* (linha 21). Ao iniciar a execução, *Start* cria um elemento a partir da super-entidade *a* (linha 28) e executa os métodos *x()* e *y()*. Na linha 29, *Start* cria um elemento a partir da sub-entidade *b1* e executa o método *x()* que foi herdado da super-entidade *a*, o método *y()* que foi sobrescrito pela sub-entidade *b1* e o método *z()* que foi definido na sub-entidade *b1*. Da mesma forma, na linha 30 *Start* cria um elemento a partir da sub-entidade *b2* e executa o método *x()* que também foi herdado da super-entidade *a*, o método *y()* que foi sobrescrito pela sub-entidade *b2* e o método *z()* que foi definido na sub-entidade *b2*.

A figura 5.12b contém um exemplo de como um método sobrescrito pode receber mensagens do método que o sobrescreveu. A cláusula ***super*** (linha 11) invoca o método sobrescrito (definido na linha 5).

Métodos que podem ser sobrescritos devem ter a cláusula *virtual* enquanto que métodos que sobrescrevem métodos virtuais devem utilizar a cláusula *override*. Caso contrário, deve ser gerado um aviso durante a compilação.

```
1  entity a
2    method x()
3      io.writeln("Executando a::x");
4    end
5    method virtual y()
6      io.writeln("Executando a::y");
7    end
8  end
9  entity b1 inherits A
10   method override y()
11     io.writeln("Executando b1::y");
12   end
13   method z()
14     io.writeln("Executando b1::z");
15   end
16 end
17 entity b2 inherits A
18   method override y()
19     io.writeln("Executando b2::y");
20   end
21   method z()
22     io.writeln("Executando b2::z");
23   end
24 end
25 entity start
26   method constructor()
27     var element e;
28     e = a.new();  e.x(); e.y();
29     e = b1.new(); e.x(); e.y(); e.z();
30     e = b2.new(); e.x(); e.y(); e.z();
31   end
32 end
```

(a)

```
1  entity a
2    method x()
3      io.writeln("Executando a::x");
4    end
5    method virtual y()
6      io.writeln("Executando a::y");
7    end
8  end
9  entity b inherits a
10   method override y()
11     super();
12     io.writeln("Executando b::y");
13   end
14 end
15 entity start
16   method constructor()
17     var element ea, eb;
18     ea = a.new(); ea.x(); ea.y();
19     eb = b.new(); eb.x(); eb.y();
20   end
21 end
```

(b)

Figura 5.12: Polimorfismo em métodos

5.2.6 Clonagem de elementos

A clonagem de elementos ocorre durante a execução e cria um clone de determinado elemento.

A figura 5.13 contém um exemplo. A linha 7 cria o elemento *t1* a partir da entidade *test*, definido o valor *10* para sua propriedade x na linha 8. A linha 9 cria um clone do elemento *t1* e o armazena no elemento *t2*. A linha 10 exibe os valores de *t1.x* e *t2.x* (ambos contém o valor *10*). A linha 11 e 12 definem novos valores para a propriedade x de *t1* e *t2*. A linha 13 exibe os novos valores de *t1.x* e *t2.x* (*1* e *2*, respectivamente).

```
1  entity test
2    prop int x get set;
3  end
4
5  entity start
6    method constructor()
7      var element t1 = test.new();
8      t1.x = 10;
9      var element t2 = t1.clone();
10     io.writeln("t1.x=", t1.x, " t2.x=", t2.x);
11     t1.x = 1;
12     t2.x = 2;
13     io.writeln("t1.x=", t1.x, " t2.x=", t2.x);
14   end
15 end
```

Figura 5.13: Clonagem de elementos

5.3 Contextos privados

Ao iniciar a aplicação, os contextos privados são implicitamente criados pelo ambiente de execução. Existem quatro contextos privados:

- ***Location***: contém as informações de localização simbólica e física. *location.symbolic* contém a descrição simbólica da localização enquanto que *location.physical* contém a descrição física. Exemplos de localização simbólica são: "próximo ao mercado", "no trem" e "em casa". Exemplos de localização física, armazenadas em *location.physical.gps* são "latitude=12345", "longitude=2323" e "altitude=10";

- ***Identity***: contém informações de identificação como, por exemplo, "name=Alex", "cpf=123.456.789-0", "mobile=(51) 9999-9999", entre outros;

- ***Time***: contém informações relativas a data e hora como, por exemplo, "date=18/01/2010", "day=18", "month=1", "year=2010", "day_of_week=segunda", "now=17:17:35", entre outros;

- ***Activity***: contém informações sobre o que está ocorrendo no momento como, por exemplo, "what=Trabalhando no texto da dissertação".

As informações existentes nos contextos privados somente são acessíveis aos elementos do aplicativo.

A figura 5.14 contém um exemplo. Neste exemplo, as linhas entre 3 e 7 exibem algumas das informações contextuais existentes nos contextos privados. A tabela 5.8 contém a lista completa.

```
1  entity start
2    method constructor()
3      io.writeln("Host atual : " + identity.host);
4      io.writeln("Dispositivo: " + identity.device);
5      io.writeln("Tempo      : " + time);
6      io.writeln("Atividade  : " + activity.what);
7      io.writeln("Onde estou : " + location.symbolic);
8    end
9  end
```

Figura 5.14: Informações contextuais existentes nos contextos privados

Salienta-se que as informações existentes neste contextos podem ser explicitamente informadas pela aplicação bem como obtidas pelo ambiente de execução através do uso de sensores (seção 6.2.4). Por exemplo, o usuário pode informar o seu nome e a aplicação armazená-lo em *identity.name*. Da mesma forma, a localização pode ser obtida através de um sensor e armazenada no contexto *location*.

Tabela 5.8: Informações disponíveis nos contextos privados

Contexto	**Informação**	**Descrição**
Location	*physical.gps.latitude*	Latitude
	physical.gps.longitude	Longitude
	physical.gps.altitude	Altitude
	symbolic	Descrição simbólica
Identity	*name*	Identificação de quem está executando o aplicativo
	host	Identificação textual do dispositivo atual
	device	Identificação do dispositivo atual
	ip	Endereço IP
	mac_address	Endereço MAC
Time	*date*	Dia, mês e ano atual
	day	Dia atual
	month	Mês atual
	year	Ano atual
	now	Hora, minuto e segundo atual
	hour	Hora atual
	minute	Minuto atual
	second	Segundo atual
	day_of_week	Dia da semana
Activity	*what*	O que está sendo feito

O desenvolvedor em UbiL também pode criar novas informações contextuais nestes contextos. Para tal, basta que o desenvolvedor configure um valor para uma informação qualquer. O exemplo da figura 5.15 define três novas informações no

contexto *identity*: *cpf* (linha 3), *rg* (linha 4) e *celular* (linha 5). Caso seja definido um valor para uma informação inexistente (linhas 3, 4 e 5 deste exemplo), esta informação automaticamente é criada e inicializada com o valor definido; caso seja solicitado o valor de uma informação que não existe, o valor nulo (*nil*) é retornado como resultado da operação.

```
1  entity start
2    method constructor()
3      identity.cpf     = "123.456.789-01";
4      identity.rg      = "1234567890";
5      identity.celular = "(51) 9999-9999";
6    end
7  end
```

Figura 5.15: Criando informações personalizadas nos contextos

5.4 Contextos públicos

Os contextos públicos são formados dinamicamente durante a execução da aplicação. Para auxiliar no compartilhamento de informações entre os elementos, cada contexto mantém as seguintes informações contextuais: membros, conteúdos e serviços.

5.4.1 Membros

Membros são os elementos, de uma ou mais aplicações, que fazem parte do contexto. Durante a execução da aplicação os elementos podem associar-se e desassociar-se dos contextos.

A lista contendo os membros do contexto é atualizada pelo ambiente de execução através dos seguintes métodos:

- *mjoin*: associa o elemento atual a um contexto;
- *mleave*: desassocia o elemento atual a um contexto.

A figura 5.16 contém um exemplo onde um elemento associa-se e desassocia-se de um contexto. O uso do operador chaves ("{}") indica a utilização de comandos relativos a contextos públicos. Neste exemplo, na linha 3 o elemento

atual é associado ao contexto "MeuContexto", com a identificação "Alex". Neste momento, se outros elementos tentarem identificar os membros deste contexto, "Alex" será listado como um deles. Na linha 5 o elemento atual é desassociado do contexto.

```
1  entity start
2    method constructor()
3      {"MeuContexto"}.mjoin("Alex");
4      io.writeln("Estou associado ao contexto MeuContexto");
5      {"MeuContexto"}.mleave();
6      io.writeln("Nao estou mais associado ao contexto MeuContexto");
7    end
8  end
```

Figura 5.16: Associação e desassociação em um contexto

Neste último exemplo, o nome do contexto foi definido em tempo de compilação. Porém, os nomes dos contextos podem ser obtidos durante a execução, ou descobertos dinamicamente através do uso dos contextos hierárquicos (explicados posteriormente nesta seção).

A figura 5.17 contém um exemplo onde o nome do contexto é informado pelo usuário. A função *readln*, da biblioteca *io*, lê dados do teclado. Neste exemplo, na linha 4, o usuário informa o nome do contexto em qual deseja associar-se. Na linha 5 ocorre a associação ao contexto informado, enquanto que na linha 7 ocorre a desassociação.

```
1  entity start
2    method constructor()
3      var string contexto;
4      contexto = io.readln();
5      {contexto}.mjoin("Alex");
6      io.writeln("Estou associado ao contexto ", contexto);
7      {contexto}.mleave();
8      io.writeln("Nao estou mais associado ao contexto ", contexto);
9    end
10 end
```

Figura 5.17: Associação e desassociação em um contexto informado pelo usuário

5.4.2 Conteúdos compartilhados

A publicação, busca e remoção de conteúdos ocorre dinamicamente durante a execução das aplicações.

A lista contendo os conteúdos compartilhados do contexto é atualizada pelo ambiente de execução através dos seguintes métodos:

- *cpublish*: sempre não bloqueante, publica determinado conteúdo no contexto;
- *clist*: sempre não bloqueante, lista os conteúdos do contexto;
- *cfind*: bloqueante, busca determinado conteúdo no contexto. Caso não exista, bloqueia a aplicação até que o conteúdo esteja disponível;
- *cfindnb*: não bloqueante, busca determinado conteúdo no contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução;
- *cget*: bloqueante, busca e remove determinado conteúdo do contexto. Caso não exista, bloqueia a aplicação até que o conteúdo esteja disponível;
- *cgetnb*: não bloqueante, busca e remove determinado conteúdo do contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução;
- *cremove*: bloqueante, remove determinado conteúdo do contexto. Caso não exista, bloqueia a aplicação até que o conteúdo esteja disponível;
- *cremovenb*: não bloqueante, remove determinado conteúdo do contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução.

Um elemento, ao compartilhar um conteúdo, informa o ambiente de execução que este conteúdo será compartilhado no contexto, podendo ser lido, alterado e removido pelos outros elementos. A visibilidade dos conteúdos sempre é *public*, ou seja, são acessíveis por todos os elementos.

A figura 5.18 contém um exemplo sobre como publicar e remover conteúdos de um contexto. Neste exemplo, na linha 3 o conteúdo "54321", com chave de identificação *"Número 1"*, é disponibilizado no contexto "MeuContexto" através do método *cpublish*. Após isso, este conteúdo é retirado através do método *cget* na linha 4. A sentença da linha 4 também invoca a função *writeln* que exibe uma mensagem contendo o conteúdo *54321*, obtido do contexto.

```
1  entity start
2    method constructor()
3      {"MeuContexto"}.cpublish("Numero 1" => 54321);
4      io.writeln("Informacao retirada do contexto: " + {"MeuContexto"}.cget("Numero 1"));
5    end
6  end
```

Figura 5.18: Publicação e remoção de conteúdos em um contexto

5.4.3 Serviços compartilhados

A publicação, execução e remoção de serviços ocorre dinamicamente durante a execução das aplicações.

A lista contendo os serviços compartilhados do contexto é atualizada pelo ambiente de execução através dos seguintes métodos:

- *spublish*: sempre não bloqueante, publica determinado serviço no contexto;
- *slist*: sempre não bloqueante, lista os serviços do contexto;
- *sfind*: bloqueante, busca determinado serviço no contexto. Caso não exista, bloqueia a aplicação até que o serviço esteja disponível;
- *sfindnb*: não bloqueante, busca determinado serviço no contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução;
- *srun*: bloqueante, busca e executa determinado serviço no contexto. Caso não exista, bloqueia a aplicação até que o serviço esteja disponível;
- *srunnb*: não bloqueante, busca e executa determinado serviço no contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução;
- *sremove*: bloqueante, remove determinado serviço do contexto. Caso não exista, bloqueia a aplicação até que o serviço esteja disponível;
- *sremovenb*: não bloqueante, remove determinado serviço do contexto. Caso não exista, informa a aplicação mas não bloqueia sua execução.

Um elemento, ao compartilhar um serviço, informa o ambiente de execução que este serviço será compartilhado no contexto, podendo ser executado por outros

elementos. A visibilidade dos serviços sempre é *public*, ou seja, são acessíveis por todos os elementos.

A figura 5.19a contém um exemplo de publicação, execução e remoção de serviços. No construtor desta entidade, nas linhas 3 e 4, o método *spublish* compartilha, respectivamente, os serviços *service1* e *service2*, no contexto *MeuContexto*. No destrutor, as linhas 7 e 8 utilizam o método *sremove* para remover os serviços publicados.

```
1  entity start
2    method constructor()
3      {"MeuContexto"}.spublish("service1");
4      {"MeuContexto"}.spublish("service2");
5    end
6    method destructor()
7      {"MeuContexto"}.sremove("service"1);
8      {"MeuContexto"}.sremove("service2");
9    end
10   service service1() [int]
11     return 10;
12   end
13   service service2() [string]
14     return "mensagem";
15   end
16 end
```

(a)

```
1  entity start
2    method constructor()
3      io.writeln("Resultado do servico 1: " +
4          {"MeuContexto"}.srun("service1"));
5      io.writeln("Resultado do servico 2: " +
6          {"MeuContexto"}.srun("service2"));
7    end
8  end
```

(b)

Figura 5.19: Publicação, execução e remoção de serviços

Quando um elemento necessita de um serviço, ele busca nos contextos quem pode prestar este serviço. A figura 5.19b contém um exemplo. Nas linhas 3 e 5 pode ser vista a sintaxe para execução de serviços. Para tal, basta utilizar o método *srun* do contexto, informando o serviço a ser executado e seus argumentos (caso existam). Neste exemplo, ao solicitar a execução dos serviços *service1* e *service2* (linhas 3 e 5), o ambiente de execução automaticamente realiza a busca pelos serviços no contexto *MeuContexto*, e solicita sua execução diretamente ao elemento que os publicou (figura 5.19a). A linha 3 exibe a mensagem "Resultado do servico 1: 10", enquanto que a linha 5 exibe a mensagem "Resultado do servico 2: mensagem".

A figura 5.20 contém um exemplo que lista os serviços disponíveis no contexto *MeuContexto*, bem como suas características. O método *slist* (linha 3) obtém a lista dos serviços de determinado contexto. As propriedades *name* (linha 4), *description* (linha 5) e *tbf* (linha 6) são algumas das características existentes.

A tabela 5.9 contém a lista completa das características disponíveis sobre um

```
1  entity start
2    method constructor()
3      foreach(tuple service in {"MeuContexto"}.slist())
4        io.writeln("Nome     : ", service.name);
5        io.writeln("Descricao: ", service.description);
6        io.writeln("TBF      : ", service.tbf);
7      end
8    end
9  end
```

Figura 5.20: Listando características dos serviços

serviço. Estas características, presentes em todos os serviços publicados, podem ser utilizadas para selecionar um serviço, dentre os disponíveis.

Tabela 5.9: Características existentes em um serviço

Característica	**Descrição**
Name	Nome do serviço
Description	Descrição do serviço
Return_values	Valores de retorno do serviço
Arguments	Argumentos necessários a execução do serviço
TBF	Time between failures (tempo médio entre falhas)
Response_time	Tempo médio para processar um requisição

Cabe aqui ressaltar que o nome do contexto e do serviço não necessitam serem conhecidos durante a compilação do programa. Estas informações podem ser obtidas durante a execução ou descobertos dinamicamente através do uso dos contextos hierárquicos (explicados posteriormente nesta seção). A figura 5.21 contém um exemplo. As linhas 5 e 6 realizam o mesmo processamento. Nestas linhas, o serviço *service1*, do contexto *MeuContexto*, é executado.

```
1  entity start
2    method constructor()
3      var string context = "MeuContexto";
4      var string service = "service1";
5      io.writeln("Resultado do servico 1: " + {context}.srun(service));
6      io.writeln("Resultado do servico 1: " + {"MeuContexto"}.srun("service1"));
7    end
8  end
```

Figura 5.21: Descoberta dinâmica de serviços

Assim como os métodos, os serviços também podem ser sobrecarregados. Quando um elemento recebe uma solicitação para executar um serviço, a assinatura define qual a versão sobrecarregada do serviço será executada. Na figura 5.22(a) o serviço *print* é sobrecarregado em duas versões distintas: linha 2, com um

argumento do tipo *string*, e linha 5, com um argumento do tipo *int*. Na linha 9, as duas versões do serviço *print* são compartilhadas no contexto *MeuContexto*. Na figura 5.22(a), na linha 10, o serviço *print:string* é invocado enquanto que, na linha 11, *print::int* é executado.

```
1  entity start
2    service print(string x)
3      io.writeln("Caracter: " + x);
4    end
5    service print(int x)
6      io.writeln("Numero: " + x);
7    end
8    method constructor()
9      {"MeuContexto"}.spublish("print");
10   end
11 end
```

(a)

```
entity start
  method constructor()
    {"MeuContexto"}.srun("print","Oi");
    {"MeuContexto"}.srun("print",10);
  end
end
```

(b)

Figura 5.22: Sobrecarga de serviços

Assim como os métodos, cada serviço sobrecarregado possui uma assinatura distinta. Com isso, não podem existir dois serviços com a mesma assinatura, em uma mesma entidade.

5.5 Contextos hierárquicos

Contextos hierárquicos possibilitam que, a partir de um contexto inicial, contextos públicos e informações contextuais mais especializadas possam ser descobertas, propagando esta busca até o último nível da hierarquia.

A figura 5.23 contém um exemplo onde, a partir de um caminho, é possível identificar os contextos existentes. O método *csearch* (linha 4), existente nos contextos, lista os contextos encontrados. Para cada contexto encontrado, é possível obter informações como nome, caminho, entre outras características. Neste exemplo, a linha 4 busca, a partir do caminho "/Unisinos/Disciplinas", os contextos que tenham "programacao" no seu nome. A expressão regular passada como argumento para *csearch* filtra os contextos que farão parte do resultado. Após isso, o nome e o caminho dos contextos encontrados são listados na linha 7.

A figura 5.24 contém um exemplo onde, a partir de um caminho, é possível identificar os serviços existentes. O método *ssearch*, existente nos contextos, lista os serviços encontrados. Para cada serviço encontrado, é possível obter

```
1  entity start
2    method constructor()
3      var table context_info;
4      var table context_list = {"/Unisinos/Disciplinas"}.csearch("*programacao*");
5      io.writeln("Contextos encontrados:");
6      foreach(context_info in context_list)
7        io.writeln("Nome: ", context_info["name"], " Caminho: ", context_info["path"]);
8      end
9    end
10 end
```

Figura 5.23: Busca por contextos

informações como nome, descrição, argumentos, resultados esperados, entre outras características. Neste exemplo, a linha 4 busca, a partir do caminho *"/"*, os serviços com o nome *"conversao_ moedas"*. A expressão regular passada como argumento para *ssearch* filtra os serviços que farão parte do resultado. Após isso, o nome e a descrição dos serviços encontrados são listados na linha 7.

```
1  entity start
2    method constructor()
3      var table service_info;
4      var table service_list = {"/"}.ssearch("conversao_moedas");
5      io.writeln("Servicos encontrados:");
6      foreach(service_info in service_list)
7        io.writeln("Nome: ", service_info["name"], " Descricao: ", service_info["description"]);
8      end
9    end
10 end
```

Figura 5.24: Busca por serviços

Da mesma forma como ocorre com as informações contextuais, as informações sobre os contextos hierárquicos ficam distribuídas entre as várias aplicações em execução, sendo acessíveis por qualquer elemento.

5.6 Adaptação das aplicações

As regras para adaptação das aplicações podem ser inseridas diretamente nos programas através de cláusulas específicas da UbiL. Existem três formas de adaptação: através dos eventos, durante a resolução de nomes de entidades e durante a resolução de nomes de sub-rotinas (métodos ou serviços).

5.6.1 Adaptação através dos eventos

Os eventos auxiliam a adaptação das aplicações às mudanças ocorridas nos contextos. Existem sete tipos de eventos referentes aos contextos privados e públicos. São eles:

- *on_content_changed*: ocorre quando alterações são realizadas em determinado conteúdo do contexto;
- *on_publish_content*: ocorre quando conteúdos são publicados em um contexto;
- *on_remove_content*: ocorre quando conteúdos são removidos de um contexto;
- *on_publish_service*: ocorre quando serviços são publicados em um contexto;
- *on_remove_service*: ocorre quando serviços são removidos de um contexto;
- *on_member_join*: ocorre quando elementos são associados em determinado contexto;
- por fim, *on_member_leave*: ocorre quando elementos são desassociados de determinado contexto.

A figura 5.25a contém um exemplo que informa a localização simbólica atual do dispositivo onde o aplicativo está sendo executado. O evento *on_content_changed* ocorre quando determinado conteúdo de um contexto é alterado. A linha 3 define que o método *alt_loc* será executado quando o conteúdo *location.symbolic* for alterado. *alt_loc* exibe a antiga e a nova localização simbólica (linha 8). A figura 5.25b contém o mesmo exemplo, porém sem o uso de eventos. Salienta-se que, nesta versão sem o uso dos eventos, além do código menos legível e, conseqüentemente, mais complexo, o consumo de CPU também é mais elevado porque o programa fica em um laço infinito verificando se ocorreu uma alteração na localização.

Além dos eventos pré-definidos, entidades definidas pelo usuário podem criar seus próprios eventos. A figura 5.26 contém um exemplo. A entidade *reservatorio*

```
1  entity start
2    method constructor()
3      location.symbolic.on_content_changed
4        += alt_loc;
5    end
6
7    method alt_loc(string old, string new)
8      io.writeln("Localizacao " + old +
9          " mudou para " + new);
10   end
11 end
```

(a)

```
1  entity start
2    method constructor()
3      var string old_loc = location.symbolic;
4      while (true)
5        if (old_loc <> location.symbolic)
6          io.writeln("Localizacao " + old_loc +
7              " mudou para " +
8              location.symbolic);
9          old_loc = location.symbolic;
10       end
11     end
12   end
13 end
```

(b)

Figura 5.25: Atualização da localização do usuário

define o evento *cheio* (linha 2), evento este que é executado sempre que o número de litros for maior ou igual a 1000. No *set* da propriedade *litros* (definido na linha 5) é verificado se o evento deve ocorrer. Se *litros* contiver um valor maior ou igual a *1000* (linha 7), o evento *cheio* é disparado (linha 9). Durante a execução de *Start*, a linha 17 define que o método *reservatorio_ cheio* deve ser executado quando o evento *cheio* ocorrer. Na linha 19 *litros* recebe o valor *100* e, neste caso, nada ocorre. Na linha 20 *litros* recebe o valor *1000*, o que ocasiona que o método *reservatorio_ cheio*, que está associado ao evento *cheio*, seja executado.

```
1  entity reservatorio
2    prop event cheio;
3    prop int litros
4      get return litros;
5      set
6        litros = value;
7        if (litros >= 1000)
8          litros = 1000;
9          raise cheio;
10       end
11     end
12   end
13 end
14
15 entity start
16   method constructor()
17     var element agua = reservatorio.new();
18     agua.cheio += reservatorio_cheio;
19     agua.litros = 100;
20     agua.litros = 1000;
21   end
22   method reservatorio_cheio()
23     io.writeln("O reservatorio esta cheio");
24   end
25 end
```

Figura 5.26: Definição de eventos em uma entidade

5.6.2 Adaptação durante a resolução de nomes de entidades

A figura 5.27a contém um exemplo onde a entidade *test* que será instanciada em *Start*, na linha 17, depende do contexto *activity.what*. A cláusula *when* define em quais contextos uma entidade se adapta. Neste exemplo, caso *activity.what* seja "teaching", a entidade definida na linha 1 será utilizada; caso seja "learning", a entidade definida na linha 8 será utilizada; caso contrário, não existe uma entidade que se adapta ao contexto atual, então uma mensagem de erro é exibida pelo ambiente de execução, e a aplicação é finalizada. A figura 5.27b contém o mesmo exemplo, porém sem utilizar o suporte a adaptação da linguagem.

```
1  entity test
2    when (activity.what == "teaching")
3    method run()
4      io.writeln( "Estou ensinando..." );
5    end
6  end
7
8  entity test
9    when (activity.what == "learning")
10   method run()
11     io.writeln( "Estou aprendendo..." );
12   end
13 end
14
15 entity start
16   method constructor()
17     var element etest = test.new();
18     etest.run();
19   end
20 end
```

(a)

```
1  entity test_teaching
2    method run()
3      io.writeln( "Estou ensinando..." );
4    end
5  end
6  entity test_learning
7    method run()
8      io.writeln( "Estou aprendendo..." );
9    end
10 end
11 entity start
12   method constructor()
13     var element etest;
14     if (activity.what == "teaching")
15       etest = test_teaching.new();
16     else if (activity.what == "learning")
17       etest = test_learning.new();
18     else
19       abort("Contexto nao suportado !!!");
20     end
21     etest.run();
22   end
23 end
```

(b)

Figura 5.27: Adaptação na resolução de nomes de entidades

A figura 5.28a contém outro exemplo onde a entidade *test* que será instanciada em *Start*, na linha 23, também depende do contexto "activity.what"; porém, uma das definições da entidade *test* se adapta a todos os contextos. Quando não existe a cláusula *when*, significa que a entidade se adapta a todos os contextos. Neste exemplo, caso *activity.what* seja "teaching", a entidade definida na linha 1 será utilizada; caso seja "learning", a entidade definida na linha 8 será utilizada; caso contrário, a entidade que se adapta a todos os contextos, definida na linha 15, será utilizada. A figura 5.28b contém o mesmo exemplo, porém sem utilizar o suporte a adaptação da linguagem.

```
1  entity test
2    when (activity.what == "teaching")
3    method run()
4      io.writeln( "Estou ensinando..." );
5    end
6  end
7
8  entity test
9    when (activity.what == "learning")
10   method run()
11     io.writeln( "Estou aprendendo..." );
12   end
13 end
14
15 entity test
16   method run()
17     io.writeln( "Nao sei..." );
18   end
19 end
20
21 entity start
22   method constructor()
23     var element etest = test.new();
24     etest.run();
25   end
26 end
```

(a)

```
1  entity test_teaching
2    method run()
3      io.writeln( "Estou ensinando..." );
4    end
5  end
6  entity test_learning
7    method run()
8      io.writeln( "Estou aprendendo..." );
9    end
10 end
11 entity test_undefined
12   method run()
13     io.writeln( "Nao sei..." );
14   end
15 end
16 entity start
17   method constructor()
18     var element etest;
19     if (activity.what == "teaching")
20       etest = test_teaching.new();
21     else if (activity.what == "learning")
22       etest = test_learning.new();
23     else
24       etest = test_undefined.new();
25     end
26     etest.run();
27   end
28 end
```

(b)

Figura 5.28: Outro exemplo de adaptação na resolução de nomes de entidades

5.6.3 Adaptação durante a resolução de nomes de sub-rotinas

A figura 5.29a contém um exemplo onde o método *run* que será executado em *Start*, na linha 4, depende do contexto *activity.what*. Neste exemplo, caso *activity.what* seja "teaching", o método definido na linha 9 será utilizado; caso seja "learning", o método definido na linha 13 será utilizado; caso contrário, não existe um método que se adapta ao contexto atual, então uma mensagem de erro é exibida pelo ambiente de execução, e a aplicação é finalizada. A figura 5.29b contém o mesmo exemplo, porém sem utilizar o suporte a adaptação da linguagem.

A figura 5.30a contém um exemplo onde o serviço *run*, compartilhado por *Start* na linha 3, se adapta ao contexto *activity.what*. Neste exemplo, caso *activity.what* seja "teaching", o serviço definido na linha 5 será executado; caso seja "learning", o serviço definido na linha 9 será executado; caso contrário, não existe um serviço que se adapta ao contexto atual, então a solicitação de serviço é ignorada pela aplicação. A figura 5.29b contém o mesmo exemplo, porém sem utilizar o suporte a adaptação da linguagem.

```
entity start
  method constructor()
    var element etest = test.new();
    etest.run();
  end
end

entity test
  method run()
    when (activity.what == "teaching")
    io.writeln("Estou ensinando...");
  end
  method run()
    when (activity.what == "learning")
    io.writeln("Estou aprendendo...");
  end
end
```

(a)

```
entity start
  method constructor()
    var element etest = test.new();
    if (activity.what == "teaching")
      etest.run_teaching();
    else if (activity.what == "learning")
      etest.run_learning();
    else
      abort("Contexto nao suportado !!!");
    end
  end
end
entity test
  method run_teaching()
    io.writeln("Estou ensinando...");
  end
  method run_learning()
    io.writeln("Estou aprendendo...");
  end
end
```

(b)

Figura 5.29: Adaptação durante a resolução de nomes de métodos

```
entity start
  method constructor()
    {"MeuContexto"}.spublish("run");
  end
  service run() [string]
    when (activity.what == "teaching")
    return "Estou ensinando...";
  end
  service run() [string]
    when (activity.what == "learning")
    return "Estou aprendendo...";
  end
end
```

(a)

```
entity start
  method constructor()
    {"MeuContexto"}.spublish("run");
  end
  method run_teaching() [string]
    return "Estou ensinando...";
  end
  method run_learning() [string]
    return "Estou aprendendo...";
  end
  service run() [string]
    if (activity.what == "teaching")
      return run_teaching();
    else if (activity.what == "learning")
      return run_learning();
    end
  end
end
```

(b)

Figura 5.30: Adaptação durante a resolução de nomes de serviços

Da mesma forma como ocorre com os métodos, quando não existe a cláusula *when*, significa que o serviço se adapta a todos os contextos.

5.7 Mobilidade forte de código

A figura 5.31 contém um exemplo de mobilidade forte de código através da migração. O método *move* (linha 4) faz com que o aplicativo em execução mova-se entre dispositivos distintos. No exemplo, "desktop.casa" é a identificação simbólica que identifica o dispositivo para o qual o aplicativo será movido. As linhas 3 e 5

exibem, respectivamente, o dispositivo em que o aplicativo está sendo executado, antes e depois da mobilidade ocorrer.

```
1  entity start
2    method constructor()
3      io.writeln("Executando em " + identity.host + ". Vou ir para outro host...");
4      move("desktop.casa");
5      io.writeln("Executando em " + identity.host);
6    end
7  end
```

Figura 5.31: Mobilidade de código

Os aplicativos podem ser movidos entre dispositivos distintos, bem como entre arquiteturas distintas. O estado das propriedades, dos conteúdos e dos serviços compartilhados nos contextos por seus elementos, são movidos juntamente com o aplicativo. Após a movimentação, os elementos permanecem nos mesmos contextos em que se encontravam antes da mobilidade, compartilhando as mesmas informações contextuais nestes contextos. Ao buscar os conteúdos e serviços compartilhados pelos elementos do aplicativo que foi movido, eles serão encontrados na nova localização.

5.8 Concorrência

Para facilitar a implementação de aplicativos concorrentes, os métodos podem ser executados em um fluxo de execução independente.

A figura 5.32 contém um exemplo. A cláusula *concurrent* identifica os métodos que serão executados concorrentemente. Da mesma forma, o método *cmwait* interrompe a execução do programa até que o método concorrente termine sua execução. Neste exemplo, *Start* executa os métodos *method1* e *method2* (linhas 3 e 4). Em função da cláusula *concurrent*, estes métodos executam concorrentemente ao fluxo de execução do elemento *Start*. Após isso, *Start* executa seu processamento (linhas 5, 6 e 7) e aguarda o término das execuções de *method1* e *method2* através do método *cmwait* (linhas 8 e 9).

A figura 5.33 contém os fluxos de execução existentes durante a execução deste exemplo. Salienta-se que durante a invocação dos métodos *method1* e *method2*, dois novos fluxos de execução são criados, fluxos estes que rodam

```
1  entity start
2    method constructor()
3      var thread m1 = method1() [concurrent];
4      var thread m2 = method2() [concurrent];
5      for(int x = 1; x <= 5; x++)
6        io.writeln( "Elemento start esta na iteracao ", x );
7      end
8      m1.wait();
9      io.writeln("Method1 terminou...");
10     m2.wait();
11     io.writeln("Method2 terminou...");
12   end
13   method method1()
14     for(int i = 1; x <= 10; x++)
15       io.writeln("Running method1...");
16     end
17   end
18   method method2()
19     for(int i = 1; x <= 10; x++)
20       io.writeln("Running method2...");
21     end
22   end
23 end
```

Figura 5.32: Métodos concorrentes

concorrentemente ao fluxo de execução da aplicação.

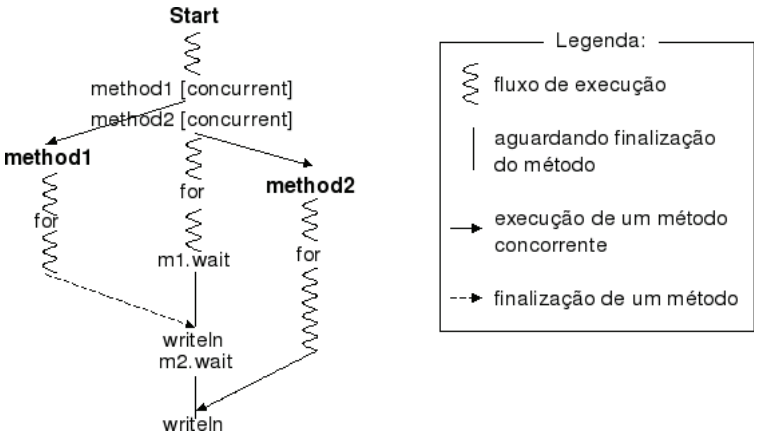


Figura 5.33: Fluxos de execução existentes com os métodos concorrentes

A figura 5.34 contém um exemplo que calcula o fatorial de dois números concorrentemente. *Start* executa duas vezes o método *fatorial* (linhas 3 e 4). Em função da cláusula *concurrent*, estas duas invocações ao método *fatorial* executam concorrentemente ao fluxo de execução do elemento *Start*. Após isso, *Start* aguarda o término da execução destes fluxos concorrentes e exibe o resultado de cada

execução (linhas 5 e 6).

```
1  entity start
2    method constructor()
3      var thread m1 = fatorial(6) [concurrent];
4      var thread m2 = fatorial(7) [concurrent];
5      io.writeln("Fatorial de 6: ", m1.wait());
6      io.writeln("Fatorial de 7: ", m2.wait());
7    end
8    method fatorial(int n) [int]
9      if (n == 0)
10       return 1;
11     else
12       return n*fatorial(n -1);
13     end
14   end
15 end
```

Figura 5.34: Execução concorrente do cálculo de fatorial

A sincronia entre dois ou mais fluxos de execução pode ser realizada de duas formas: (1) através de mensagens bloqueantes ou (2) através do contexto.

A figura 5.32 é um exemplo de sincronia através de mensagens. *Start*, após executar os métodos concorrentemente, aguarda que eles sinalizem o término da sua execução através do método *cmwait*.

A figura 5.35 contém um exemplo de sincronia através do contexto. *Start* cria uma instância da entidade *Process* (linha 3) e aguarda (linha 4) que esta entidade insira o conteúdo "Resultado" no contexto. O método construtor de *Process* é executado concorrentemente ao fluxo de execução do elemento *Start*. O construtor de *Process* insere, na linha 10, o conteúdo "Resultado", aguardado por *Start* no contexto; a partir deste momento, *Start* obtém do contexto o conteúdo solicitado, continuando sua execução.

```
1  entity start
2    method constructor()
3      var element p = process.new() [concurrent];
4      io.writeln("Resultado: " + {"MeuContexto"}.cget("Resultado"));
5    end
6  end
7
8  entity process
9    method constructor()
10     {"MeuContexto"}.cpublish("Resultado" => 10 * 2);
11   end
12 end
13
```

Figura 5.35: Sincronia através do contexto

5.9 Gramática básica

A figura 5.36 contém a versão simplificada da gramática, utilizando a notação EBNF. Algumas construções foram abstraídas para permitir uma visão geral e simples da organização da linguagem. A versão completa da gramática pode ser consultada no apêndice A.

```
compilation_unit    : (import_declaration)* (entity_definition)+ EOF
import_declaration  : 'import' IDENTIFIER ('.' IDENTIFIER)* ';'
entity_definition   : 'entity' IDENTIFIER (entity_options)? (valid_context)?
                      (property_definition)* (method_definition|service_definition)+ 'end'
entity_options      : ...
adapted_context     : ...
property_definition : ...
method_definition   : 'method' ('public'|'private')? IDENTIFIER '(' (params_definition)? ')'
                      (returns_definition)?
                      (var_definition)* (statement)* 'end'
service_definition  : 'service' IDENTIFIER '(' (params_definition)? ')' returns_definition
                      (var_definition)* (statement)* 'end'
params_definition   : ...
returns_definition  : ...
var_definition      : ...
statement           : ...
```

Figura 5.36: Versão simplificada da gramática

5.10 Considerações sobre o capítulo

Este capítulo abordou a UbiL, uma linguagem de programação que implementa os conceitos propostos no UOP. UbiL facilita a exploração dos conceitos de computação ubíqua propostos no UOP, diminuindo assim o *gap* semântico existente no desenvolvimento de aplicações ubíquas.

A UbiL facilita o desenvolvimento de aplicativos ubíquos ao auxiliar na implementação das seguintes funcionalidades:

- compartilhamento de conteúdos e serviços, disponibilizados pelos elementos, através dos contextos;
- comunicação e sincronia entre os elementos;
- utilização de vários contextos simultaneamente;
- descoberta de contextos, serviços e conteúdos, durante a execução;

- organização dos contextos em hierarquia, auxiliando na descoberta por informações contextuais mais especializadas;
- obtenção dos membros, serviços e conteúdos de um contexto;
- suporte a adaptação ao contexto;
- implementação de aplicações concorrentes;
- implementação da mobilidade de código.

Além disso, embutir o suporte a adaptação ao contexto, na linguagem, proporciona algumas vantagens:

- o desenvolvedor que utiliza as entidades com suporte a adaptação não precisa saber qual a entidade ou sub-rotina mais adequada ao contexto atual, porque esta decisão está embutida na própria entidade;
- o número de linhas dos programas que utilizam o suporte a adaptação é menor do que os programas equivalentes que não utilizam;
- maior abstração e clareza de código, visto que a cláusula *when* sempre está associada a algum tipo de adaptação ao contexto.

O próximo capítulo aborda a UbiVM, a máquina virtual responsável por executar os aplicativos ubíquos propostos pela UbiL.

Capítulo 6

Ubiquitous Virtual Machine

Este capítulo apresenta a *Ubiquitous Virtual Machine* (abreviadamente, UbiVM), a máquina virtual responsável por executar os aplicativos ubíquos desenvolvidos usando a UbiL. A especificação da UbiVM é focada nos recursos existentes nesta linguagem, que são: contextos, serviços, sensibilidade ao contexto, adaptação ao contexto, mobilidade forte de código e concorrência.

O capítulo está estruturado da seguinte forma. A seção 6.1 apresenta as principais características da UbiVM; a seção 6.2 apresenta as informações sobre os componentes existentes na arquitetura; a seção 6.3 apresenta a definição da linguagem de máquina; a seção 6.4 apresenta como é possível explorar os recursos da UbiVM; por fim a seção 6.5 apresenta as considerações finais deste capítulo.

6.1 Características da UbiVM

A UbiVM é o componente responsável pela independência de hardware e sistema operacional dos aplicativos escritos em UbiL. Para tal cria uma camada de abstração entre os programas e a máquina física em que está sendo executada, possibilitando que um *bytecode* gerado pelo UbiC execute em qualquer plataforma onde exista uma implementação da UbiVM.

A UbiVM é uma **máquina baseada em pilha**, ou seja, não há registradores de máquina de propósito geral. A UbiVM possui uma pilha de operandos, similar as pilhas de linguagens convencionais como C e Pascal, que é utilizada para

armazenar os operandos necessários durante a execução das instruções. Esta pilha é manipulada explicitamente através de instruções que carregam e retiram dados. Por exemplo, para somar dois operandos, utiliza-se uma instrução para carregar o primeiro e o segundo operando na pilha; após isso, utiliza-se outra instrução que retira estes operandos da pilha, executa a operação de soma e armazena o resultado nesta mesma pilha.

Durante a execução das aplicações, a UbiVM gerencia três áreas de dados:

- ***EntityPool* (área de entidades)**: é uma imagem estática das entidades existentes em uma aplicação, contendo as informações sobre suas propriedades, métodos e serviços. É compartilhada por todos os fluxos em execução, sendo similar a área de armazenamento para código compilado de uma linguagem convencional, ou similar ao segmento "texto" de um processo UNIX [Lindholm e Yellin 1999];

- ***ElementPool* (área de elementos)**: é responsável por armazenar a imagem dos elementos em execução. Na instanciação de um elemento, é criada uma entrada nesta área que referencia sua imagem estática (armazenada na *EntityPool*);

- ***ConstantPool* (área de constantes)**: é uma imagem estática das constantes existentes em uma aplicação. Esta área é única para cada aplicativo em execução, sendo similar à tabela de símbolos de um compilador tradicional [Aho, Sethi e Ullman 1988, Lindholm e Yellin 1999]. *ConstantPool* armazena constantes necessárias na descrição das entidades e na execução das instruções existentes na aplicação. Por exemplo, informações como o nome de uma entidade, o nome de um método, constantes numéricas e *strings* são armazenadas nesta área.

EntityPool, *ElementPool* e *ConstantPool* são criadas na inicialização da aplicação, e destruídas somente quando a aplicação finaliza sua execução.

EntityPool e *ConstantPool* são estáticas, ou seja, não são alteradas durante a execução da aplicação. *ElementPool* é dinâmica, ou seja, é alterada durante a execução da aplicação pelas instruções que criam e destroem elementos.

Os aplicativos escritos em UbiL são armazenados em um arquivo padronizado, chamado arquivo UVM. Este arquivo, gerado pelo UbiC, contém a representação binária das entidades (propriedades, métodos e serviços) existentes em um aplicativo escrito em UbiL. A descrição detalhada deste arquivo pode ser vista no apêndice D.

6.2 Arquitetura

A UbiVM cria uma abstração sobre a arquitetura (hardware) bem como sobre o sistema computacional nos quais está sendo executada.

A figura 6.1 contém os componentes presentes na sua arquitetura. São eles:

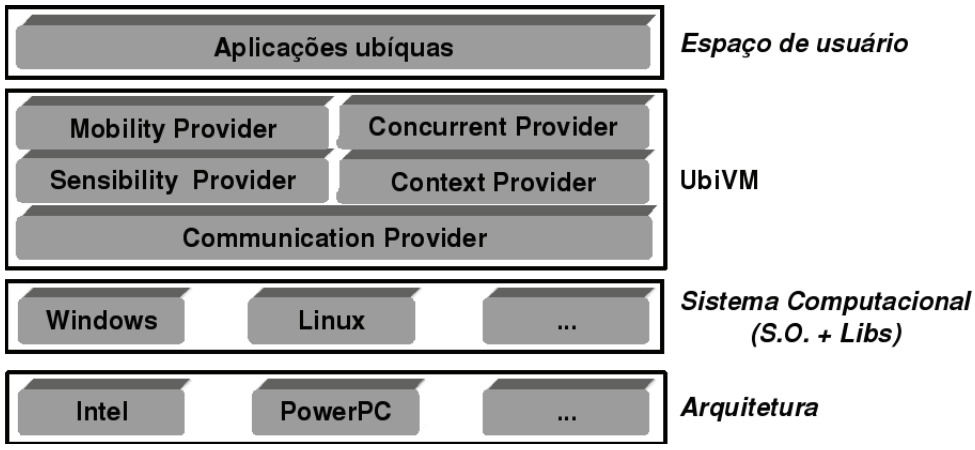


Figura 6.1: Arquitetura da UbiVM

- *Concurrent Provider:* responsável por gerenciar a execução simultânea de vários fluxos de execução;
- *Communication Provider:* responsável por gerenciar a comunicação entre UbiVMs distintas;
- *Context Provider:* responsável por compartilhar os conteúdos e serviços existentes nos contextos públicos;
- *Sensibility Provider:* responsável por gerenciar os sensores que populam os contextos privados com as informações obtidas do ambiente (contexto);

- *Mobility Provider*: responsável por gerenciar a mobilidade de código.

Estes componentes são detalhados nas próximas subseções.

6.2.1 *Concurrent Provider*

Concurrent Provider é o componente responsável por gerenciar a execução simultânea de vários fluxos de execução. Cada fluxo (figura 6.2) possui sua própria pilha, e sempre está associado a uma *thread* de execução.

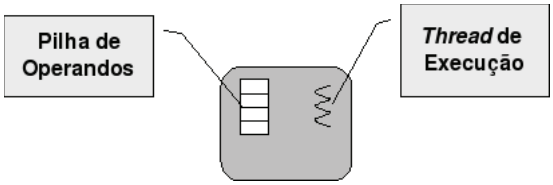


Figura 6.2: Fluxo de execução

A figura 6.3 contém os fluxos de execução existentes durante a execução do aplicativo *Aplicativo1*. *Start* é o fluxo do elemento principal criado implicitamente pela UbiVM ao iniciar a execução do aplicativo. *Aplicativo1* invoca os métodos concorrentes *Método1* e *Método2*, gerando dois novos fluxos que executam concorrentemente aos outros fluxos em execução.

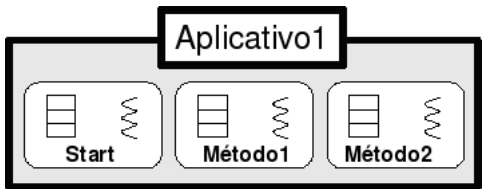


Figura 6.3: Exploração da concorrência

A sincronia entre os fluxos de execução pode ocorrer de duas formas:

- Forma síncrona: ocorre diretamente entre os fluxos, através do uso de mensagens. Na figura 6.4, os N fluxos do aplicativo *Aplicativo1* utilizam mensagens para realizar a sincronia;
- Forma assíncrona: ocorre indiretamente entre os fluxos, onde o contexto é utilizado para realizar a sincronia. Na figura 6.5, os aplicativos utilizam

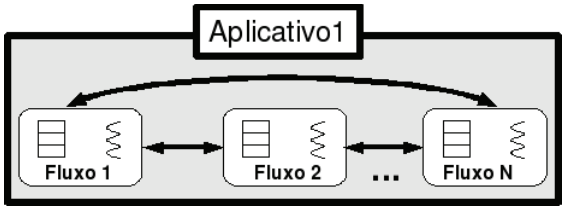


Figura 6.4: Sincronia através de mensagens

o contexto *Contexto1* para realizar sua sincronia. No passo 1, o *Fluxo 1* do aplicativo aguarda por uma informação no contexto, informação esta que é publicada no passo 2 pelo *Fluxo 2* do outro aplicativo. Assim, os contextos tornam-se um espaço tanto para compartilhamento de informações contextuais bem como um meio para realizar a sincronia entre fluxos de execução independentes.

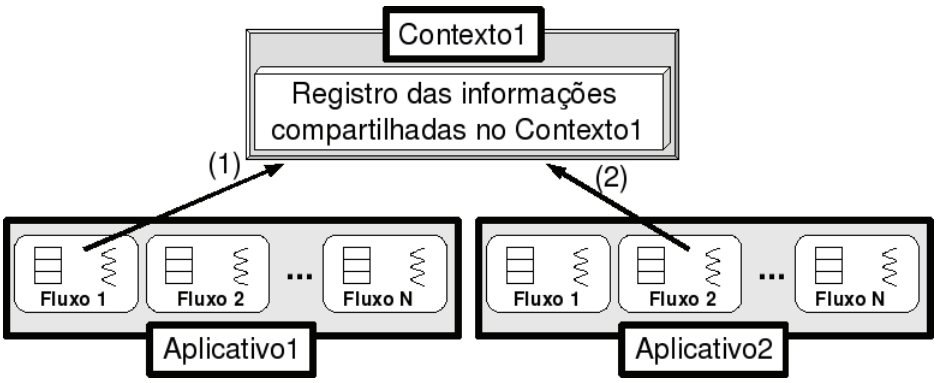


Figura 6.5: Sincronia através do contexto

6.2.2 *Communication Provider*

Communication Provider é o componente responsável por realizar a comunicação entre UbiVMs distintas. Na figura 6.6, duas UbiVMs, sendo executadas em nodos distintos (*Nodo 1* e *Nodo 2*), comunicam-se através de um canal de comunicação. O canal de comunicação utilizado pode ser a rede local, *WiFi*, *bluetooth*, enfim, qualquer tecnologia disponível no momento para a qual a UbiVM tenha suporte.

Através deste canal são enviadas e recebidas as requisições que atuam sobre os membros, os conteúdos e os serviços do contexto. Após receber uma

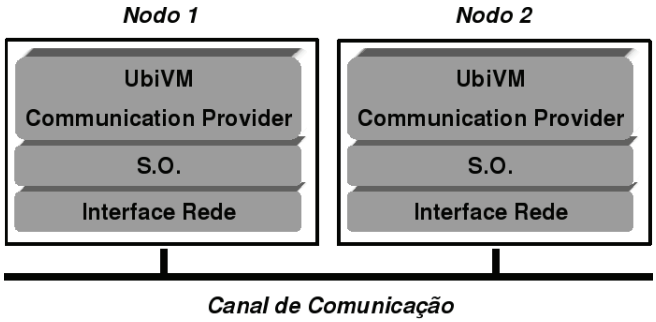


Figura 6.6: *Communication Provider*

requisição (figura 6.7), *Communication Provider* repassa a requisição para o *provider* apropriado (denominado *provider executor*). Após a resposta do *executor*, *Communication Provider* encapsula e envia a resposta da requisição.

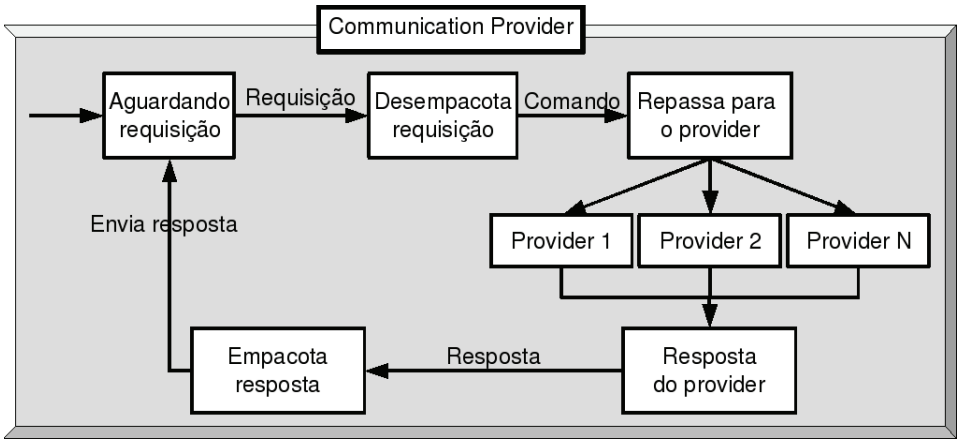


Figura 6.7: Processamento de requisições

6.2.3 *Context Provider*

Context Provider é o componente responsável por compartilhar as informações existentes nos contextos públicos. *Context Provider* implementa uma arquitetura distribuída onde cada aplicativo mantém registro apenas das informações disponibilizadas pelos seus elementos. Informações disponibilizadas por elementos de outras aplicações, quando necessárias, devem ser obtidas através do *Communication Provider*.

As requisições pertinentes a este componente são classificadas em três

categorias:

- Referentes aos membros: lista, associa e retira elementos;
- Referentes aos conteúdos: publica, busca, lista, obtém e remove conteúdos;
- Referentes aos serviços: publica, busca, lista, obtém, remove e executa serviços.

Além disso, *Context Provider* mantém, para cada contexto, três informações:

- *member_list*: contém a lista dos membros do contexto, e é atualizada por instruções que associam e removem elementos do contexto;
- *content_list*: contém a lista dos conteúdos compartilhados, e é atualizada por instruções que publicam e removem conteúdos;
- *service_list*: contém a lista dos serviços compartilhados, e é atualizada por instruções que publicam e removem serviços.

Durante a execução da aplicação, *member_list*, *content_list* e *service_list* são utilizadas para determinar a existência (ou não) de determinados membros, conteúdos e serviços.

6.2.4 *Sensibility Provider*

Sensibility provider é o componente responsável por gerenciar os sensores. Os sensores são responsáveis por obter e monitorar as alterações nas informações do ambiente, disponibilizando-as nos contextos privados.

A figura 6.8 contém um cenário onde existem quatro sensores monitorando informações do usuário: *symbolic location*, que obtém a localização simbólica; *physical location*, que obtém a localização física; *heartbeat*, que obtém o batimento cardíaco e *pressure*, que obtém a pressão sanguínea. Estes sensores, ao detectarem alterações pertinentes no ambiente, sinalizam o *Sensibility Provider* sobre as alterações ocorridas.

Sensibility provider disponibiliza uma interface que possibilita a execução e monitoramento destes sensores, possibilitando que as informações obtidas no ambiente sejam refletidas nos contextos privados e sinalizadas ao *provider*.

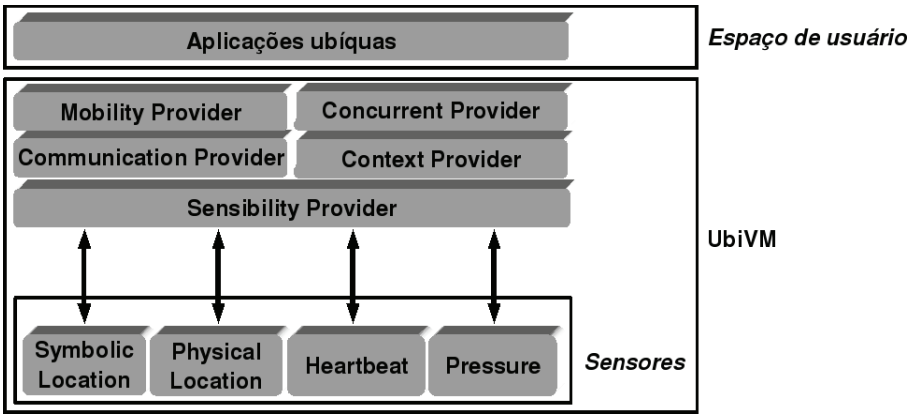


Figura 6.8: Sensores em execução

6.2.5 *Mobility Provider*

Mobility Provider é o componente responsável por realizar a mobilidade forte de código durante a execução das aplicações.

A mobilidade do aplicativo ocorre entre duas UbiVMs distintas: a UbiVM origem e a UbiVM destino. Para tal, ocorrem as seguintes etapas:

1. os fluxos de execução existentes no aplicativo são suspensos na UbiVM origem;
2. os elementos, junto com o estado das suas propriedades e informações contextuais, são serializadas;
3. o *bytecode*, contendo as instruções e suas constantes, é serializado;
4. os elementos serializados, em conjunto com o *bytecode* serializado, são enviados para a UbiVM destino;
5. o novo aplicativo é criado na UbiVM destino, sendo seu estado interno restaurado;
6. o estado interno dos elementos é restaurado;
7. o aplicativo na UbiVM origem é extinto;
8. os fluxos de execução do novo aplicativo são liberados para continuarem sua execução.

Após estas etapas, o novo aplicativo continua sua execução na UbiVM destino, a partir do ponto onde foi suspenso.

6.3 UbiAssembly

UbiAssembly (abreviadamente UbiA) é a linguagem de máquina da UbiVM. É uma representação textual das instruções existentes no *bytecode*. UbiA é uma linguagem *assembly* que mantém as construções de alto nível da UbiL como entidades, propriedades, métodos e serviços. O seu propósito é facilitar o entendimento da UbiVM e de suas instruções.

A figura 6.9 contém a estrutura de um programa em UbiA. Um programa é composto pela descrição de suas entidades, onde cada entidade possui propriedades e sub-rotinas. As sub-rotinas, por sua vez, possuem parâmetros, variáveis locais, resultados e instruções.

```
.entity <nome_entidade>
  .prop <property_type> <property_name>
  .method <method_name>
    .param  <param_type> <param_name>
    .var    <var_type>   <var_name>
    .result <result_type>
    <instruções>
  .end
  .service <service_name>
    .param  <param_type> <param_name>
    .var    <var_type>   <var_name>
    .result <result_type>
    <instruções>
  .end
.end
```

Figura 6.9: Estrutura de um programa em UbiAssembly

A figura 6.10 exemplifica o formato das instruções. Cada instrução é composta por um *label*, um mnemônico e um operando. ***Label*** é um identificador numérico, precedido pelo símbolo de dois pontos (":"). É opcional, e deve ser definido quando alguma instrução de salto direciona o fluxo de execução para esta instrução. **Mnemônico** é a representação textual de um *opcode*. Por fim, **Operando** é utilizado na execução da instrução. É opcional, e o mnemônico determina se deve ou não existir. Neste exemplo, *ldconst*, *add* e *jmp* são exemplos de mnemônicos. *ldconst* e *jmp* recebem um operando, enquanto que *add* não recebe operandos. *jmp* é uma instrução de salto, e seu operando indica o *label* da

instrução para onde o fluxo de execução será desviado. No exemplo, *jmp* desvia o fluxo para a instrução com o *label* ":1", existente na linha 1.

```
1  :1  ldconst 1
2      ldconst 2
3      add
4      jmp 1
```

Figura 6.10: Formato das instruções

O apêndice C apresenta a descrição detalhada das instruções presentes na UbiVM. Este tópico não faz parte deste capítulo por conter uma descrição extensiva destas instruções.

Para facilitar o entendimento, sempre que pertinente, os exemplos deste capítulo estão escritos em UbiL (figura a) e sua tradução para UbiA (figura b).

A figura 6.11 contém o exemplo "Olá mundo !!!". A instrução *ldconst* carrega literais na pilha enquanto que a instrução *lcall* realiza chamadas as bibliotecas da UbiL. Neste exemplo, a instrução *ldconst* (linha 3) carrega na pilha o literal "Olá mundo !!!" (constante que será exibida pela função *io.writeln*); a instrução *ldconst* (linha 4) carrega na pilha o literal *1* (o número de argumentos a serem exibidos pela função *io.writeln*); por fim, a instrução *lcall* (linha 5) realiza uma chamada de biblioteca para *io.writeln*, que retira da pilha o número de argumentos (no exemplo, o literal *1*) e os argumentos a serem exibidos (no exemplo, "Olá mundo !!!"), exibindo-os na saída padrão.

```
1  entity start
2    method constructor()
3      io.writeln( "Olá mundo !!!" );
4    end
5  end
```

(a)

```
.entity start
  .method constructor:nil
    ldconst "Olá mundo !!!"
    ldconst 1
    lcall io.writeln
  .end
.end
```

(b)

Figura 6.11: "Olá mundo !!!" em UbiL e UbiA

Para possibilitar a sobrecarga de métodos (explicada logo a seguir), tanto na definição quanto na invocação, a assinatura completa do método deve ser utilizada. Neste exemplo, o método *constructor*, por não possuir argumentos, é sucedido por ":nil".

A figura 6.12 contém um exemplo com o estado da pilha durante a execução de um trecho de código. As figuras 6.12(a), 6.12(b) e 6.12(c) contém,

respectivamente, dois comandos em UbiL, sua tradução para UbiA e o estado da pilha após a execução de cada instrução. A instrução *add* retira dois literais da pilha, executa a operação de soma e insere o resultado na pilha novamente; a instrução *mul* retira dois literais da pilha, executa a operação de multiplicação e insere o resultado na pilha novamente; a instrução *stvar* retira um literal da pilha e armazena na variável especificada; por fim, a instrução *ldvar* carrega o conteúdo de uma variável na pilha. Neste exemplo, x é uma variável local; a instrução *"ldconst 2"* (linha 1) carrega o literal *"2"* na pilha; *"ldconst 7"* (linha 2) carrega o literal *"7"* na pilha; *"ldconst 3"* (linha 3) carrega o literal *"3"* na pilha; *"add"* (linha 4) retira os literais *"3"* e *"7"* e armazena o resultado da soma (*"10"*) na pilha; *"mul"* (linha 5) retira os literais *"2"* e *"10"* e armazena o resultado da multiplicação (*"20"*) na pilha; *"stvar x"* (linha 6) retira o literal *"20"* e armazena na variável x; *"ldvar x"* (linha 7) carrega o conteúdo da variável x ("20") na pilha; *"ldconst 1"* (linha 8) carrega o literal *"1"* na pilha; por fim, *"lcall io.write"* retira o número de argumentos (*"1"*) e o argumento a ser impresso (*"20"*) da pilha, exibindo-o na saída padrão.

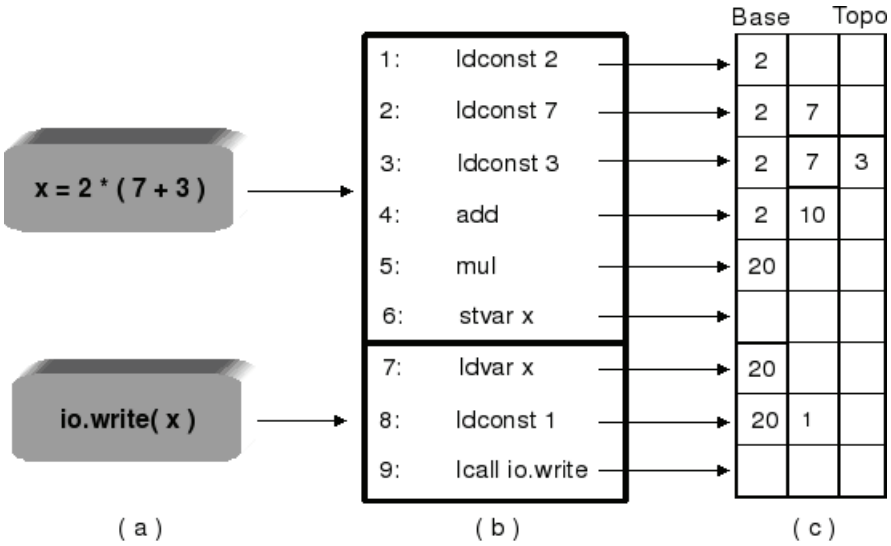


Figura 6.12: Estado da pilha durante a execução

6.4 Recursos disponíveis

Esta seção detalha como explorar os recursos existentes na UbiVM.

6.4.1 Variáveis

A figura 6.13 contém um exemplo sobre como utilizar variáveis. A cláusula *var* define o tipo e o nome de cada variável; a instrução *ldvar* carrega o conteúdo de uma variável na pilha; por fim, a instrução *stvar* retira um literal da pilha e o armazena em uma variável. Neste exemplo, na linha 3 é definida a variável *x*, do tipo *int*; as instruções *ldconst* das linhas 4 e 5 carregam os literais "10" e "20" na pilha; a instrução *add* (linha 6) retira dois literais da pilha ("10" e "20") e armazena o resultado da soma destes literais ("30") nesta mesma pilha; a instrução *stvar* (linha 7) retira um literal da pilha ("30") e armazena na variável local *x*; a instrução *ldvar* (linha 8) carrega o conteúdo da variável local *x* ("30") na pilha; a instrução *ldconst* (linha 9) carrega na pilha o número de argumentos que serão exibidos por *io.writeln*; por fim, a instrução lcall (linha 10) executa a função *writeln* da biblioteca *io*, que retira da pilha o número de argumentos ("1") e o argumento ("30"), exibindo-o na saída padrão.

```
1 entity start
2   method constructor()
3     var int x = 10 + 20;
4     io.writeln(x);
5   end
6 end
```

(a)

```
.entity start                          1
  .method constructor:nil              2
    .var int x                         3
    ldconst 10                         4
    ldconst 20                         5
    add                                6
    stvar x                            7
    ldvar x                            8
    ldconst 1                          9
    lcall io.writeln                   10
  .end                                 11
.end                                   12
```

(b)

Figura 6.13: Uso de variáveis

6.4.2 Métodos

A figura 6.14 contém um exemplo sobre como enviar mensagens. Neste exemplo, o elemento envia mensagens para ele mesmo; para tal, primeiramente é carregado o elemento em execução na pilha através da instrução *ldself* e, após isso, a instrução *mcall* é executada, informando-se a mensagem que será enviada. Neste exemplo, a linha 8 carrega o elemento atual na pilha, enquanto que na linha 9 é enviada a mensagem *m1:nil*.

A figura 6.15 contém um exemplo com um método recebendo um argumento.

```
1 entity start
2   method m1()
3     io.writeln("m1");
4   end
5   method constructor()
6     m1();
7   end
8 end
```

(a)

```
.entity start
  .method m1:nil
    ldconst "m1"
    ldconst 1
    lcall io.writeln
  .end
  .method constructor:nil
    ldself
    mcall m1:nil
  .end
.end
```

(b)

Figura 6.14: Envio de mensagens

A cláusula *param* define o tipo e o nome de cada parâmetro; a instrução *ldparam* carrega o conteúdo de um parâmetro na pilha. Neste exemplo, a instrução *ldconst* (linha 3) carrega o literal "oi" na pilha; a instrução *ldself* (linha 4) carrega na pilha o elemento que irá receber a mensagem, enquanto que *mcall* (linha 5) envia a mensagem *show:string*; a instrução *ldparam* (linha 10) carrega na pilha o parâmetro "oi", que foi passado como argumento para o método, e será exibido pela instrução *lcall* (linha 12).

```
1 entity start
2   method constructor()
3     show("oi");
4   end
5   method show(string msg)
6     io.writeln("msg=", msg);
7   end
8 end
```

(a)

```
.entity start
  .method constructor:nil
    ldconst "oi"
    ldself
    mcall show:string
  .end
  .method show:string
    .param string msg
    ldconst "msg="
    ldparam msg
    ldconst 2
    lcall io.writeln
  .end
.end
```

(b)

Figura 6.15: Método com argumentos

A figura 6.16 contém um exemplo com um método que recebe dois argumentos, e retorna um resultado. A cláusula *result* define o tipo de cada resultado; a instrução *stresult* retira um literal da pilha e armazena no resultado especificado. Neste exemplo, o método *soma* retorna um resultado; as linhas 12 e 13 empilham os literais "10" e "20", respectivamente; a instrução *mcall* (linha 15) invoca o método *soma* definido na linha 2; por fim, após retornar da execução do método, o resultado ("30") está na pilha, e é utilizado na função *io.writeln*, na linha 17.

```
1 entity start
2   method soma(int x, int y) [int]
3     return x+y;
4   end
5   method constructor()
6     io.writeln(soma(10,20));
7   end
8 end
```

(a)

```
.entity start
  .method soma:int:int
    .param int x
    .param int y
    .result int
    ldparam x
    ldparam y
    add
    stresult 0
  .end
  .method constructor:nil
    ldconst 10
    ldconst 20
    ldself
    mcall soma:int:int
    ldconst 1
    lcall io.writeln
  .end
.end
```

(b)

Figura 6.16: Método retornando um resultado

A figura 6.17 contém um exemplo com um método retornando dois resultados. Neste exemplo, as linhas 3 e 4 definem que o método recebe os argumentos x e y; as linhas 5 e 6 definem que o método retorna dois resultados do tipo *int*; nas linhas 19 e 20 os argumentos para o método *calcula* são empilhados; a instrução *mcall* (linha 22) invoca o método *calcula*; após retornar da execução do método, os resultados estão na pilha, e são desempilhados pelas instruções das linhas 23 e 24; por fim, as linhas entre 25 e 30 exibem os resultados.

A figura 6.18 contém um exemplo com sobrecarga de métodos. Cada método é gerado com a sua assinatura pelo UbiC (linhas 2, 9 e 16). Além disso, na sua invocação, a assinatura completa do método é utilizada (linhas 19 e 22).

6.4.3 Elementos

A figura 6.19 contém um exemplo sobre como instanciar novos elementos e invocar os seus métodos. A instrução *newelem* cria uma nova instância de uma entidade e insere na pilha. Neste exemplo, *newelem* (linha 12) cria uma nova instância da entidade *sample*, que é armazenada na variável *ex* pela instrução da linha 13; a instrução *ldvar* (linha 14) carrega o elemento que contém o método a ser invocado; por fim, a instrução *mcall* (linha 15) invoca o método *print*.

A figura 6.20 contém um exemplo sobre como destruir elementos. A instrução *delelem* retira o elemento a ser destruído da pilha. Neste exemplo, *newelem*

```
1  entity start
2    method calcula(int x, int y) [int, int]
3      return(x + y, x - y);
4    end
5    method constructor()
6      var int a, s;
7      a, s = calcula(10,20);
8      io.writeln("Adicao=", a,
9          " Subtracao=", s);
10   end
11 end
```

(a)

```
.entity start                                1
  .method calcula:int:int                    2
    .param int x                             3
    .param int y                             4
    .result int                              5
    .result int                              6
    ldparam x                                7
    ldparam y                                8
    sum                                      9
    stresult 0                               10
    ldparam x                                11
    ldparam y                                12
    sub                                      13
    stresult 1                               14
  .end                                       15
  .method constructor:nil                    16
    .var int a                               17
    .var int s                               18
    ldconst 10                               19
    ldconst 20                               20
    ldself                                   21
    mcall calcula:int:int                    22
    stvar a                                  23
    stvar s                                  24
    ldconst "Adicao="                        25
    ldvar a                                  26
    ldconst " Subtracao="                    27
    ldvar s                                  28
    ldconst 4                                29
    lcall io.writeln                         30
  .end                                       31
.end                                         32
```

(b)

Figura 6.17: Método retornando dois resultados

(linha 12) cria uma nova instância da entidade *sample*, que é armazenada na variável *ex* pela instrução da linha 13; a instrução *ldvar* (linha 16) carrega o elemento que será destruído pela instrução *delelem*, na linha 17.

6.4.4 Propriedades

A figura 6.21 contém um exemplo sobre como utilizar propriedades. A cláusula *prop* define o tipo e o nome de cada propriedade; a instrução *stprop* retira um literal da pilha e armazena na propriedade informada; por fim, *ldprop* carrega na pilha o literal armazenado na propriedade especificada. Neste exemplo, a linha 2 define a propriedade *txt* do tipo *string*; a instrução *ldconst*, na linha 4, carrega na pilha o literal "Mensagem" que será armazenado na propriedade *txt* pela instrução *stprop* da linha 5; a instrução *ldprop*, na linha 6, carrega na pilha o conteúdo da propriedade *txt* (neste exemplo, "Mensagem"); a instrução *ldconst*,

```
1  entity start
2    method print(string x)
3      io.writeln("String: ", x);
4    end
5    method print(int x)
6      io.writeln("Numero: ", x);
7    end
8    method constructor()
9      print("Oi");
10     print(10);
11   end
12 end
```

(a)

```
.entity start                          1
  .method print:string                 2
    .param string x                    3
    ldconst "String: "                 4
    ldparam x                          5
    ldconst 2                          6
    lcall io.writeln                   7
  .end                                 8
  .method print:int                    9
    .param int x                       10
    ldconst "Numero: "                 11
    ldparam x                          12
    ldconst 2                          13
    lcall io.writeln                   14
  .end                                 15
  .method constructor:nil              16
    ldconst "Oi"                       17
    ldself                             18
    mcall print:string                 19
    ldconst 10                         20
    ldself                             21
    mcall print:int                    22
  .end                                 23
.end                                   24
```

(b)

Figura 6.18: Sobrecarga de métodos

```
1  entity sample
2    method print()
3      io.writeln("Executando print...");
4    end
5  end
6
7  entity start
8    method constructor()
9      var element ex = sample.new();
10     ex.print();
11   end
12 end
13
```

(a)

```
.entity sample                         1
  .method print:nil                    2
    ldconst "Executando print..."      3
    ldconst 1                          4
    lcall io.writeln                   5
  .end                                 6
.end                                   7
                                       8
.entity start                          9
  .method constructor:nil              10
    .var element ex                    11
    newelem sample                     12
    stvar ex                           13
    ldvar ex                           14
    mcall print:nil                    15
  .end                                 16
.end                                   17
```

(b)

Figura 6.19: Instanciando elementos

na linha 7, carrega na pilha o literal "1"; por fim, os últimos dois literais da pilha são utilizados pela função *io.writeln*, na linha 8, que exibe "Mensagem" na saída padrão.

6.4.5 Herança entre entidades

A figura 6.22 contém um exemplo de herança. Este exemplo define a entidade *b*, que estende *a*, herdando desta o método *x*, e definindo o novo método *y*. A

```
1  entity exemplo
2    method print()
3      io.writeln("Executando print...");
4    end
5  end
6
7  entity start
8    method constructor()
9      var int loop;
10     var element ex = exemplo.new();
11     ex.print();
12     ex.delete();
13   end
14 end
```

(a)

```
1  .entity exemplo
2    .method print
3      ldconst "Executando print..."
4      ldconst 1
5      lcall io.writeln
6    .end
7  .end
8
9  .entity start
10   .method constructor
11     .var element ex
12     newelem exemplo
13     stvar ex
14     ldvar ex
15     mcall print
16     ldvar ex
17     delelem
18   .end
19 .end
```

(b)

Figura 6.20: Destruindo elementos

```
1  entity start
2    prop string txt;
3    method constructor()
4      txt = "Mensagem";
5      io.writeln(txt);
6    end
7  end
```

(a)

```
1  .entity start
2    .prop string txt
3    .method constructor:nil
4      ldconst "Mensagem"
5      stprop txt
6      ldprop txt
7      ldconst 1
8      lcall io.writeln
9    .end
10 .end
```

(b)

Figura 6.21: Uso de propriedades

cláusula *inherits* identifica a herança; a instrução *super* indica que o método original (método que foi sobrescrito) deve ser chamado. Neste exemplo, a instrução *mcall*, na linha 30, exibe a mensagem "Executando a::x"; a instrução *mcall* na linha 32 exibe a mensagem "Executando a::x"; por fim, a instrução *mcall* na linha 34 exibe a mensagem "Executando b::y".

6.4.6 Métodos polimórficos

Na UbiL o polimorfismo é implementado através do uso de funções virtuais. A UbiVM, para prover este suporte, implementa uma técnica de *late binding* conhecida como *virtual table* (*vtable*). *Vtable* [Milewski 2001] é uma tabela de busca de funções utilizada para resolver chamadas de funções em uma forma de ligação tardia.

Utilizando *vtables*, a UbiVM garante que, durante a execução da aplicação,

```
1  entity a
2    method x()
3      io.writeln("Executando a::x");
4    end
5  end
6
7  entity b inherits a
8    method y()
9      io.writeln("Executando b::y");
10   end
11 end
12
13 entity start
14   method constructor()
15     var element ea = a.new();
16     var element eb = b.new();
17     ea.x();
18     eb.x();
19     eb.y();
20   end
21 end
```

(a)

```
.entity a                                          1
  .method x:nil                                    2
    ldconst "Executando a::x"                      3
    ldconst 1                                      4
    lcall io.writeln                               5
  .end                                             6
.end                                               7
                                                   8
.entity b                                          9
  .inherits a                                      10
  .method x:nil                                    11
    super                                          12
  .end                                             13
  .method y:nil                                    14
    ldconst "Executando b::y"                      15
    ldconst 1                                      16
    lcall io.writeln                               17
  .end                                             18
.end                                               19
                                                   20
.entity start                                      21
  .method constructor:nil                          22
    .var element ea                                23
    .var element eb                                24
    newelem a                                      25
    stvar ea                                       26
    newelem b                                      27
    stvar eb                                       28
    ldvar ea                                       29
    mcall x:nil                                    30
    ldvar eb                                       31
    mcall x:nil                                    32
    ldvar eb                                       33
    mcall y:nil                                    34
  .end                                             35
.end                                               36
```

(b)

Figura 6.22: Herança

as chamadas de métodos serão corretamente resolvidas para a função virtual apropriada.

6.4.7 Clonagem de elementos

A clonagem entre elementos ocorre durante a execução e cria uma cópia das propriedades, métodos e serviços do elemento.

A figura 6.23 contém um exemplo onde um elemento é clonado, e alterações posteriores em um elemento não são refletidas no outro. A instrução *clone* realiza a clonagem de elementos. Neste exemplo, as linhas entre 19 e 23 criam o elemento *t1*, a partir da entidade *test*, e inicializam sua propriedade x com o valor "10"; as linhas entre 24 e 26 clonam o elemento *t1*, criando o elemento *t2*; as linhas

entre 27 e 32 exibem o valor da propriedade x de *t1* e *t2*, que, após a clonagem, são iguais ("10"); as linhas entre 33 e 38 alteram a propriedade *x*, de ambos os elementos; por fim, as linhas entre 39 e 46 exibem os valores de *t1.x* e *t2.x*, que são, respectivamente, "1" e "2".

```
1  entity test
2    prop int x get set;
3  end
4
5  entity start
6    method constructor()
7      var element t1 = test.new();
8      t1.x = 10;
9      var element t2 = t1.clone();
10     io.writeln(t1.x, t2.x);
11     t1.x = 1;
12     t2.x = 2;
13     io.writeln(t1.x, t2.x);
14   end
15 end
```

(a)

```
.entity test                          1
  .prop int x                         2
  .method __get_x:nil                 3
    .result int                       4
    ldprop x                          5
    stresult 0                        6
  .end                                7
  .method __set_x:int                 8
    .param int value                  9
    ldparam value                     10
    stprop x                          11
  .end                                12
.end                                  13
                                      14
.entity start                         15
  .method constructor:nil             16
    .var element t1                   17
    .var element t2                   18
    newelem test                      19
    stvar t1                          20
    ldvar t1                          21
    ldconst 10                        22
    mcall __set_x:int                 23
    ldvar t1                          24
    clone                             25
    stvar t2                          26
    ldvar t1                          27
    mcall __get_x:nil                 28
    ldvar t2                          29
    mcall __get_x:nil                 30
    ldconst 2                         31
    lcall io.writeln                  32
    ldvar t1                          33
    ldconst 1                         34
    mcall __set_x:int                 35
    ldvar t2                          36
    ldconst 2                         37
    mcall __set_x:int                 38
    ldvar t1                          39
    mcall __get_x:nil                 40
    ldvar t2                          41
    mcall __get_x:nil                 42
    ldconst 2                         43
    lcall io.writeln                  44
  .end                                45
.end                                  46
```

(b)

Figura 6.23: Clonagem de elementos

6.4.8 Contextos privados

Contextos privados armazenam informações sobre a aplicação em execução.

A figura 6.24 contém um exemplo que exibe algumas informações existentes nos contextos privados. A instrução *stcontexti* retira um literal da pilha e armazena no contexto enquanto que *ldcontexti* carrega a informação de um contexto na pilha.

```
1  entity start
2    method constructor()
3      identity.name = "Alex";
4      io.writeln("Host atual : " +
5          identity.host);
6      io.writeln("Onde estou : " +
7          location.symbolic);
8    end
9  end
```

(a)

```
1  .entity start
2    .method constructor:nil
3      ldconst "Alex"
4      stcontexti "identity.name"
5      ldcontexti "identity.host"
6      ldconst "Host atual : "
7      ldconst 2
8      lcall io.writeln
9      ldcontexti "location.symbolic"
10     ldconst "Onde estou : "
11     ldconst 2
12     lcall io.writeln
13   .end
14 .end
```

(b)

Figura 6.24: Informações existentes nos contextos privados

Caso seja definido um valor para uma informação inexistente em um contexto, esta informação automaticamente é criada e inicializada com o valor definido; caso seja solicitado o valor de uma informação que não existe no contexto, o valor nulo (*nil*) é retornado como resultado da operação.

A forma como as informações dos contextos privados são preenchidas depende do tipo do contexto. As informações de *host*, *device*, *ip* e *mac_address*, existentes no contexto *identity*, bem como as informações do contexto *time*, são mantidas atualizadas pela UbiVM. A informação do contexto *activity* é populada pelo aplicativo em execução, podendo ser informada pelo usuário, obtida através de sensores ou obtida através de serviços existentes nos contextos. As informações existentes em *location.physical* são obtidas através de sensores que se comunicam com dispositivos específicos que fornecem localização física. A informação *location.symbolic* é populada pelo aplicativo em execução, podendo ser informada pelo usuário (por exemplo, "No trem, indo para casa"), obtida através de serviços existentes nos contextos, ou populada por sensores em execução na UbiVM.

6.4.9 Conteúdos compartilhados

A figura 6.25 contém um exemplo onde um aplicativo publica um conteúdo no contexto "MeuContexto". A instrução *cpublish*, para publicar um conteúdo, retira o nome do contexto e o conteúdo da pilha. Neste exemplo, as linhas entre 3 e 8 contém as instruções necessárias para publicar o conteúdo, com chave "Numero 1", e valor "54321", no contexto "MeuContexto"; a linha 3 carrega na pilha o nome do contexto ("MeuContexto"); a linha 4 carrega na pilha a chave do conteúdo ("Numero 1"); a linha 5 carrega na pilha o valor do conteúdo ("54321"); a linha 6 carrega na pilha o número de valores da chave ("1"); a linha 7 carrega na pilha o número de valores do resultado ("1"); por fim, a instrução da linha 8 retira seus argumentos da pilha e publica o conteúdo. Na linha 9, a função "io.key_press" trava o fluxo de execução até que uma tecla seja pressionada.

```
1 entity start
2   method constructor()
3     {"MeuContexto"}.cpublish(
4         "Numero 1" => 54321);
5     io.key_press();
6   end
7 end
```

(a)

```
.entity start                                  1
  .method constructor:nil                      2
    ldconst "MeuContexto"                      3
    ldconst "Numero 1"                         4
    ldconst 54321                              5
    ldconst 1                                  6
    ldconst 1                                  7
    cpublish                                   8
    lcall io.key_press                         9
  .end                                         10
.end                                           11
```

(b)

Figura 6.25: Publicando conteúdos

A figura 6.26 contém um exemplo onde um aplicativo obtém, no contexto "MeuContexto", o conteúdo "Numero 1" que foi publicado pelo aplicativo da figura 6.25. A instrução *cget*, para obter um conteúdo, retira o nome do contexto e a chave do conteúdo da pilha. Neste exemplo, as linhas entre 4 e 7 contém as instruções necessárias para buscar o conteúdo "Numero 1", no contexto "MeuContexto"; a linha 4 carrega na pilha o nome do contexto ("MeuContexto"); a linha 5 carrega na pilha a chave do conteúdo ("Numero 1"); a linha 6 carrega na pilha o número de valores da chave ("1"); por fim, a instrução da linha 7 retira seus argumentos da pilha e envia a solicitação para obter o conteúdo, deixando-o na pilha.

Os conteúdos publicados ficam disponíveis até que o elemento que o publicou seja destruído ou algum elemento decida removê-lo.

```
1  entity start
2    method constructor()
3      io.writeln("Informacao publicada: " +
4          {"MeuContexto"}.cget("Numero 1");
5    end
6  end
```

(a)

```
.entity start
  .method constructor:nil
    ldconst "Informacao publicada: "
    ldconst "MeuContexto"
    ldconst "Numero 1"
    ldconst 1
    cget
    ldconst 2
    lcall io.writeln
  end
end
```

(b)

Figura 6.26: Obtendo conteúdos

6.4.10 Serviços compartilhados

A figura 6.27 contém um exemplo onde um aplicativo publica dois serviços no contexto "MeuContexto". A instrução *spublish*, para publicar um serviço, retira o nome do contexto e do serviço da pilha. Neste exemplo, as linhas entre 3 e 5 contém as instruções necessárias para publicar o serviço "service1:int", no contexto "MeuContexto"; a linha 3 carrega na pilha o nome do contexto ("MeuContexto"); a linha 4 carrega na pilha o nome do serviço ("service1:int"); por fim, a instrução da linha 5 retira seus argumentos da pilha e publica o serviço. As linhas entre 6 e 8 contém as instruções necessárias para publicar o serviço "service2:string", no contexto "MeuContexto". Na linha 9, a função "io.key_press" aguarda que uma tecla seja pressionada.

A figura 6.28 contém um exemplo onde um aplicativo busca no contexto "MeuContexto" quem pode prestar os serviços que necessita, serviços estes que foram publicados pelo aplicativo da figura 6.27. A instrução *srun*, para executar um serviço, retira o nome do contexto e do serviço da pilha. Neste exemplo, as linhas entre 4 e 8 contém as instruções necessárias para solicitar a execução do serviço "service1:int", com o argumento "5", no contexto "MeuContexto"; a linha 4 carrega na pilha o argumento que será passado como parâmetro ao serviço ("5"); a linha 5 carrega na pilha o número de argumentos de *service1* ("1"); a linha 6 carrega na pilha o nome do contexto ("MeuContexto"); a linha 7 carrega na pilha o nome do serviço ("service1:int"); por fim, a instrução da linha 8 retira seus argumentos da pilha e envia a solicitação de execução do serviço. As linhas entre 12 e 16 contém as instruções para executar o serviço "service2:string", com o argumento "Oi", no contexto "MeuContexto".

```
entity start
  method constructor()
    {"MeuContexto"}.spublish("service1");
    {"MeuContexto"}.spublish("service2");
    io.key_press();
  end
  service service1(int value) [int]
    return value*10;
  end
  service service2(string value) [string]
    return value + " !!!";
  end
end
```

(a)

```
.entity start
  .method constructor:nil
    ldconst "MeuContexto"
    ldconst "service1:int"
    spublish
    ldconst "MeuContexto"
    ldconst "service2:string"
    spublish
    lcall io.key_press
  .end
  .service service1:int
    .param int value
    .result int
    ldparam value
    ldconst 10
    mul
    stresult 0
  .end
  .service service2:string
    .param string value
    .result string
    ldparam value
    ldconst " !!!"
    add
    stresult 0
  .end
.end
```

(b)

Figura 6.27: Publicando serviços

```
entity start
  method constructor()
    io.writeln("Resultado do servico 1: " +
      {"MeuContexto"}.srun("service1",5));
    io.writeln("Resultado do servico 2: " +
      {"MeuContexto"}.srun("service2","Oi"));
  end
end
```

(a)

```
.entity start
  .method constructor:nil
    ldconst "Resultado do servico 1: "
    ldconst 5
    ldconst 1
    ldconst "MeuContexto"
    ldconst "service1:int
    srun
    ldconst 2
    lcall io.writeln
    ldconst "Resultado do servico 2: "
    ldconst "Oi"
    ldconst 1
    ldconst "MeuContexto"
    ldconst "service2:string
    srun
    ldconst 2
    lcall io.writeln
  .end
.end
```

(b)

Figura 6.28: Executando serviços

Os serviços publicados ficam disponíveis até que o elemento que o publicou seja destruído ou decida removê-lo.

6.4.11 Contextos hierárquicos

A figura 6.29 contém um exemplo onde, a partir de um caminho, é possível identificar os contextos existentes. A instrução *csearch*, ao buscar contextos, retira da pilha a máscara de busca e o contexto a partir do qual será iniciada a busca. As instruções *le*, *ifnot* e *jmp* são utilizadas para traduzir a sentença *for* para UbiA (seção 7.1 contém uma explicação mais detalhada). *le* retira dois operandos da pilha, verifica se o primeiro é menor ou igual ao segundo, e coloca o resultado deste teste na pilha; *ifnot* retira um operando da pilha e, caso seja *false*, salta para o *label* determinado; *jmp* salta para o *label* determinado. Neste exemplo, as linhas entre 6 e 8 buscam os contextos existentes, com máscara "*programação*", a partir do contexto inicial "/Unisinos/Disciplinas". As linhas entre 21 e 28 exibem as informações dos contextos encontrados.

```
1  entity start
2    method constructor()
3      var table cinfo;
4      var int index;
5      var table clist =
6        {"/Unisinos/Disciplinas"}.
7        csearch("*programacao*");
8      io.writeln("Contextos encontrados:");
9      for(index=1;index<=clist.size();index++)
10       cinfo = clist[index];
11       io.writeln(
12         "Nome: ", cinfo["name"],
13         " Caminho: ", cinfo["path"]);
14     end
15   end
16 end
```

(a)

```
.entity start                                    1
  .method constructor                            2
    .var table cinfo                             3
    .var int index                               4
    .var table clist                             5
    ldconst "/Unisinos/Disciplinas"              6
    ldconst "*programacao*"                      7
    csearch                                      8
    ldconst "Contextos encontrados:"             9
    ldconst 1                                    10
    lcall io.writeln                             11
    ldconst 1                                    12
    stvar index                                  13
:1  ldvar index                                  14
    tabsize clist                                15
    le                                           16
    ifnot 2                                      17
    ldvar index                                  18
    ldtab clist                                  19
    stvar cinfo                                  20
    ldconst "Nome: "                             21
    ldconst "name"                               22
    ldtab cinfo                                  23
    ldconst " Caminho: "                         24
    ldconst "path"                               25
    ldtab cinfo                                  26
    ldconst 4                                    27
    lcall io.writeln                             28
    jmp 1                                        29
:2  ret                                          30
  .end                                           31
.end                                             32
```

(b)

Figura 6.29: Busca por contextos

A figura 6.30 contém um exemplo onde, a partir de um caminho, é possível

identificar os serviços existentes. A instrução *ssearch*, ao buscar serviços, retira da pilha o nome do serviço e o contexto a partir do qual será iniciada a busca. Neste exemplo, as linhas entre 6 e 8 buscam os serviços existentes, com nome "conversao_moedas", a partir do contexto inicial "/". As linhas entre 21 e 28 exibem as informações dos serviços encontrados.

```
1  entity start
2    method constructor()
3      var table sinfo;
4      var int index;
5      var table slist = {"/"}.
6        ssearch("conversao_moedas");
7      io.writeln("Servicos encontrados:");
8      for(index=1;index<=slist.size();index++)
9        sinfo = slist[index];
10       io.writeln(
11         "Nome: ", sinfo["name"],
12         " Descricao: ", sinfo["description"]);
13     end
14   end
15 end
```

(a)

```
.entity start                                          1
  .method constructor                                  2
    .var table sinfo                                   3
    .var int index                                     4
    .var table slist                                   5
    ldconst "/"                                        6
    ldconst "conversao_moedas"                         7
    ssearch                                            8
    ldconst "Servicos encontrados:"                    9
    ldconst 1                                          10
    lcall io.writeln                                   11
    ldconst 1                                          12
    stvar index                                        13
:1  ldvar index                                        14
    tabsize slist                                      15
    le                                                 16
    ifnot 2                                            17
    ldvar index                                        18
    ldtab slist                                        19
    stvar sinfo                                        20
    ldconst "Nome: "                                   21
    ldconst "name"                                     22
    ldtab sinfo                                        23
    ldconst " Descricao: "                             24
    ldconst "description"                              25
    ldtab sinfo                                        26
    ldconst 4                                          27
    lcall io.writeln                                   28
    jmp 1                                              29
:2  ret                                                30
  .end                                                 31
.end                                                   32
```

(b)

Figura 6.30: Busca por serviços

6.4.12 Adaptação através dos eventos

A figura 6.31 contém um exemplo de uso dos eventos pré-definidos. *start* associa um método ao evento *on_ content_ changed* do contexto *time.now*. Caso ocorra este evento, a nova hora é informada. A instrução *bevent* associa um método a determinado evento. Neste exemplo, na linha 6, o método *update* é associado ao evento *on_ content_ changed*, existente no contexto *time.now*. O método *update*, quando executado, recebe dois argumentos: *old_ value*, que contém o valor antigo

da informação; e *new_ value*, que contém o novo valor.

```
1  entity start
2    method constructor()
3      time.now.on_context_changed += update;
4    end
5    method update(string old_value,
6                  string new_value)
7      io.writeln("Hora atual: ", new_value);
8    end
9  end
```

(a)

```
.entity start                                          1
  .method constructor:nil                              2
    ldconst "update:string:string"                     3
    ldconst "on_content_changed"                       4
    ldconst "time.now"                                 5
    bevent                                             6
  .end                                                 7
  .method update:string:string                         8
    .param string old_value                            9
    .param string new_value                            10
    ldconst "Hora atual: "                             11
    ldparam new_value                                  12
    ldconst 2                                          13
    lcall io.writeln                                   14
:1  ret                                                15
  .end                                                 16
.end                                                   17
```

(b)

Figura 6.31: Uso dos eventos pré-definidos

A figura 6.32 contém um exemplo que define um novo evento. A entidade *reservatorio* define o evento *alteracao_ capacidade* (linha 2), evento este que é executado sempre que a propriedade *capacidade* for alterada. *start* cria uma instância de *reservatorio*, e define que o método *alteracao* deve ser executado quando o evento *alteracao_ capacidade* ocorrer. A instrução *revent* executa os métodos associados a determinado evento (caso existam). Neste exemplo em UbiA, a instrução revent (linha 8) executa os métodos associados ao evento *alteracao_ capacidade*, quando necessário; a instrução *bevent* (linha 19) associa o método *alteracao:nil* ao evento *alteracao_ capacidade*.

6.4.13 Adaptação durante a resolução de nomes de entidades

A figura 6.33 contém um exemplo onde a entidade *test* que será instanciada em *start*, na linha 35, depende do contexto *activity.what*. Uma das definições da entidade *test* se adapta a todos os contextos. A cláusula *when* de uma entidade define os contextos em que a entidade se adapta; caso não exista, a entidade se adapta a todos os contextos. Neste exemplo, caso *activity.what* seja "teaching", a entidade definida na linha 1 será utilizada; caso seja "learning", a entidade definida na linha 13 será utilizada; caso contrário, a entidade que se adapta a todos os contextos, definida na linha 25, será utilizada.

```
1  entity reservatorio
2    prop event alteracao_capacidade;
3    prop int capacidade
4      set
5        capacidade = value;
6        raise alteracao_capacidade();
7      end
8    end
9  end
10
11 entity start
12   method constructor()
13     var element agua = reservatorio.new();
14     agua.alteracao_capacidade += alteracao;
15     agua.litros = 1000;
16   end
17   method alteracao()
18     io.writeln("Capacidade alterada");
19   end
20 end
```

(a)

```
.entity reservatorio                                    1
  .prop event alteracao_capacidade                      2
  .prop int capacidade                                  3
  .method __set_capacidade:int                          4
    .param int value                                    5
    ldparam value                                       6
    stprop capacidade                                   7
    revent alteracao_capacidade                         8
  .end                                                  9
.end                                                    10
                                                        11
.entity start                                           12
  .method constructor:nil                               13
    .var element agua                                   14
    newelem reservatorio                                15
    stvar agua                                          16
    ldvar agua                                          17
    ldconst "alteracao_capacidade"                      18
    bevent alteracao:nil                                19
    ldconst 1000                                        20
    ldvar agua                                          21
    mcall __set_capacidade:int                          22
  .end                                                  23
  .method alteracao:nil                                 24
    ldconst "Capacidade alterada"                       25
    ldconst 1                                           26
    lcall io.writeln                                    27
  .end                                                  28
.end                                                    29
```

(b)

Figura 6.32: Definição e uso de novos eventos

6.4.14 Adaptação durante a resolução de nomes de sub-rotinas

A figura 6.34 contém um exemplo onde o método *run* que será executado em *start*, na linha 7, depende do contexto *activity.what*. Neste exemplo, caso *activity.what* seja "teaching", o método definido na linha 11 será utilizado; caso seja "learning", o método definido na linha 21 será utilizado; caso contrário, não existe um método que se adapta ao contexto atual, então uma mensagem de erro é exibida pela UbiVM, e a aplicação é finalizada.

Da mesma forma como ocorre com as entidades, quando não existe a cláusula *.when*, significa que o método se adapta a todos os contextos.

6.4.15 Mobilidade de código

A figura 6.35 contém um exemplo de mobilidade de código. Neste exemplo um aplicativo move-se entre duas UbiVMs distintas. A instrução *move* instrui a UbiVM a suspender temporariamente o aplicativo em execução, serializar seus dados e movê-los para o destino especificado. Neste exemplo, a instrução *ldcontexti*

```
1  entity test
2    when (activity.what == "teaching")
3    method run()
4      io.writeln( "Estou ensinando..." );
5    end
6  end
7
8  entity test
9    when (activity.what == "learning")
10   method run()
11     io.writeln( "Estou aprendendo..." );
12   end
13 end
14
15 entity test
16   method run()
17     io.writeln( "Nao sei..." );
18   end
19 end
20
21 entity start
22   method constructor()
23     var element etest = test.new();
24     etest.run();
25   end
26 end
```

(a)

```
1  .entity test
2    .when
3      ldcontexti "activity.what"
4      ldconst "teaching"
5      eq
6    .end
7    .method run:nil
8      ldconst "Estou ensinando..."
9      ldconst 1
10     lcall io.writeln
11   .end
12 .end
13 .entity test
14   .when
15     ldcontexti "activity.what"
16     ldconst "learning"
17     eq
18   .end
19   .method run:nil
20     ldconst "Estou aprendendo..."
21     ldconst 1
22     lcall io.writeln
23   .end
24 .end
25 .entity test
26   .method run:nil
27     ldconst "Nao sei.."
28     ldconst 1
29     lcall io.writeln
30   .end
31 .end
32 .entity start
33   .method constructor:nil
34     .var element etest
35     newelem test
36     stvar etest
37     ldvar etest
38     mcall run:nil
39   .end
40 .end
```

(b)

Figura 6.33: Adaptação durante a resolução de nomes de entidades

(linha 3) carrega na pilha o nome do dispositivo onde a UbiVM está sendo executada. A instrução *ldconst* (linha 7) carrega na pilha o dispositivo destino para o qual o aplicativo em execução será movido. A instrução *move* (linha 8) move o aplicativo para o destino especificado. Após isso, a UbiVM continua a execução do aplicativo na UbiVM destino, do ponto onde parou. A linha 12 exibe o nome do novo dispositivo onde o aplicativo está sendo executado. Salienta-se que as instruções posteriores a instrução *move* são executadas na UbiVM destino.

Antes da movimentação, o aplicativo é parado, seu estado interno serializado, e então movido para a nova UbiVM no dispositivo destino. Após a mobilidade, o

```
entity start
  method constructor()
    var element etest = test.new();
    etest.run();
  end
end

entity test
  method run()
    when (activity.what == "teaching")
    io.writeln("Estou ensinando...");
  end
  method run()
    when (activity.what == "learning")
    io.writeln("Estou aprendendo...");
  end
end
```

(a)

```
.entity start
  .method constructor:nil
    .var element etest
    newelem test
    stvar etest
    ldvar etest
    mcall run:nil
  .end
.end
.entity test
  .method run:nil
    .when
      ldcontexti "activity.what"
      ldconst "teaching"
      eq
    .end
    ldconst "Estou ensinando..."
    ldconst 1
    lcall io.writeln
  .end
  .method run:nil
    .when
      ldcontexti "activity.what"
      ldconst "learning"
      eq
    .end
    ldconst "Estou aprendendo..."
    ldconst 1
    lcall io.writeln
  .end
.end
```

(b)

Figura 6.34: Adaptação durante a resolução de nomes de métodos

```
entity start
  method constructor()
    io.writeln("Estou em ", identity.host);
    move("MeuDesktop");
    io.writeln("Estou em ", identity.host);
  end
end
```

(a)

```
.entity start
  .method constructor:nil
    ldconst "Estou em "
    ldcontexti "identity.host"
    ldconst 2
    lcall io.writeln
    ldconst "MeuDesktop"
    move
    ldconst "Estou em "
    ldcontexti "identity.host"
    ldconst 2
    lcall io.writeln
  .end
.end
```

(b)

Figura 6.35: Mobilidade de código

aplicativo continua sua execução do ponto em que parou, e os seus elementos permanecem nos mesmos contextos em que se encontravam anteriormente, compartilhando as mesmas informações contextuais.

6.4.16 Concorrência

A figura 6.36 contém um exemplo de métodos concorrentes. Neste exemplo, *start* executa o método *calc* duas vezes, concorrentemente; após isso, *start* aguarda o término destes métodos para exibir o seu resultado. A instrução *mccall* executa um método concorrente enquanto que a instrução *mcwait* aguarda o término de um método concorrente. Neste exemplo, *start* executa duas vezes o método *calc* concorrentemente (linhas 6 e 8); após isso, *start* aguarda o término da execução destes fluxos concorrentes (linhas 11 e 13) e exibe o resultado de cada execução (linha 15).

```
1  entity start
2    method constructor()
3      var thread m1 = calc(10) [concurrent];
4      var thread m2 = calc(20) [concurrent];
5      io.writeln("Calculos: ",
6          m1.wait(),
7          m2.wait());
8    end
9    method calc(int n) [int]
10     return n*10;
11   end
12 end
```

(a)

```
.entity start                                  1
  .method constructor:nil                      2
    .var thread m1                             3
    .var thread m2                             4
    ldconst 10                                 5
    mccall calc:int                            6
    ldconst 20                                 7
    mccall calc:int                            8
    ldconst "Calculos: "                       9
    ldvar m1                                   10
    cmwait                                     11
    ldvar m2                                   12
    cmwait                                     13
    ldconst 3                                  14
    lcall io.writeln                           15
  .end                                         16
  .method calc:int                             17
    .param int n                               18
    .result int                                19
    ldparam n                                  20
    ldconst 10                                 21
    mul                                        22
    stresult 0                                 23
  .end                                         24
.end                                           25
```

(b)

Figura 6.36: Métodos concorrentes

6.5 Considerações sobre o capítulo

Este capítulo apresentou a UbiVM, a máquina virtual que suporta a execução dos aplicativos desenvolvidos em UbiL. A UbiVM provê suporte à serviços, contextos, sensibilidade ao contexto, adaptação ao contexto, mobilidade forte de código e concorrência.

Este capítulo não documenta uma implementação específica da UbiVM.

Uma implementação da UbiVM apenas deve ler o arquivo que contém o *bytecode* (arquivo UVM) e executar corretamente as instruções existentes. Detalhes de implementação não fazem parte desta especificação, porque desnecessariamente cercearíam a criatividade dos desenvolvedores. Por exemplo, o *layout* de memória das áreas de dados durante a execução e qualquer otimização interna na implementação das instruções da UbiVM, não são abordadas nesta especificação, sendo totalmente deixadas à cargo do desenvolvedor.

O próximo capítulo apresenta os protótipos do UbiC e da UbiVM, as ferramentas que possibilitaram o desenvolvimento e execução dos aplicativos escritos em UbiL.

Capítulo 7

Implementação

Este capítulo aborda os protótipos do UbiC e da UbiVM. O UbiC e a UbiVM possibilitaram o desenvolvimento e a execução dos aplicativos ubíquos desenvolvidos em UbiL.

Para auxiliar na prototipagem do UbiC e da UbiVM, a LibUVM foi desenvolvida. A LibUVM é uma biblioteca de código que auxilia a leitura e a geração do arquivo UVM. Esta biblioteca é utilizada tanto pelo UbiC (para gerar o arquivo UVM) como pela UbiVM (para carregar o arquivo UVM).

O diagrama de classes da LibUVM (figura 7.1) contém as principais classes existentes nesta biblioteca. *CAssemblyDefinition* é a classe principal; a partir dela é possível definir várias entidades (classe *CEntityDefinition*) e obter o *bytecode* propriamente dito. *CEntityDefinition* contém a definição de uma entidade e possibilita a definição das propriedades, métodos, serviços e eventos de uma entidade. *CMethodDefinition* contém a definição de um método e possibilita a identificação dos parâmetros, valores de retorno, variáveis locais e instruções de um método.

7.1 Protótipo do UbiC

O UbiCompiler (abreviadamente, UbiC) é o compilador responsável por ler um arquivo contendo sentenças em UbiL e traduzí-las para *bytecode* (figura 7.2). O *bytecode* é uma representação binária da UbiA, vista no capítulo 6. Durante esta

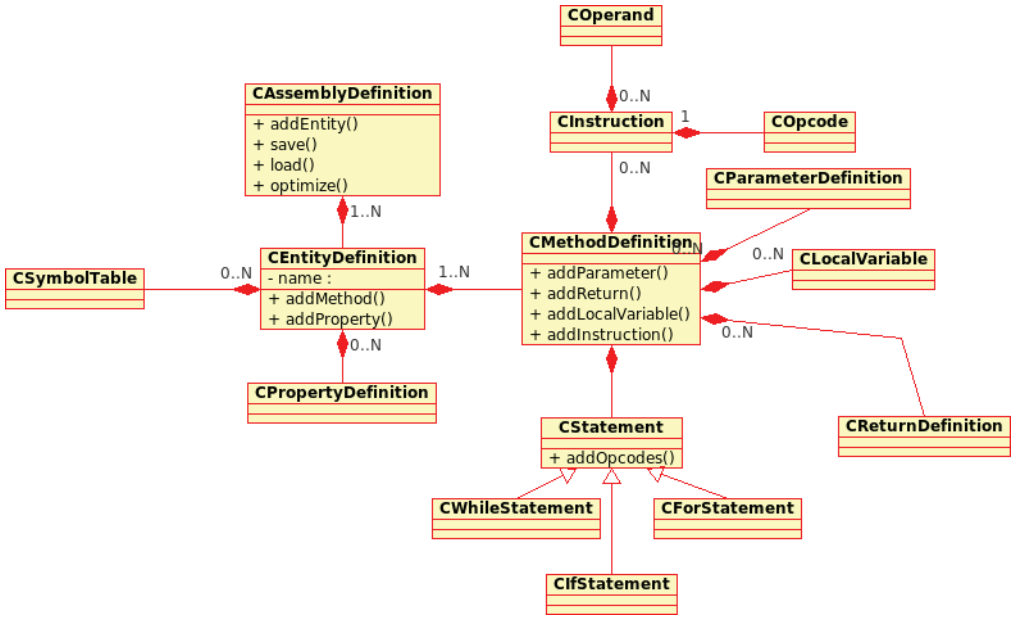


Figura 7.1: Diagrama de classes da LibUVM

conversão o UbiC gera o *opcode* correto de cada instrução, resolve endereçamentos e realiza a precedência de operadores. A partir deste *bytecode* o UbiC gera o arquivo UVM para posterior execução pela UbiVM.

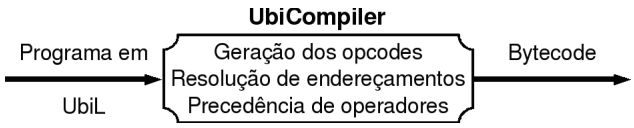


Figura 7.2: Protótipo do UbiC

A figura 7.3 apresenta o contexto de desenvolvimento do UbiC. Os analisadores léxico, sintático e semântico [Aho, Sethi e Ullman 1988, Price e Toscani 2001] foram gerados com o auxílio do gerador de parsers AntLR [AntLR. Site oficial. 2009]. A entrada do AntLR é a gramática da UbiL acrescida de ações semânticas, ações estas que invocam funções da LibUVM para auxiliar na geração do *bytecode*. A gramática da UbiL contém 904 linhas contendo 23998 *bytes*. Quando aplicado ao AntLR, esta gramática gera sete arquivos contendo o analisador léxico, sintático e semântico do UbiC. Juntos, estes arquivos possuem 22302 linhas de código, totalizando 715450 *bytes*. O apêndice A contém a gramática utilizada.

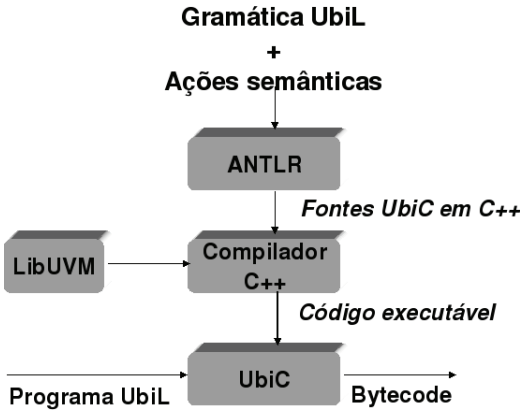


Figura 7.3: Contexto de desenvolvimento do UbiC

O diagrama de classes do UbiC (figura 7.4) contém as principais classes existentes neste protótipo. A classe *CUbiC* é responsável por coordenar a execução do compilador, e utiliza *CSyntaxAnalyzer* para coordenar a execução do *parser*. *CAssemblyDefinition* é a classe da LibUVM responsável por gerar o arquivo UVM com o *bytecode*. A partir de *CAssemblyDefinition* é possível definir as entidades (*CEntityDefinition*), juntamente com suas propriedades (*CPropertyDefinition*) e métodos (*CMethodDefinition*).

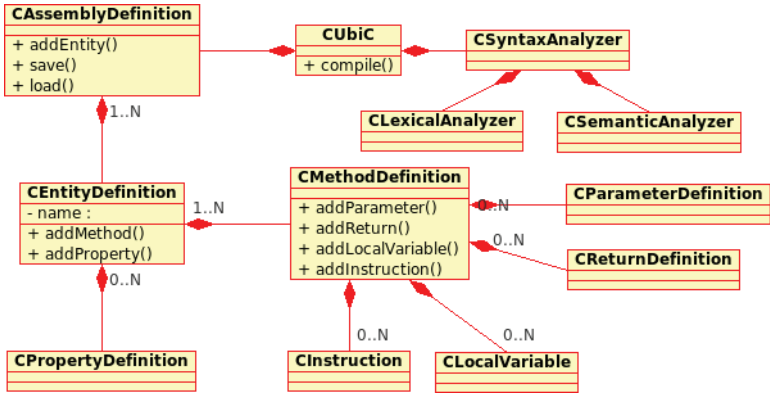


Figura 7.4: Diagrama de classes do UbiC

A figura 7.5 contém um exemplo em que o UbiC resolve a precedência de operadores utilizando a pilha de operandos. Neste exemplo, as linhas entre 4 e 6 carregam as constantes “12”, “5” e “10” na pilha. Após isso, a operação de maior precedência é realizada (multiplicação, linha 7), antes da operação de menor

precedência (subtração, linha 8).

```
1 entity start
2     method constructor()
3         io.writeln(12-5*10);
4     end
5 end
6
```

(a)

```
.entity start                          1
    .method constructor                2
        // io.writeln(12-5*10);        3
        ldconst 12                     4
        ldconst 5                      5
        ldconst 10                     6
        mul                            7
        sub                            8
        ldconst 1                      9
        lcall io.writeln               10
    .end                               11
.end                                   12
```

(b)

Figura 7.5: Resolução da precedência dos operadores

A figura 7.6 contém um exemplo de tradução da sentença "*if-then-else*" para UbiA. Para tal, são utilizadas instruções relacionais e instruções de salto. A instrução relacional *ge* retira os dois últimos operandos da pilha, verifica se o primeiro é maior ou igual ao segundo, e coloca o resultado deste teste na pilha. A instrução de salto condicional *ifnot* retira o último operando da pilha e, caso seja *false*, salta para o *label* determinado. A instrução de salto incondicional *jmp* salta para o *label* determinado. Neste exemplo, as linhas 5 e 6 inicializam a variável x com a constante "1"; as linhas entre 8 e 10 verificam se x é maior ou igual a *1*, inserindo o resultado na pilha; se o resultado for *false*, a instrução da linha 11 direciona o fluxo de execução para o *label* 9 (*else* da sentença); caso contrário o código das linhas 13 e 14 é executado (*then* da sentença). A instrução *jmp* da linha 16 direciona o fluxo de execução para a próxima sentença após o *if* que, neste exemplo, é o fim do método.

A figura 7.7 contém um exemplo de tradução para UbiA da sentença "*for(inicialização; teste; passo)*". Para tal, são utilizadas instruções aritméticas, instruções relacionais e instruções de salto. A instrução relacional *le* retira os dois últimos operandos da pilha, verifica se o primeiro é menor ou igual ao segundo, e coloca o resultado deste teste na pilha. Neste exemplo, as linhas 6 e 7 realizam a inicialização do *for* ("i=1"); o bloco que inicia no *label* 2 realiza o teste condicional ("i<=10"), direcionando o fluxo para a próxima sentença após o *for*, caso necessário; a linha 12 direciona o fluxo para o bloco de execução do *for*; o bloco que inicia no *label* 7 realiza o passo da sentença ("i=i+1") e direciona o fluxo para novamente realizar o teste condicional; o bloco que inicia no *label* 12

```
entity start
    method constructor()
        var int x=1;
        if (x>=1)
            x=10;
        else
            x=20;
        end
    end
end

```

(a)

```
.entity start
    .method constructor
        // var int x=1;
        .var 0 int x
        ldconst 1
        stvar x
        // if (x>=1)
        ldvar x
        ldconst 1
        ge
        ifnot 9
        // x=10;
        ldconst 10
        stvar x
        // end
        jmp 11
        // x=20;
    :9  ldconst 20
        stvar x
    :11 exit
    .end
.end
```

(b)

Figura 7.6: Tradução da sentença *if-then-else*

executa o bloco de comandos existentes no *for*, e direciona o fluxo para novamente executar o passo da sentença.

```
entity start
    method constructor()
        var int i;
        for(i=1; i<=10; i=i+1)
            io.writeln(i);
        end
    end
end

```

(a)

```
.entity start
    .method constructor
        // var int i;
        .var 0 int i
        // for(i=1; i<=10; i=i+1)
        ldconst 1
        stvar i
    :2  ldvar i
        ldconst 10
        le
        ifnot 16
        jmp 12
    :7  ldvar i
        ldconst 1
        add
        stvar i
        jmp 2
        // io.writeln(i);
    :12 ldvar i
        ldconst 1
        lcall io.writeln
        // end
        jmp 7
    :16 exit
    .end
.end
```

(b)

Figura 7.7: Tradução da sentença *for*

7.2 Protótipo da UbiVM

Para avaliar o ambiente de desenvolvimento foi criado um protótipo da UbiVM com os principais recursos que suportam a execução de aplicativos ubíquos. O diagrama de classes da figura 7.8 contém as principais classes existentes neste protótipo. A classe *CUbiVM* coordena a execução da MV. Ela utiliza *CAssemblyDefinition*, existente na LibUVM, para carregar e verificar o *bytecode* existente no arquivo UVM. Por sua vez, *CBytecodeExecutor* executa as instruções propriamente ditas enquanto que *CSensor* instancia os sensores necessários para a correta execução da aplicação.

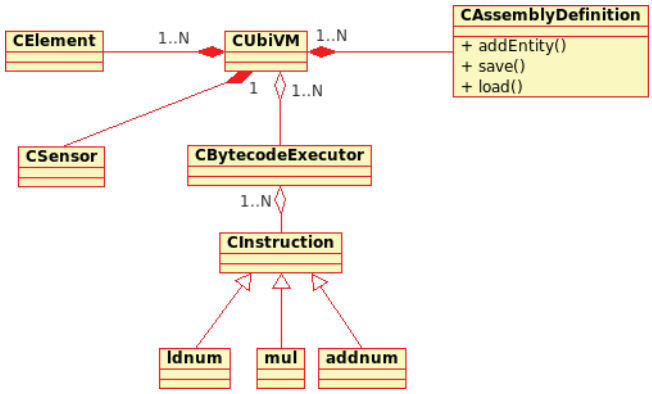


Figura 7.8: Diagrama de classes da UbiVM

Os sensores obtém dados pertinentes do ambiente, armazenando-os nos contextos estáticos. Foi desenvolvida uma API em C/C++ para possibilitar que os sensores acessem e modifiquem os dados existentes nestes contextos.

Existem dois tipos de sensores: **sensores lógicos**, que obtém dados simulados, e **sensores físicos**, que obtém dados reais. Para obter-se a localização simbólica e física, necessárias nas aplicações implementadas no capítulo 8, dois sensores lógicos foram implementados. São eles:

- Sensor de localização simbólica: obtém a localização simbólica dos usuários, conforme o passo da simulação, do arquivo de configuração "slocation.data";
- Sensor de localização física: obtém a localização física dos usuários , conforme o passo da simulação, do arquivo de configuração "plocation.data".

Durante a execução, o passo da simulação é um valor numérico, iniciando em 1, e incrementado a cada 15 segundos. A cada mudança no valor do passo, a nova localização (simbólica ou física) é obtida do arquivo de configuração do sensor e atualizada no contexto *location*.

A figura 7.9 contém um exemplo com o formato do arquivo para o sensor de localização simbólica. Para cada passo de simulação, é definida a localização simbólica de cada usuário. Se em determinado passo não houver uma localização para determinado usuário, a antiga localização é mantida. Neste exemplo, o usuário "alex" troca sua localização a cada passo, enquanto que o usuário "jorge" mantém sua localização em "mobilab" durante toda a simulação.

```
# step identity slocation
  1    alex
  2    alex     laboratorio geral
  3    alex     mobilab
  4    alex     mini biblioteca
  1    jorge    mobilab
```

Figura 7.9: Arquivo de configuração do sensor de localização simbólica

A figura 7.10 contém um exemplo com o formato do arquivo para o sensor de localização física. Para cada passo de simulação, é definida a latitude, longitude e altitude de cada usuário. Assim como ocorre no sensor de localização simbólica, se em determinado passo, não houver uma localização para determinado usuário, a antiga localização é mantida.

```
# step identity latitude longitude altitude
  1    alex     1        2         0
  2    alex     4        5         0
  3    alex     10       18        0
  4    alex     14       8         0
```

Figura 7.10: Arquivo de configuração do sensor de localização física

Ao executar a UbiVM, é necessário definir qual sensor será executado através do parâmetro "-p". Por exemplo, "ubivm -pslocation app1" executa a aplicação "app1" com o sensor de localização simbólica. Da mesma forma, "ubivm -pplocation app1" executa a aplicação "app1" com o sensor de localização física.

Para possibilitar que as informações contextuais fiquem distribuídas entre várias UbiVMs, uma MV não tem ciência das outras MVs em execução no ambiente. No protótipo atual, quando uma UbiVM necessita de informações

sobre determinado contexto, o *Communication Provider* envia uma mensagem de *broadcast* no canal de comunicação, e aguarda as respostas das UbiVMs contendo suas informações sobre o contexto. Esta abordagem foi adotada por ser de fácil implementação e por tolerar falhas como indisponibilidade momentânea de recursos. Possíveis melhorias serão abordadas em trabalhos futuros.

7.3 Considerações sobre o capítulo

Este capítulo apresentou os protótipos do UbiC e da UbiVM. A figura 7.11 contém a tela obtida após o UbiC compilar o exemplo *hello_ world*, enquanto que a figura 7.12 contém a tela obtida após a UbiVM executar este exemplo.

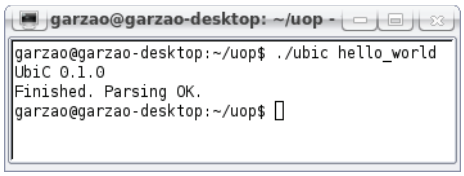


Figura 7.11: Screenshot após UbiC compilar *hello_ world*

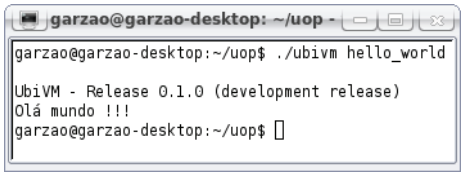


Figura 7.12: Screenshot após UbiVM executar *hello_ world*

As seguintes observações devem ser destacadas:

- a geração de código correta, avaliando a precedência de operadores, é facilitada em máquinas virtuais baseadas em pilha;
- o UbiC não está realizando nenhum tipo de otimização no código UbiA gerado;
- sentenças de alto nível da UbiL como *if*, *while* e *for* são decompostas em instruções relacionais, de salto e condicionais na UbiVM;

- o uso de sensores facilita a sensibilidade ao contexto;
- o uso de mensagens de *broadcast* na comunicação entre as UbiVMs auxilia na tarefa de manter os contextos de forma distribuída.

Salienta-se que foi priorizado o desenvolvimento dos recursos pertinentes a computação ubíqua nestes protótipos. Além disso, a meta foi desenvolver protótipos de fácil manutenção, e um melhor desempenho durante a execução é meta para trabalhos futuros.

Os protótipos foram desenvolvidos e testados no ambiente Linux com o compilador GCC versão 4.2.4. O código fonte foi desenvolvido utilizando-se a Linguagem C++, sob a licença GPL. A documentação utiliza a licença FDL.

O próximo capítulo apresenta os experimentos realizados usando os protótipos.

Capítulo 8

Experimentos

Este capítulo apresenta os experimentos realizados em UbiL utilizando os protótipos do UbiC e da UbiVM.

Apesar da computação ubíqua possibilitar oportunidades em diversas áreas, foram selecionadas apenas três destas para a realização de experimentos: educação [Barbosa et al. 2008, Barbosa et al. 2006], comércio [Franco et al. 2009] e redes sociais [Costa 2009]. Para cada experimento foi definido um cenário de teste contendo sua descrição, possíveis situações envolvendo os participantes, e uma discussão com os resultados obtidos. Como o objetivo é comprovar que o modelo do UOP facilitou o desenvolvimento dos cenários, a discussão é focada nos recursos específicos da computação ubíqua, ressaltando os trechos de código relevantes em UbiL. O apêndice B contém o código completo dos experimentos.

Os equipamentos utilizados nos experimentos foram:

- *Desktop* 1: Pentium IV 2.4, *dual core*, 2 GB memória, 320 GB disco, placa de rede 10/100;
- *Desktop* 2: Pentium IV 1.4, 512 MB memória, 160 GB disco, placa de rede 10/100;
- Roteador 10/100 com 4 portas.

Para a realização dos experimentos, os dois *desktops* foram interligados através do roteador. As localizações físicas e simbólicas foram obtidas através

dos sensores lógicos (seção 7.2). O conteúdo dos arquivos de configuração destes sensores são exibidos, quando pertinente.

Na implementação atual, uma UbiVM somente é capaz de executar uma aplicação. Por este motivo, conforme o cenário, cada *desktop* executou uma ou mais instâncias da UbiVM simultaneamente, onde cada instância executava uma das aplicações implementadas.

As seções 8.1, 8.2 e 8.3 apresentam, respectivamente, os experimentos com educação ubíqua, comércio ubíquo e redes sociais; a seção 8.4 finaliza este capítulo apresentando as considerações finais.

8.1 Educação Ubíqua

A disseminação da computação ubíqua ocasionará um impacto significativo na área da Educação (*Ubiquitous Learning* [Yau et al. 2003, Rogers et al. 2005, Barbosa et al. 2006]). No cenário da educação apoiada pela computação ubíqua, novos pressupostos educacionais devem ser pensados, uma vez que os recursos pedagógicos podem ser acessados a qualquer momento e em qualquer lugar. O suporte ubíquo permite a construção de programas de aprendizagem relacionados com questões dinâmicas do contexto do aprendiz. O ambiente controla as aplicações orientadas à educação, possibilitando que o contexto seja vinculado com os objetivos pedagógicos. A educação neste cenário é dinâmica e os recursos educacionais estão distribuídos em contextos. Baseado nos objetivos do aprendiz, o sistema pode gerar intervenções do tipo: "um material/pessoa/dispositivo que se relaciona com seu objetivo está disponível para você agora (contexto)".

O cenário proposto é o "Relacionamento por interesses similares entre aprendizes", e foi baseado no cenário proposto em [Barbosa et al. 2008, Barbosa et al. 2006]. Cada aprendiz informa, para a aplicação, seus interesses e os objetos de aprendizagem que gostaria de compartilhar. A partir deste momento, conforme o aprendiz move-se de um ambiente para outro, as oportunidades de aprendizado são apresentadas. As oportunidades são identificadas quando aprendizes com interesses similares estão na mesma localização simbólica.

8.1.1 Cenário

O aprendiz Alex está a caminho do PIPCA para acertar os detalhes finais da sua dissertação com o professor Jorge. Alex, ao entrar no PIPCA, informa ao seu aplicativo "Oportunidades Pedagógicas" o interesse sobre Compiladores, em especial sobre otimização de código. Também aproveita e divulga o objeto de aprendizagem sobre compiladores de que dispõe. Jorge também é um aprendiz na área de Compiladores, e já havia informado ao seu aplicativo "Oportunidades Pedagógicas" que tem interesse nesta área, em especial sobre análise semântica, e que tem alguns objetos de aprendizagem sobre este tema. O aprendiz Gabriel já registrou seu interesse em Compiladores, em especial sobre geração de código, e seus objetos de aprendizagem. A aprendiz Graciele também já registrou seu interesse na área de Sistemas Operacionais, em especial sobre gerência de memória virtual. Ao entrar no PIPCA, Alex dirige-se ao Laboratório Geral onde, entre alguns colegas de aula, está Graciele. A UbiVM, em execução no PDA do Alex, tenta identificar possíveis oportunidades pedagógicas nesta localização, mas não encontra, visto que Graciele não tem interesses similares a Alex. Após isso, Alex dirige-se ao Mobilab para conversar com o professor Jorge. Ao entrar nesta localização, a UbiVM em execução no seu PDA identifica que os usuários Jorge e Gabriel tem interesse em Compiladores, e lista os objetos de aprendizagem destes usuários. Alex ficou surpreso pois não sabia do interesse de Gabriel por compiladores. Alex aproveita a oportunidade e solicita uma cópia dos objetos de aprendizagem que são de seu interesse. Após acertar os detalhes sobre a dissertação, Alex sai do PIPCA.

A figura 8.1 apresenta o ambiente que foi simulado, enquanto que a tabela 8.1 apresenta os detalhes desta execução, com uma evolução temporal das ações de cada participante. Neste cenário, o contexto PIPCA compartilha as informações de localização simbólica, interesses e objetos de aprendizagem de cada aprendiz. Localização simbólica, interesse geral e interesse específico utilizam um registro contendo uma chave composta por "LEARNER" e o nome do usuário, enquanto que as informações sobre os objetos de aprendizagem utilizam um registro contendo uma chave composta por "OBJECT", nome do usuário e o número do objeto. Quatro localizações simbólicas foram utilizadas: Mobilab, Mini biblioteca,

Laboratório geral e Área Convivência. Conforme os aprendizes se deslocam pelo ambiente, sua nova localização simbólica é obtida através do sensor e publicada no contexto, possibilitando identificar os aprendizes com interesses similares, na mesma localização simbólica. Como a localização simbólica do aprendiz é relevante, este cenário utiliza o sensor de localização simbólica configurado conforme a tabela 8.2.

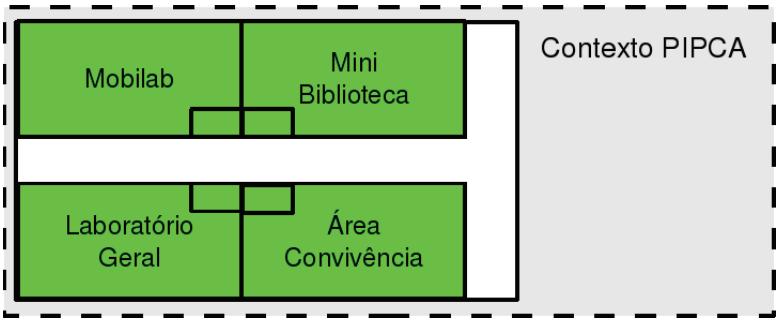


Figura 8.1: Ambiente simulado do PIPCA

Para a simulação deste ambiente, o *desktop 1* foi utilizado como sendo o PDA do aprendiz Alex, enquanto que o *desktop 2* foi utilizado como sendo o PDA dos aprendizes Gabriel, Graciele e Jorge. Neste cenário foi utilizado o aplicativo Oportunidades pedagógicas que contém 123 linhas de código fonte, 4780 caracteres e resulta em um arquivo UVM com 3476 bytes. Este aplicativo foi executado no PDA do Alex, Jorge, Gabriel e Graciele com a linha de comando "ubivm -pslocation oportunidades_pedagogicas".

A figura 8.2 contém a interface utilizada pelos aprendizes. Neste momento, Alex estava no Mobilab e a sua interface mostrava os objetos de aprendizagem dos aprendizes Jorge e Gabriel.

8.1.2 Discussão

O compartilhamento de conteúdos nos contextos é realizado através do método *cpublish*. Nos tempos 1, 2, 3 e 4 os interesses dos aprendizes e as suas localizações são publicadas. O trecho abaixo contém o código que publica um conteúdo, com uma chave composta pela constante "LEARNER" e o nome do aprendiz (*identity.name*), contendo como resultado o interesse geral (*identity.general_interest*), específico (*identity.specific_interest*) e a localização

Tabela 8.1: Relacionamento entre os aprendizes no PIPCA

Tempo	**Personagem**	**Ações**
1	Jorge	Executa a aplicação "Oportunidades pedagógicas" no seu PDA. Cadastra o interesse geral "Compiladores", interesse específico "Análise semântica", e seus dois objetos de aprendizagem: - Artigo "Advanced compiler optimizations for supercomputers", ACM, 1986. - Artigo "Dependence graphs and compiler optimizations", ACM, 1981.
	PDA do Jorge	UbiVM publica os interesses de Jorge, sua localização e as informações sobre os seus objetos no contexto PIPCA.
2	Gabriel	Executa a aplicação "Oportunidades pedagógicas" no seu PDA. Cadastra o interesse geral "Compiladores", interesse específico "Geração de código", e seu objeto de aprendizagem: - Artigo "Code generation in the Columbia Esterel Compiler", EURASIP, 2007.
	PDA do Gabriel	UbiVM publica os interesses de Gabriel, sua localização e as informações sobre o seu objeto no contexto PIPCA.
3	Graciele	Executa a aplicação "Oportunidades pedagógicas" no seu PDA. Cadastra o interesse geral "Sistemas operacionais" e interesse específico "Gerência de memória virtual".
	PDA da Graciele	UbiVM publica os interesses de Graciele e sua localização no contexto PIPCA.
4	Alex	Executa a aplicação "Oportunidades pedagógicas" no seu PDA. Cadastra o interesse geral "Compiladores", interesse específico "Otimização de código", e seu objeto de aprendizagem: - Livro "Compilers: Principles, Techniques, and Tools", Addison Wesley, 1988.
	PDA do Alex	UbiVM publica os interesses de Alex, sua localização e as informações sobre o seu objeto no contexto PIPCA.
5	Alex	Dirige-se ao Laboratório Geral.
	PDA do Alex	UbiVM identifica a localização simbólica "Laboratório geral", publica sua nova localização e envia solicitação para listar os conteúdos do contexto PIPCA.
6	PDA da Graciele	UbiVM recebe a solicitação e responde com os interesses e a localização de Graciele.
	PDA do Jorge	UbiVM recebe a solicitação e responde com os interesses e a localização de Jorge.
	PDA do Gabriel	UbiVM recebe a solicitação e responde com os interesses e a localização de Gabriel.
7	PDA do Alex	UbiVM filtra por interesses similares, na mesma localização, e não identifica oportunidades pedagógicas.
8	Alex	Dirige-se ao MobiLab.
	PDA do Alex	UbiVM identifica a localização simbólica "Mobilab", publica sua nova localização e envia solicitação para listar quem está no contexto.
9	PDA do Jorge	UbiVM recebe a solicitação e responde com os interesses e a localização de Jorge.
	PDA do Gabriel	UbiVM recebe a solicitação e responde com os interesses e a localização de Gabriel.
	PDA da Graciele	UbiVM recebe a solicitação e responde com os interesses e a localização de Graciele.
10	PDA do Alex	UbiVM filtra por interesses similares, na mesma localização, e identifica que Jorge e Gabriel também tem interesse em Compiladores. Solicita os objetos de aprendizados publicados por eles no contexto.
11	PDA do Jorge	UbiVM recebe a solicitação vinda do PDA do Alex e responde com seus objetos de aprendizagem.
	PDA do Gabriel	UbiVM recebe a solicitação vinda do PDA do Alex e responde com seus objetos de aprendizagem.
12	PDA do Alex	UbiVM recebe e lista os objetos de aprendizagem de Jorge e Gabriel.
13	Alex	Visualiza a mensagem sobre os interesses de Jorge e Gabriel, com a lista dos seus objetos. Aproveita a oportunidade para solicitar a cópia de alguns.
14	Alex	Finaliza a execução do seu aplicativo e sai do PIPCA.

Tabela 8.2: Dados do sensor no PIPCA

Passo	Identificação	Localização simbólica
1	alex	
1	jorge	Mobilab
1	gabriel	Mobilab
1	graciele	Laboratório Geral
2	alex	Laboratório Geral
3	alex	Mobilab
4	alex	

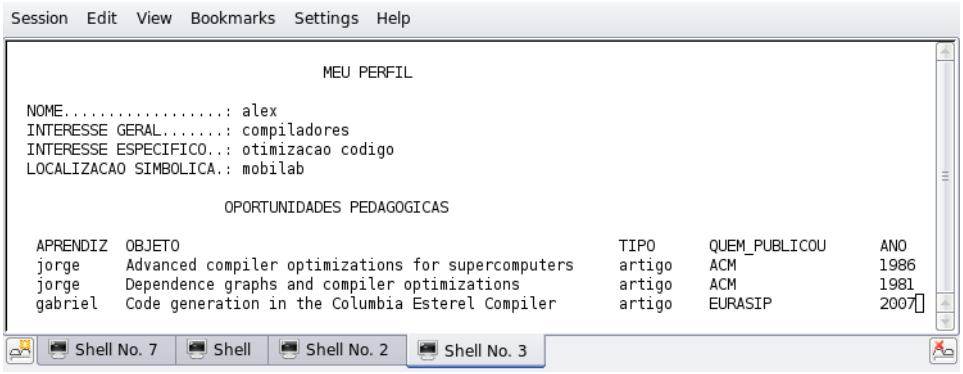


Figura 8.2: Interface dos aprendizes

do aprendiz (*location.symbolic*). *identity.name*, *identity.general_interest* e *identity.specific_interest* foram informados pelo aprendiz enquanto que *location.symbolic* foi obtido através do sensor. Tanto a chave como o resultado dos conteúdos existentes nos contextos podem ser compostos por N informações, o que facilita o compartilhamento de conteúdos entre as aplicações.

```
{"/PIPCA"}.cpublish("LEARNER", identity.name =>
        identity.general_interest, identity.specific_interest, location.symbolic);
```

O método abaixo, utilizado nos tempos 1, 2 e 3, publica os objetos de aprendizagem informados pelos aprendizes. Cada objeto de aprendizagem possui uma chave composta pela constante "OBJECT", o nome do aprendiz e o número do objeto de aprendizagem. O resultado é composto pelo nome do objeto, o tipo, quem publicou e o ano.

```
method add_object(string name, string type, string event, string year)
    _obj_count++;
    {"/PIPCA"}.cpublish("OBJECT", identity.name, _obj_count => name, type, event, year);
end
```

Nos tempos 5 e 8, a nova localização do aprendiz Alex é publicada. O trecho abaixo contém o código utilizado.

```
{"/PIPCA"}.cpublish("LEARNER", identity.name =>
        identity.general_interest, identity.specific_interest, location.symbolic);
```

Nos tempos 5 e 8, o trecho abaixo identifica quem está no mesmo contexto. A sentença "{"/PIPCA"}.clist()" obtém a lista com todos os conteúdos existentes no contexto PIPCA, conteúdos estes que podem estar em aplicações distintas, inclusive sendo executadas em dispositivos distintos.

```
learner_list = {"/PIPCA"}.clist();
for(learner_number=1; learner_number<=learner_list.size(); learner_number++)
    // ...
end
```

Nos tempos 7 e 10, o trecho abaixo filtra quem está no mesmo contexto, com a mesma localização, e com interesses similares. O *for* filtra os conteúdos do contexto PIPCA com "LEARNER" na chave, com interesses similares e a mesma localização. Os objetos de aprendizagem destes aprendizes são listados com o método *show_ learner_ objects*.

```
learner_list = {"/PIPCA"}.clist();
for(learner_number=1; learner_number<=learner_list.size(); learner_number++)
    learner = learner_list[learner_number];
    if (learner.keys[1] == "LEARNER")
        if (learner.keys[2] <> _user_name and learner.results[1] == _general_interest and
            learner.results[3] == location.symbolic)
            show_learner_objects(learner.keys[2]);
        end
    end
end
```

No tempo 10 são encontrados os aprendizes Jorge e Gabriel, com interesses similares, e o método *show_ learner_ objects* (listado abaixo) é executado para cada aprendiz. O *for* filtra os conteúdos do contexto PIPCA com "OBJECT" na chave, e que sejam do aprendiz em questão. A função *mvwrite*, da biblioteca *ncurses*, exibe a mensagem na linha e coluna determinadas.

```
method show_learner_objects(string user_name)
    var tuple obj;
    var table obj_list;
    var int   obj_number;

    obj_list = {"/PIPCA"}.clist();
    for(obj_number=1; obj_number<=obj_list.size(); obj_number++)
        obj = obj_list[obj_number];
        if (obj.keys[1] == "OBJECT")
            if (obj.keys[2] == user_name)
                ncurses.mvwrite(_line, 2, user_name);
                ncurses.mvwrite(_line, 12, obj.results[1]);
                ncurses.mvwrite(_line, 67, obj.results[2]);
                ncurses.mvwrite(_line, 77, obj.results[3]);
                ncurses.mvwrite(_line, 96, obj.results[4]);
                _line++;
            end
        end
    end
end
```

8.2 Comércio ubíquo

A computação ubíqua vem gerando novas oportunidades para o comércio [Galanxhi-Janaqi e Nah 2004, Gershman 2002]. A área de pesquisa dedicada à exploração de negócios em ambientes ubíquos é denominada Comércio Ubíquo.

No cenário do comércio apoiado pela computação ubíqua, novos pressupostos comerciais devem ser pensados, uma vez que produtos e serviços podem ser ofertados em qualquer lugar e a qualquer momento. O comércio neste cenário é dinâmico e as oportunidades estão distribuídas em contextos. Por exemplo, baseado nos desejos de consumo do cliente, um sistema pode gerar intervenções do tipo: "foi encontrado um lojista neste local, que possui uma oferta para o seu desejo de consumo".

O cenário proposto é o "Relacionamento entre Cliente e Lojista no *Shopping*", e foi baseado no cenário proposto em [Franco et al. 2009]. Os lojistas publicam suas ofertas na praça de alimentação enquanto que o cliente cadastra o desejo por um determinado tipo de almoço. Após visualizar as ofertas que atendem 100% do seu desejo, o cliente seleciona uma e realiza o pedido no seu PDA, utilizando o serviço disponibilizado pelo lojista.

8.2.1 Cenário

Graciele está a caminho do *Shopping Center* para almoçar com sua família. Ela dispõe de R$ 15,00 e gostaria de comer um prato a base de massa. As ofertas do Restaurante da Olga e da Lancheria do Victório já foram publicadas. O restaurante possui cardápios diferenciados para almoço e janta enquanto que a lancheria possui um único cardápio. Quando Graciele entra na praça de alimentação do *shopping*, a UbiVM em execução no seu PDA verifica o "Repositório de Ofertas" de todas as lojas (Restaurantes e Lancherias), disponível na Praça de Alimentação, tentando identificar pratos a base de massa que estejam dentro do valor máximo pretendido. Neste instante, a UbiVM em execução no PDA da Graciele emite um alerta, permitindo que ela visualize as opções. Após verificar as ofertas, Graciele decide por um prato de "Massa a Carbonara" que custa R$ 13,00. Através do serviço de "Registro de pedido", disponibilizado pelo lojista do prato escolhido, Graciele recebe o número do seu pedido, e se encaminha ao caixa para realizar o pagamento. Enquanto isso o seu pedido já está sendo preparado e em minutos será disponibilizado.

A tabela 8.3 apresenta os detalhes desta execução, com uma evolução temporal das ações de cada participante. Neste cenário, o contexto "/shopping/praca_alimentacao" compartilha as informações sobre as ofertas dos lojistas. Cada oferta é publicada em um registro contendo uma chave composta pelo nome do lojista e o número da oferta, tendo como resultado o nome do prato, tipo (lanche ou refeição), base do prato, ingredientes, valor e o nome do serviço para realizar o pedido. Como a localização do cliente ou lojista é irrelevante, este cenário não utiliza sensores e não define localização simbólica ou física.

Para a simulação deste ambiente, o *desktop 1* foi utilizado como sendo o PDA da cliente Graciele, enquanto que o *desktop 2* foi utilizado como sendo o PDA do Restaurante e da Lancheria. Os seguintes aplicativos foram utilizados:

- Busca de ofertas: contendo 58 linhas de código fonte, 2457 caracteres, e resultando em um arquivo UVM com 1951 bytes, este aplicativo foi executado no PDA da Graciele com a linha de comando "ubivm busca_ofertas";

- Ofertas do restaurante: contendo 55 linhas de código fonte, 2111 caracteres, e

Tabela 8.3: Relacionamento entre Cliente e Lojista no *Shopping*

Tempo	**Personagem**	**Ações**
1	Restaurante	Executa a aplicação "Ofertas Restaurante". Cadastra a oferta "Massa a Carbonara" com suas características: - Lojista: Restaurante da Olga - Categoria: refeição - Componente base: massa - Ingredientes: Ovos, creme de leite, bacon, queijo parmesão, macarrão - Valor: R$ 13,00 - Serviço para realizar o pedido: *restaurante_checkout*
	PDA do Restaurante	UbiVM publica a oferta no contexto "/shopping/praca_alimentacao". Também publica o serviço *restaurante_checkout*, utilizado pelos clientes para realizarem seus pedidos.
2	Lancheria	Executa a aplicação "Ofertas Lancheria". Cadastra a oferta "Pizza presunto" com suas características: - Lojista: Lancheria do Victório - Categoria: lanche - Componente base: massa - Ingredientes: Presunto, queijo, orégano - Valor: R$ 11,00 - Serviço para realizar o pedido: *lancheria_checkout*
	PDA da Lancheria	UbiVM publica a oferta no contexto "/shopping/praca_alimentacao". Também publica o serviço *lancheria_checkout*, utilizado pelos clientes para realizarem seus pedidos.
3	Graciele	Executa a aplicação "Busca ofertas". Cadastra o desejo "Prato de massa" com as características: - Categoria: refeição - Componente base: massa - Valor: R$ 15,00
	PDA da Graciele	UbiVM verifica as ofertas de todas as lojas no contexto "/shopping/praca_alimentacao". Ao identificar que uma das ofertas atende 100% das características do desejo de Graciele, gera um aviso para Graciele com a oportunidade identificada, disponibilizando o serviço *restaurante_checkout*.
4	Graciele	Visualiza a mensagem com a oferta do prato de "Massa a Carbonara" e solicita a realização do pedido.
	PDA da Graciele	UbiVM utiliza o serviço do restaurante para realizar o pedido, e fica aguardando o recebimento do número do pedido.
5	PDA do Restaurante	UbiVM executa a solicitação de serviço, vindo do PDA da Graciele. Este serviço executa uma ordem para a fila de preparo e de faturamento do pedido. Após isso, envia o número do pedido para o PDA da Graciele.
6	PDA da Graciele	UbiVM recebe o número do pedido e exibe para Graciele. Este aviso contém o número do pedido a ser informado no caixa, tanto para pagamento como para a retirada do prato.
7	Graciele	Verifica o número do pedido e dirige-se ao caixa. Seu prato já está na fila de preparo.

resultando em um arquivo UVM com 1761 bytes, este aplicativo foi executado no PDA do restaurante com a linha de comando "ubivm ofertas_restaurante";

- Ofertas da lancheria: contendo 25 linhas de código fonte, 820 caracteres, e resultando em um arquivo UVM com 775 bytes, este aplicativo foi executado no PDA da lancheria com a linha de comando "ubivm ofertas_lancheria".

A figura 8.3 contém a interface utilizada por Graciele. Neste momento,

Graciele escolheu o seu pedido através do menu, e o número do seu pedido foi informado. A figura 8.4 contém a interface utilizado pelo restaurante. Neste momento, Graciele já tinha feito seu pedido, que foi registrado na interface.

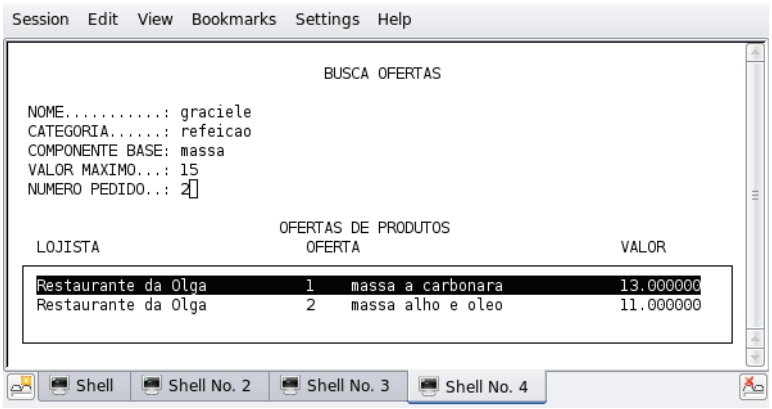


Figura 8.3: Interface para realização do pedido

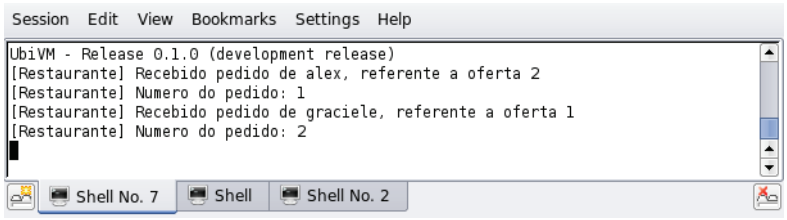


Figura 8.4: Interface do Restaurante

8.2.2 Discussão

No tempo 1, as ofertas do restaurante são publicadas. Como o restaurante possui cardápios diferenciados para almoço e janta, o horário é quem determina as ofertas que serão publicadas. O trecho abaixo contém as duas versões da entidade *restaurante_offers*, responsável por publicar as ofertas. A primeira versão é adaptada para o horário do almoço, enquanto que a segunda é adaptada para o horário da janta. As ofertas são publicadas com suas características no contexto "/shopping/praca_alimentacao". Dentre as características consta o nome do serviço que deve ser executado para realizar pedidos ao restaurante que, neste exemplo, é *restaurante_checkout*.

```
entity restaurante_offers when (time.hour >= 11 and time.hour <= 14)
    // Horario do almoco
    method constructor()
        {"/shopping/praca_alimentacao"}.cpublish(
            "Restaurante da Olga", "1" => "massa a carbonara", "refeicao", "massa",
            "ovos, creme de leite, bacon, queijo parmesao, macarrao", 13.00,
            "restaurante_checkout");
        // ...
    end
end

entity restaurante_offers when (time.hour >= 18 and time.hour <= 22)
    // Horario da janta
    method constructor()
        {"/shopping/praca_alimentacao"}.cpublish(
            "Restaurante da Olga", "1" => "sopa creme ervilha", "refeicao", "sopa",
            "creme de erviha, leite", 9.00,
            "restaurante_checkout");
        // ...
    end
end
```

Ainda no tempo 1, o serviço *restaurante_checkout* é publicado no contexto "/shopping/praca_alimentacao". Os argumentos deste serviço são o nome do cliente e o número da oferta.

```
{"/shopping/praca_alimentacao"}.spublish("restaurante_checkout");
```

No tempo 2, um trecho similar ao tempo 1 publica a oferta "pizza presunto" da lancheria. Nesta oferta, o serviço disponibilizado para realizar pedidos é *lancheria_checkout*. Os argumentos deste serviço são os mesmos do serviço disponibilizado pelo restaurante (nome do cliente e número da oferta).

```
{"/shopping/praca_alimentacao"}.cpublish(
    "Lancheria do Victorio", "1" =>
    "pizza presunto", "lanche", "massa", "presunto, queijo, oregano", 11.00,
    "lancheria_checkout");
{"/shopping/praca_alimentacao"}.spublish("restaurante_checkout");
```

No tempo 3, o trecho abaixo identifica, entre as ofertas disponíveis, quais satisfazem os requisitos definidos por Graciele (categoria, componente base e valor máximo). Um menu com as ofertas selecionadas é montado com o auxílio da biblioteca *ncurses_menu*.

```
offer_list = {"/shopping/praca_alimentacao"}.clist();
for(offer_number=1; offer_number<=offer_list.size(); offer_number++)
    offer = offer_list[offer_number];
    if (offer.results[2] == identity.categoria and
        offer.results[3] == identity.componente_base and
        offer.results[5] <= identity.valor_maximo)
        ncurses_menu.new_item(
            tools.pad(offer.keys[1], 20) + tools.pad(offer.keys[2],5) +
            tools.pad(offer.results[1], 30) + tools.ftoa(offer.results[5]));
        count++;
        offer_code_in_menu[count] = offer.keys[2];
        service_name_in_menu[count] = offer.results[6];
    end
end
```

É possível a execução de serviços dinamicamente, onde o nome do serviço somente será conhecido durante a execução. Sempre que executa-se um serviço, o nome do serviço é uma *string*. Sendo assim, ele pode ser identificado pelo usuário, obtido de uma variável, ser passado como parâmetro para o aplicativo, ou, como no exemplo abaixo, ser obtido do contexto. A execução de serviços dinâmicos auxilia a implementação dos LBS. No trecho abaixo, o nome do serviço é obtido da oferta que estava no contexto. O serviço é então executado com seus argumentos (nome do cliente e número da oferta). Este serviço retorna o número do pedido, que é exibido para o cliente.

```
offer_number = ncurses_menu.show(5, 80, 11, 1);
identity.checkout_number =
        {"/shopping/praca_alimentacao"}.srun(service_name_in_menu[offer_number],
        identity.name, offer_code_in_menu[offer_number]);
ncurses.mvwrite(7, 19, identity.checkout_number);
```

8.3 Redes sociais

Para tirar um maior proveito dos LBS, além de utilizar uma conexão a uma rede tecnológica, pode-se conectar as pessoas fisicamente presentes naquele local em uma rede social [Costa 2009]. Fazer parte de uma rede social já é processo usual em muitos serviços que usamos diariamente, tais como *Orkut*, *Facebook*, *LinkedIn*, *Twitter*, *Wikipedia*, *digg*, *YouTube* e *flickr*.

Além da construção de espaços virtuais e do estímulo a auto-expressão, as redes sociais tem alterado nosso estilo de comunicação, apesar de manter os

princípios da interação social humana [Kleinberg 2008]. Por causa disso, nada mais natural, e portanto mais próximo do conceito de computação ubíqua, do que aliar a interação ubíqua e a oferta de serviços baseados em localização com a formação de uma rede social espontânea, ou seja, uma rede social formada por pessoas presentes em locais específicos utilizando dispositivos móveis [Mani, Ngyuen e Crespi 2009].

As redes sociais são estruturas formadas por indivíduos, pessoas ou organizações, que são conectadas por algum tipo de interdependências, tais como interesses comuns, relações de amizade, etc. [Watts 2003].

O cenário proposto é "Relacionamento entre pessoas e serviços no parque". O parque disponibiliza serviços e conteúdos, além de proporcionar a interação entre as pessoas fisicamente presentes no local. O parque disponibiliza conteúdos como roteiros de caminhadas e localização dos seus serviços. Como serviços, o parque oferece um sistema para reserva de vagas (estacionamento, time de basquete e time de futebol) e um sistema de reserva de canchas e quadras. Um biólogo disponibiliza o serviço de visita assistida ao parque. A medida que a pessoa percorre o parque, este serviço disponibiliza informações descrevendo a fauna e a flora baseando-se na sua localização.

8.3.1 Cenário

A família Garzão (Alex, Graciele e Gabriel) resolve ir ao parque. Alex e Gabriel querem praticar basquete enquanto que Graciele quer utilizar o serviço de visita assistida ao parque.

Alex utiliza o serviço "Reserva de vagas no estacionamento" para verificar a disponibilidade de vagas. Como existem vagas, uma reserva é feita, e Alex visualiza a vaga disponível. Alex e Gabriel utilizam o serviço "Reserva de vagas no time de basquete". Eles são informados que existem vagas em um time, e que suas reservas foram feitas. Após isso visualizam a informação com o horário de início da partida. Graciele solicita informações sobre o serviço "Visita assistida". Graciele visualiza a descrição do serviço e os possíveis pontos de visitação no parque, e aciona o serviço. Pedro é outro usuário do parque. Chegou lá há algumas horas, e neste momento decidiu verificar se existem vagas para o próximo jogo de basquete. Ao utilizar o serviço de reservas, recebe a informação indicando que uma reserva

foi feita e o horário de início da partida. Alex, Gabriel e Pedro dirigem-se para a quadra, pois o jogo inicia em 10 minutos. A família Garzão permaneceu no parque por aproximadamente duas horas. Graciele aprendeu muito com a visita assistida. Alex e Gabriel, além de praticar basquete, também conheceram pessoas interessantes como o Pedro, com quem identificaram muitos interesses em comum. Agora, a família Garzão dirige-se para casa. Pedro permaneceu no parque para a próxima partida.

A figura 8.5 apresenta o ambiente que foi simulado. As coordenadas X e Y, existentes nesta figura, foram utilizadas para determinar as áreas de interesse para o serviço de Visita assistida. A tabela 8.4 apresenta os detalhes desta execução, com uma evolução temporal das ações de cada participante. Neste cenário, o contexto parque compartilha os serviços de "Reserva de vagas no estacionamento", "Reserva de vagas no time de basquete", "Reserva de quadras" e "Reserva de canchas". Este contexto também disponibiliza as informações com roteiros de caminhadas e localização dos serviços como banheiros, bicicletário, entre outros. Também neste contexto, o biólogo Thiago disponibiliza informações bem como o serviço de "Visita assistida". O Mini zoológico, Plantas silvestres e Animais silvestres são áreas que fazem parte deste serviço. Quando uma pessoa percorre umas das áreas próximas (Informações sobre o mini zoológico, Informações sobre as plantas silvestres e Informações sobre os animais silvestres), seu PDA disponibiliza informações relativas a fauna e flora local. Como a localização física da usuária Graciele é relevante, este cenário utiliza o sensor de localização física configurado conforme a tabela 8.5.

Para a simulação deste ambiente, o *desktop 1* foi utilizado como sendo o PDA do Alex, Graciele, Gabriel e Pedro, enquanto que o *desktop 2* foi utilizado como sendo o PDA do Parque e do Thiago. Os seguintes aplicativos foram utilizados:

- Solicitação de reserva de vagas no estacionamento: contendo 18 linhas de código fonte, 476 caracteres, e resultando em um arquivo UVM com 525 bytes, este aplicativo foi executado no PDA do Alex com a linha de comando "ubivm solicita_reserva_estacionamento";

- Solicitação de reserva de vagas no time de basquete: contendo 18 linhas de código fonte, 451 caracteres, e resultando em um arquivo UVM com 510

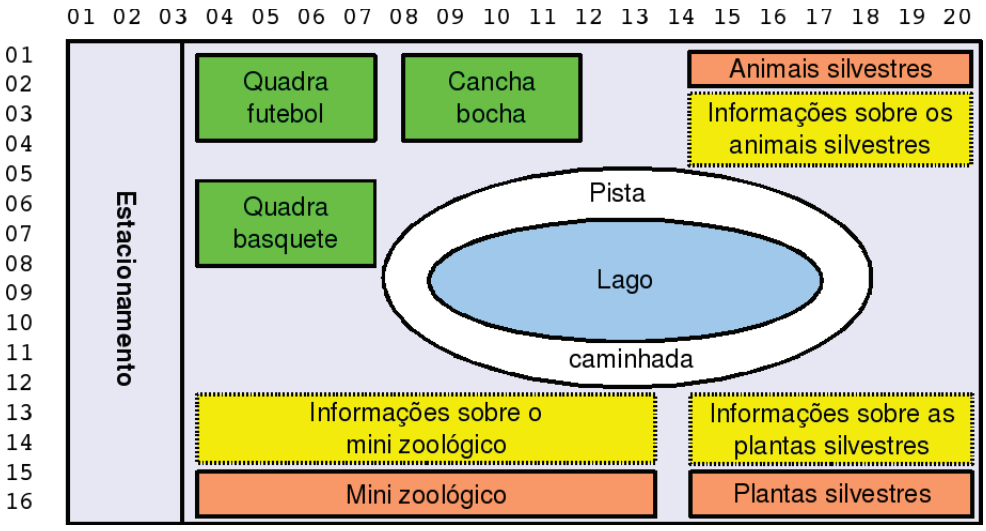


Figura 8.5: Ambiente simulado do parque

bytes, este aplicativo foi executado no PDA do Alex, Gabriel e Pedro, com a linha de comando "ubivm solicita_reserva_time_basquete";

- Informações e ativação do serviço de visita assistida: contendo 36 linhas de código fonte, 1438 caracteres, e resultando em um arquivo UVM com 1077 bytes, este aplicativo foi executado no PDA da Graciele com a linha de comando "ubivm -pplocation solicita_info_visita_assistida";

- Serviços e informações sobre o parque: contendo 78 linhas de código fonte, 2960 caracteres, e resultando em um arquivo UVM com 2358 bytes, este aplicativo foi executado no PDA do Parque com a linha de comando "ubivm parque";

- Informações e o serviço de visita assistida: contendo 63 linhas de código fonte, 2327 caracteres, e resultando em um arquivo UVM com 1721 bytes, este aplicativo foi executado no PDA do Thiago com a linha de comando "ubivm visita_assistida".

As figuras 8.6(a) e 8.6(b) contém, respectivamente, a interface utiliza por Alex para solicitar uma reserva no estacionamento e a interface utiliza por Gabriel para solicitar uma reserva no time de basquete.

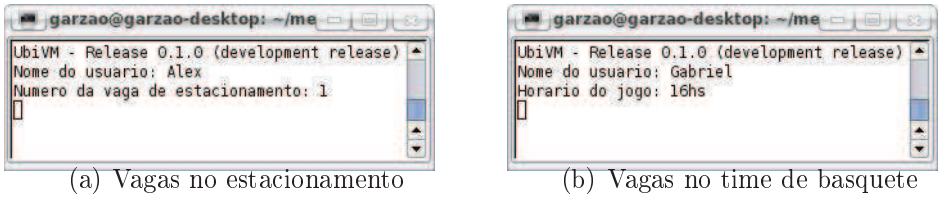
Tabela 8.4: Relacionamento entre pessoas e serviços na visita ao parque

Tempo	**Personagem**	**Ações**
1	Parque	Executa a aplicação "Parque".
	PDA do Parque	UbiVM disponibiliza no contexto *parque* os conteúdos "Roteiros de caminhada" e "Localização de serviços", e os seguintes serviços: - Reserva de vagas (estacionamento, time de basquete e time de futebol) - Reserva de quadras e canchas
2	Thiago	Executa a aplicação "Visita Assistida no Parque".
	PDA do Thiago	UbiVM disponibiliza o serviço "Visita assistida" e informações sobre ele no contexto *parque*.
3	Alex	Executa a aplicação que reserva uma vaga no estacionamento.
	PDA do Alex	UbiVM utiliza o serviço "Reserva de vagas no estacionamento", existente no contexto *parque*.
4	PDA do Parque	UbiVM recebe a solicitação do serviço, vinda do PDA do Alex. Envia o número da vaga disponível.
5	PDA do Alex	UbiVM recebe o número da vaga, exibindo um aviso para Alex.
6	Alex	Visualiza e estaciona o carro na vaga informada.
7	Alex, Gabriel e Pedro	Executam a aplicação que reserva uma vaga no time de basquete.
	PDA do Alex, Gabriel e Pedro	UbiVM utiliza o serviço "Reserva de vagas no time de basquete", existente no contexto *parque*.
8	PDA do Parque	UbiVM recebe a solicitação do serviço, vindo dos PDAs de Alex, Gabriel e Pedro. Como existem vagas no próximo time, faz as reservas e envia a informação com o horário de início do jogo.
9	PDA do Alex, Gabriel e Pedro	UbiVM recebe as informações sobre a reserva, e gera um aviso para Alex, Gabriel e Pedro.
10	Alex, Gabriel e Pedro	Visualizam as informações sobre o horário, dirigindo-se para a quadra.
11	Graciele	Executa a aplicação que exibe informações sobre o serviço "Visita assistida", ativando-o.
	PDA da Graciele	UbiVM solicita informações sobre o serviço "Visita assistida", existente no contexto *parque*.
12	PDA do Thiago	UbiVM recebe a solicitação vinda do PDA da Graciele, e envia as informações sobre o serviço.
13	PDA da Graciele	UbiVM recebe as informações sobre o serviço, e gera um aviso para Graciele.
14	Graciele	Visualiza as informações e ativa o serviço. Graciele move-se pelo parque em busca dos pontos interessantes para serem visitados.
	PDA da Graciele	Sempre que ocorre uma mudança na localização física, a UbiVM envia a nova localização para o serviço "Visita assistida".
16	PDA do Thiago	UbiVM recebe a localização e verifica que não existem informações específicas nesta localização. Informa isso ao PDA da Graciele.
17	PDA da Graciele	UbiVM recebe a resposta de que não existem informações específicas nesta localização.
18	Graciele	Move-se novamente tentando encontrar os pontos interessantes.
	PDA da Graciele	UbiVM envia a nova localização para o serviço "Visita assistida".
19	PDA do Thiago	UbiVM recebe a localização e verifica que existem informações específicas nesta localização. Informa isso ao PDA da Graciele.
20	PDA da Graciele	UbiVM recebe as informações específicas desta localização, e gera aviso para Graciele.
21	Graciele	Visualiza as informações.
22	Graciele	Continua buscando pontos de interesse.
23	Alex, Graciele e Gabriel	Fim do passeio.

A figura 8.7 contém a interface do aplicativo que disponibiliza os serviços do parque. Neste momento, Alex e Gabriel já haviam solicitados suas reservas, ambas

Tabela 8.5: Dados do sensor no parque

Passo	Identificação	Latitude	Longitude	Altitude
1	graciele	12	1	0
2	graciele	12	2	0
3	graciele	12	3	0
4	graciele	13	4	0
5	graciele	11	13	0
6	graciele	13	15	0
7	graciele	13	18	0
8	graciele	9	18	0
9	graciele	3	18	0
10	graciele	3	17	0
11	graciele	3	13	0



(a) Vagas no estacionamento (b) Vagas no time de basquete

Figura 8.6: Interfaces para reserva de vagas

registradas nesta interface. Pedro ainda não fez sua reserva.

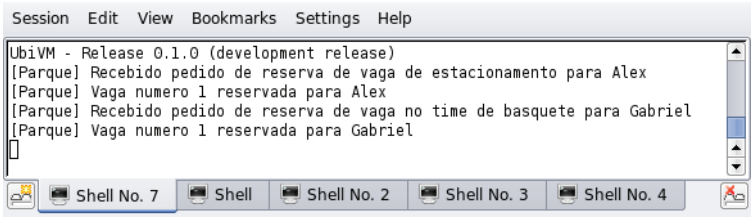


Figura 8.7: Interface do parque

A figura 8.8 contém a interface utilizada por Graciele para obter informações bem como executar o serviço de visita assistida. A cada nova posição, o aplicativo de Graciele envia suas coordenadas para o serviço, que registra na tela as solicitações recebidas (figura 8.9).

8.3.2 Discussão

No tempo 1, o trecho abaixo publica as informações e serviços do parque, no contexto *parque*. Os serviços abaixo recebem, como argumento, o nome do usuário que deseja efetuar a reserva.

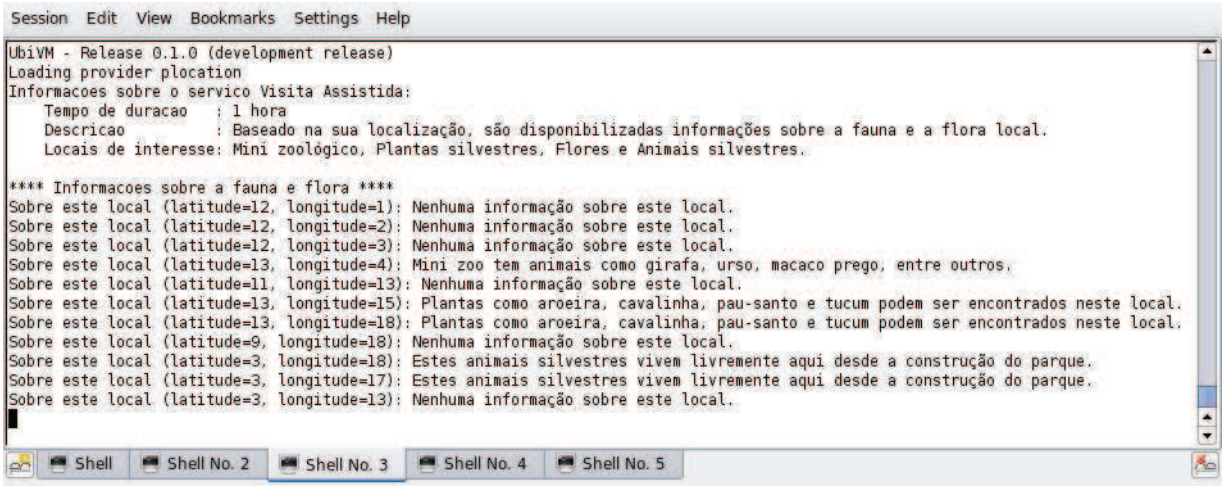


Figura 8.8: Interface com a execução do serviço de visita assistida

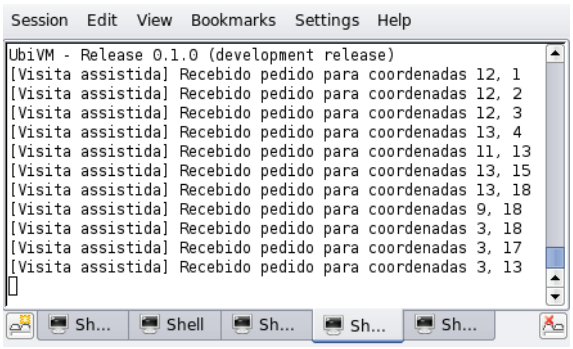


Figura 8.9: Interface do serviço de visita assistida

```
{"parque"}.cpublish("Roteiros de caminhadas" =>
    "Iniciando pelo Mini zoológico, percorra ...");
{"parque"}.cpublish("Localizacao de servicos" =>
    "Banheiros: na entrada do parque. Bicicletário: ao lado do estacionamento.");
{"parque"}.spublish("reserva_vagas_estacionamento");
{"parque"}.spublish("reserva_vagas_time_basquete");
{"parque"}.spublish("reserva_quadra_futebol");
{"parque"}.spublish("reserva_cancha_bocha");
```

No tempo 2, o trecho abaixo publica as informações e o serviço “visita assistida”, no contexto *parque*. Os conteúdos desta informação são o tempo estimado de duração, a descrição do serviço e os pontos de interesse. Os argumentos deste serviço são latitude e longitude.

```
{"parque"}.spublish("visita_assistida");
{"parque"}.cpublish(
    "Visita assistida" => "1 hora",
    "Baseado na sua localização, são disponibilizadas informações sobre a fauna e ...",
    "Mini zoológico, Plantas silvestres, Flores e Animais silvestres.");
```

No tempo 3, o trecho abaixo solicita o nome do usuário e executa o serviço de reserva de vagas no estacionamento. Após isso, exibe o número da vaga, caso exista.

```
io.writeln("Nome do usuario: "); username = io.readln();

vacancy_number = {"parque"}.srun("reserva_vagas_estacionamento", username);

if (vacancy_number == 0)
    io.writeln("Estacionamento lotado...");
else
    io.writeln("Numero da vaga de estacionamento: ", vacancy_number);
end
```

No tempo 4, o serviço abaixo verifica se existe uma vaga de estacionamento, e retorna o número desta vaga. A propriedade *_last_park_vacancy* registra o número de vagas utilizadas, de um total de 100.

```
service reserva_vagas_estacionamento(string person_name) [int]
    io.writeln("[Parque] Recebido pedido de reserva de vaga de estacionamento para ",
        person_name);
    if (_last_park_vacancy < 100)
        _last_park_vacancy++;
        io.writeln("[Parque] Vaga numero ", _last_park_vacancy,
            " reservada para ", person_name);
        return _last_park_vacancy;
    else
        io.writeln("[Parque] Estacionamento lotado ", person_name);
        return 0;
    end
end
```

No tempo 7, o trecho abaixo executa o serviço de reserva de vagas no time de basquete e exibe o horário do jogo, caso ainda existam vagas.

```
io.writeln("Nome do usuario: "); username = io.readln();

play_time = {"parque"}.srun("reserva_vagas_time_basquete", username);

if (play_time == "")
    io.writeln("Nao existem mais vagas no time...");
else
    io.writeln("Horario do jogo: ", play_time);
end
```

No tempo 8, o serviço abaixo verifica se existem vagas no time, e retorna o horário do jogo. A propriedade *_last_team_vacancy* registra o número de vagas utilizadas, de um total de 10. A propriedade *_soccer_field_play_time* armazena o horário do jogo, informado durante a reserva da quadra.

```
service reserva_vagas_time_basquete(string person_name) [string]
    io.writeln("[Parque] Recebido pedido de reserva de vaga no time de basquete para ",
        person_name);
    if (_last_team_vacancy < 10)
        _last_team_vacancy++;
        io.writeln("[Parque] Vaga numero ", _last_team_vacancy,
            " reservada para ", person_name);
        return _soccer_field_play_time;
    else
        io.writeln("[Parque] Time de basquete lotado ", person_name);
        return "";
    end
end
```

No tempo 11, o trecho abaixo exibe as informações sobre o serviço "Visita assistida", publicadas no tempo 2.

```
var tuple visita_assistida;
visita_assistida = {"parque"}.cfind("Visita assistida");
io.writeln("Informacoes sobre o servico Visita Assistida:");
io.writeln("    Tempo de duracao   : ", visita_assistida.results[1]);
io.writeln("    Descricao          : ", visita_assistida.results[2]);
io.writeln("    Locais de interesse: ", visita_assistida.results[3]);
```

Ainda no tempo 11, o trecho abaixo utiliza este serviço para exibir informações baseando-se na localização atual do usuário.

```
last_latitude  = location.latitude;
last_longitude = location.longitude;
io.writeln("Sobre este local (latitude=", last_latitude,
    ", longitude=", last_longitude, "): ",
    {"parque"}.srun("visita_assistida", last_latitude, last_longitude));
```

No tempo 12, o trecho abaixo define uma área de interesse e sua descrição. Como podem existir várias áreas de interesse, listas armazenam os dados destas áreas. A propriedade *_desc_count* registra o número de áreas de interesse armazenadas. As listas *_y1_desc*, *_x1_desc*, *_y2_desc* e *_x2_desc* armazenam as coordenadas (y1,x1,y2,x2) das áreas de interesse, enquanto que a lista *_desc* armazena a descrição da área. Neste trecho, a área de interesse, representada pela região (13,04,14,23), contém a descrição "Mini zoo tem animais como girafa, urso, macaco prego, entre outros.".

```
_desc_count++;
_y1_desc[_desc_count] = 13;
_x1_desc[_desc_count] = 04;
_y2_desc[_desc_count] = 14;
_x2_desc[_desc_count] = 12;
_desc   [_desc_count] =
    "Mini zoo tem animais como girafa, urso, macaco prego, entre outros.";
```

Caso a localização tenha alguma informação, ela é retornada pelo serviço *visita_assistida*. O trecho de código abaixo, pertencente a este serviço, varre todas as áreas armazenadas e, caso a localização faça parte de uma área de interesse, sua descrição é retornada. Neste exemplo, x e y contém a localização informada pelo usuário.

```
for(desc_number = 1; desc_number <= _desc_count; desc_number++)
    if (y >= _y1_desc[desc_number] and y <= _y2_desc[desc_number] and
        x >= _x1_desc[desc_number] and x <= _x2_desc[desc_number])
        return _desc[desc_number];
    end
end
```

8.4 Considerações sobre o capítulo

Este capítulo apresentou os experimentos realizados para avaliar a proposta do UOP.

As construções da UbiL, em conjunto com os recursos presentes na UbiVM, facilitaram o desenvolvimento de aplicativos ubíquos nos seguintes pontos:

- a publicação e a execução de serviços;

- a descoberta de serviços durante a execução;
- o compartilhamento de conteúdos entre os aplicativos;
- definição e uso dos LBS;
- a adaptação ao contexto;
- a sensibilidade ao contexto.

Os recursos de adaptação por eventos e concorrência, apesar de funcionais nos protótipos, não foram explorados nestes cenários. Os recursos de contextos hierárquicos e mobilidade de código, existentes na UbiL, não foram implementados nos protótipos, e conseqüentemente, não puderam ser explorados nestes cenários.

O próximo capítulo contém as considerações finais desta dissertação.

Capítulo 9

Considerações Finais

Este capítulo apresenta uma reflexão sobre as principais contribuições proporcionadas pelo UOP. Completando tal discussão, são exibidas as conclusões e os trabalhos futuros identificados.

9.1 Principais contribuições

A principal contribuição deste trabalho é a criação do UOP, um modelo que facilita o desenvolvimento de aplicativos ubíquos. A concretização deste modelo através da definição da UbiL, do UbiC e da UbiVM possibilitou a realização de experimentos.

Foram identificados os principais problemas e limitações existentes na área da dissertação, em função dos estudos realizados, definindo-se assim o problema de pesquisa (seção 1.2). Com base nisso, o UOP foi proposto, descrevendo-se seus objetivos (seção 1.3).

Os estudos apresentados, aliados com os resultados alcançados, permitem destacar as seguintes contribuições desta dissertação:

- desenvolvimento de estudos sobre o estado da arte no contexto da computação ubíqua;
- definição do UOP, um modelo de programação que facilita o desenvolvimento de aplicativos ubíquos;

- definição da UbiL, uma linguagem de programação que implementa os conceitos propostos no UOP;
- modelagem e implementação do UbiC, uma ferramenta para tradução de UbiL para *bytecode* de forma que os aplicativos possam ser executados na UbiVM;
- modelagem e implementação da UbiVM, uma máquina virtual que atende os requisitos propostos na UbiL, facilitando assim a exploração do modelo proposto pelo UOP;
- realização de experimentos que exploraram como a UbiL facilita o desenvolvimento de aplicativos ubíquos;
- disseminação da cultura do software livre, uma vez que todo trabalho realizado é desenvolvido considerando questões ligadas ao desenvolvimento de software livre.

Na tabela 9.1 são apresentadas novamente as características dos ambientes relacionados na seção 3.2, agora com o UOP. São elas:

- **Serviços:** indica que a solução suporta a publicação, busca e execução de serviços;
- **Contextos:** identifica que a solução suporta a definição e uso de contextos;
- **Sensibilidade ao contexto:** indica que a solução obtém informações relevantes através de sensores (hardware ou software) que permitem a adaptação do comportamento dos aplicativos ao seu contexto;
- **Adaptação ao contexto:** indica que a solução modifica o seu comportamento baseada nas informações de contexto percebidas, adaptando-se para a situação atual;
- **Mobilidade de código:** indica que a solução suporta a mobilidade de código;

- **Concorrência:** indica que a solução suporta a exploração de fluxos de execução concorrentes;
- **Suporta um modelo:** indica que a solução envolve um novo modelo de programação;
- **Linguagem de programação:** indica que a solução possui uma linguagem de programação própria;
- **Máquina virtual:** indica que a solução possui uma máquina virtual própria para a execução dos aplicativos.

Tabela 9.1: Características dos ambientes de computação ubíqua, incluindo o UOP

	Context Toolkit	**UbiHolo**	**ISAM**	**Continuum**	**One.World**	**Gaia**	**Aura**	**UOP**
Serviços				X				X
Contextos	X	X	X	X	X	X		X
Sensibilidade ao contexto	X			X	X	X	X	X
Adaptação ao contexto	X			X	X	X	X	X
Mobilidade de código		X	X	X	X			X
Concorrência		X	X	X	X	X	X	X
Suporta um paradigma		X	X					X
Linguagem de programação		X	X					X
Máquina virtual		X						X

O UOP possibilita o compartilhamento dinâmico de conteúdos e serviços através dos contextos, descoberta de contextos e serviços, permite a implementação dos LBS, suporta contextos públicos e privados, provê sensibilidade ao contexto através do uso de sensores, permite adaptação ao contexto por eventos e durante a resolução de nomes, possibilita mobilidade forte de código e a exploração da concorrência nos aplicativos.

9.2 Conclusões

Após a realização deste trabalho chegou-se à algumas conclusões. Dentre elas, pode-se citar:

- o UOP e suas ferramentas trouxeram novas abstrações que facilitam o desenvolvimento de aplicativos ubíquos;
- a criação do UOP proporciona um maior poder semântico para o desenvolvimento de aplicações ubíquas;
- a UbiA mostrou-se suficiente para facilitar a compreensão do funcionamento interno da UbiVM;
- o arquivo UVM é peça fundamental na independência de hardware e sistema operacional da UbiVM;
- o ambiente da UbiVM, apesar de ter sido construído para a UbiL, pode ser utilizado por outras linguagens; para tal, um programa escrito em uma linguagem qualquer deve ser traduzido para UbiL ou UbiA, podendo inclusive utilizar a LibUVM para facilitar esta conversão;
- o UbiC e a UbiVM facilitam a exploração e a utilização dos recursos existentes na UbiL;
- apesar da UbiL ser uma linguagem estaticamente tipada, o UOP suporta a implementação de linguagens dinamicamente tipadas e simbólicas;
- a abordagem de implementar uma máquina virtual própria proporciona o controle total desta máquina, facilitando assim a implementação e exploração de novos recursos;
- os objetivos deste trabalho, definidos na seção 1.3, foram alcançados.

9.3 Trabalhos futuros

Dentre os trabalhos futuros identificados, pode-se citar:

- Ambiente de programação: a implementação de um editor, de um depurador e de uma ferramenta de modelagem facilitaria o desenvolvimento de aplicativos escritos em UbiL;

- Simulador: a implementação de um simulador para a UbiVM facilitaria a visualização de suas estruturas internas durante a execução, facilitando o seu entendimento;
- Tratamento de exceções: apesar da importância deste tópico [Dillenburg e Barbosa 2009], na especificação atual, um fluxo em execução não possui suporte para o tratamento das exceções geradas pelas instruções da UbiVM. Qualquer exceção finaliza a máquina virtual;
- Contextos privados com memória persistente: as aplicações poderiam utilizar trilhas [Silva et al. 2009] para determinar mais precisamente o perfil do usuário;
- Segurança: isso garantiria a execução apenas de ações autorizadas;
- Modelo de comunicação eficiente: o uso de mensagens de *broadcast* na comunicação entre as UbiVMs auxiliou na tarefa de manter os contextos de forma distribuída, mas esta abordagem não é escalonável porque o consumo dos recursos de rede aumenta proporcionalmente ao número de UbiVMs;
- Contextos compartilhados: as informações são compartilhadas apenas com os membros do contexto, e a entrada de novos membros deve ser autorizada por um dos membros atuais.

Durante o desenvolvimento deste trabalho, apesar da importância, algumas características da UbiL não foram concretizadas nos protótipos, e serão abordadas em trabalhos futuros. São elas:

- contextos hierárquicos;
- mobilidade forte de código;
- sobrecarga de métodos e serviços;
- polimorfismo em métodos e serviços;
- clonagem de elementos.

Identificou-se também algumas possíveis melhorias. São elas:

- Melhorias na definição da UbiL: alguns conceitos existentes em outras linguagens facilitariam o desenvolvimento de aplicativos em UbiL. Do C++/C# cita-se a sobrecarga de operadores, programação genérica (*templates*), definição de estruturas e definição de enumeradores. Do Java cita-se reflexão (*reflection*). Além disso cita-se também a possibilidade de gerar bibliotecas de código escritas em UbiL;
- Melhoria no UbiC: dentre elas cita-se o suporte a tabelas com mais de uma dimensão, melhorias na reportagem de erros, melhorias na análise semântica, otimização do *bytecode* gerado e vários programas compilados gerarem um arquivo UVM único;
- Melhorias na UbiVM: dentre elas pode-se citar a implementação de um *Garbage Collector* tanto para o aplicativo bem como para as informações compartilhadas nos contextos públicos, carregar dinamicamente novos arquivos UVM durante a execução, melhorar o desempenho e melhorar a estabilidade.

Bibliografia

[Adiga 2007] ADIGA, H. S. Writing endian-independent code in c. IBM, 2007. Disponível em: <http://www.ibm.com/in/stats5/index.html>.

[Aho, Sethi e Ullman 1988] AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: principles, techniques, and tools*. [S.l.]: Addison Wesley, 1988.

[AntLR. Site oficial. 2009] ANTLR. Site oficial. 2009. Disponível em: <http://www.antlr.org>.

[AspectJ 2009] ASPECTJ. 2009. Disponível em: <http://eclipse.org/aspectj/>.

[Augustin 2004] AUGUSTIN, I. Isam, joing context-awareness and mobility to building pervasive applications. *Mobile Computing Handbook*, p. 73–94, 2004.

[Austin Abbie Barbir e Garg 2002] AUSTIN ABBIE BARBIR, C. F. D.; GARG, S. *Web Services Architecture Requirements*. 2002. Disponível em: <http://www.w3.org/TR/wsa-reqs/>.

[Aït-kaci 1991] AïT-KACI, H. *Warren's Abstract Machine - A Tutorial Reconstruction*. [S.l.]: MIT Press, 1991.

[Barbosa et al. 2006] BARBOSA, D. N. F. et al. Learning in a large-scale pervasive environment. In: *PERCOMW '06: Proceedings of the 4th annual IEEE international conference on Pervasive Computing and Communications Workshops*. Washington, DC, USA: IEEE Computer Society, 2006. p. 226. ISBN 0-7695-2520-2.

[Barbosa 2005] BARBOSA, J. Gholo: A multiparadigm model oriented to development of grid systems. *Future Generation Computer Systems*, v. 21, n. 1, p. 227–237, 2005.

[Barbosa et al. 2006] BARBOSA, J. et al. Local: Um modelo para suporte à aprendizagem consciente de contexto. *Simpósio Brasileiro de Informática na Educação (SBIE)*, 2006.

[Barbosa et al. 2008] BARBOSA, J. et al. Local: a model geared towards ubiquitous learning. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 40, n. 1, p. 432–436, 2008. ISSN 0097-8418.

[Barbosa 2002] BARBOSA, J. e. a. Holoparadigm: a multiparadigm model oriented to development of distributed systems. *International Conference on Parallel and Distributed Systems*, IEEE Press, p. 165–170, 2002.

[Barbosa e Geyer 2001] BARBOSA, J. L. V.; GEYER, C. F. R. Uma linguagem multiparadigma orientada ao desenvolvimento de software distribuído. *Simpósio Brasileiro de Linguagens de Programação (SBLP 2001)*, Curitiba, 2001.

[BARBOSA et al. 2007] BARBOSA, J. L. V. et al. Evaluation of a large scale ubiquitous system model through peer-to-peer protocol simulation. *IEEE International Symposium on Distributed Simulation and Real Time Applications*, n. 11, p. 175–181, 2007.

[Booth et al. 2004] BOOTH, D. et al. *Web Services Architecture*. 2004. Disponível em: <http://www.w3.org/TR/ws-arch/>.

[CLI: Common Language Infrastructure] CLI: Common Language Infrastructure. Disponível em: <http://www.ecma-international.org/publications /standards/Ecma-335.htm>.

[Costa 2008] COSTA, C. A. da. *Continuum:A Context-aware Service-based Software Infrastructure for Ubiquitous Computing*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, 2008.

[Costa 2009] COSTA, C. A. da. Mingle: Gerenciamento de interação ubíqua em uma rede social espontânea. *Projeto de pesquisa em andamento na Universidade do Vale do Rio dos Sinos (UNISINOS)*, 2009.

[Costa 2009] COSTA, C. A. da. Software infrastructure for ubiquitous computing: A context-aware service-based approach. *VDM Verlag*, n. 1, p. 174, 2009.

[Costa, Yamin e Geyer 2008] COSTA, C. A. da; YAMIN, A. C.; GEYER, C. F. Toward a general software infrastructure for ubiquitous computing. *Pervasive Computing, IEEE*, v. 7, n. 1, p. 64–73, 2008.

[Dey 2000] DEY, A. K. *Providing Architectural Support for Building Context-Aware Applications*. Tese (Doutorado) — Georgia Institute of Technology, Atlanta, USA, 2000.

[Dey 2001] DEY, A. K. Understanding and using context. *Personal Ubiquitous Computing*, Springer-Verlag, London, UK, v. 5, n. 1, p. 4–7, 2001. Disponível em: <http://citeseer.ist.psu.edu/dey01understanding.html>.

[Dillenburg e Barbosa 2009] DILLENBURG, F.; BARBOSA, J. L. V. Context-oriented exception handling. *International Journal of High Performance Systems Architecture*, n. 2, p. 16–25, 2009.

[Erl 2005] ERL, T. *Service-Oriented Architecture: Concepts, Technology, and Design*. [S.l.]: Prentice Hall PTR, 2005.

[Flanagan 2000] FLANAGAN, D. *Java: O Guia Essencial*. [S.l.]: Campus, 2000. 390 p.

[Franco et al. 2009] FRANCO, L. K. et al. Um modelo para exploração de oportunidades no comércio ubíquo. *XXXV Conferência Latino Americana de Informática (CLEI)*, p. 1–10, 2009.

[Fuggetta, Picco e Vigna 1998] FUGGETTA, A.; PICCO, G. P.; VIGNA, G. Understanding code mobility. *Software Engineering, IEEE*

Transactions on, v. 24, n. 5, p. 342–361, 1998. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=685258>.

[Galanxhi-Janaqi e Nah 2004] GALANXHI-JANAQI, H.; NAH, F. F.-H. U-commerce: emerging trends and research issues. *Industrial Management and Data Systems*, v. 104, n. 9, p. 744–755, 2004.

[Garlan et al. 2002] GARLAN, D. et al. Project aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, v. 1, n. 2, p. 22–31, 2002. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1012334>.

[Garzão e Barbosa 2002] GARZãO, A. S.; BARBOSA, J. L. V. HoloVM: Uma máquina virtual com suporte à concorrência, mobilidade e blackboards. *WSCAD*, Outubro 2002.

[Garzão e Barbosa 2003] GARZãO, A. S.; BARBOSA, J. L. V. Uma máquina virtual com suporte à concorrência, mobilidade e blackboards. In: UNIVERSIDAD MAYOR DE SAN ANDRéS. *XXIX Conferência Latinoamericana de Informática (CLEI)*. La Paz, 2003. v. 24.

[Gershman 2002] GERSHMAN, A. Ubiquitous commerce - always on, always aware, always pro-active. In: *SAINT*. [S.l.: s.n.], 2002. p. 37–38.

[Ghezi e Vigna 1997] GHEZI, C.; VIGNA, G. Mobile code paradigms and technologies: A case study. *International Workshop on Mobile Agents*, n. 1, p. 39–49, 1997.

[Ghezzi e Jazayeri 1998] GHEZZI, C.; JAZAYERI, M. *Programming Language Concepts*. [S.l.]: John Wiley & Sons, 1998.

[Grimm 2004] GRIMM, R. One.world: experiences with a pervasive computing architecture. *Pervasive Computing, IEEE*, v. 3, n. 3, p. 22–30, 2004. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1321024>.

[Harrison e Ossher 1993] HARRISON, W.; OSSHER, H. Subject-oriented programming: a critique of pure objects. In: *OOPSLA '93: Proceedings of the*

eighth annual conference on Object-oriented programming systems, languages, and applications. [S.l.]: ACM Press, 1993. v. 28, n. 10, p. 411–428. ISSN 0362-1340.

[Henricksen e Indulska 2006] HENRICKSEN, K.; INDULSKA, J. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, v. 2, n. 1, p. 37–64, 2006.

[Hightower e Borriello 2001] HIGHTOWER, J.; BORRIELLO, G. *Location Systems for Ubiquitous Computing*. 2001.

[Hightower, Lamarca e Smith 2006] HIGHTOWER, J.; LAMARCA, A.; SMITH, I. E. Practical lessons from place lab. *Pervasive Computing, IEEE*, v. 5, n. 3, p. 32–39, 2006.

[Hirschfeld, Costanza e Nierstrasz 2008] HIRSCHFELD, R.; COSTANZA, P.; NIERSTRASZ, O. Context-oriented programming. *Journal of Object Technology (JOT)*, v. 7, n. 3, p. 125–151, 2008.

[Ierusalimschy, Figueiredo e Celes] IERUSALIMSCHY, R.; FIGUEIREDO, L.; CELES, W. *The Implementation of Lua 5.0*. Disponível em: <http://www.tecgraf.puc-rio.br/ lhf/ftp/doc/jucs05.pdf>.

[Intel 2004] INTEL. Endianness white paper. 2004. Disponível em: <http://www.intel.com/design/intarch/papers/endian.pdf>.

[Kiczales et al.] KICZALES, G. et al. Aspect-oriented programming.

[Kindberg e Fox 2002] KINDBERG, T.; FOX, A. *System Software for Ubiquitous Computing*. 2002.

[Kleinberg 2008] KLEINBERG, J. The convergence of social and technological networks. *Commun. ACM*, ACM, New York, NY, USA, v. 51, n. 11, p. 66–72, 2008. ISSN 0001-0782.

[Lindholm e Yellin 1999] LINDHOLM, T.; YELLIN, F. *The Java Virtual Machine specification*. [S.l.]: Addison Wesley, 1999.

[Mani, Ngyuen e Crespi 2009] MANI, M.; NGYUEN, A.-M.; CRESPI, N. What's up: P2p spontaneous social networking. *Pervasive Computing and Communications, IEEE International Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1–2, 2009.

[Milewski 2001] MILEWSKI, B. *C++ In Action: Industrial Strength Programming Techniques*. [S.l.]: Addison Wesley, 2001.

[Moss 2000] MOSS, K. How can I measure Java Code performance ? *Dr. Dobb's Journal*, p. 135–144, October 2000.

[Naismith e Smith 2004] NAISMITH, L.; SMITH, P. Context-sensitive information delivery to visitors in a botanic garden. In: *EDMEDIA World Conference on Educational Multimedia, Hypermedia and Telecommunications*. [S.l.: s.n.], 2004. p. 5525–5530.

[Naseen 2004] NASEEN, M. K. e. a. Implementing strong code mobility. *Information Technology Journal*, p. 188–191, 2004.

[Price e Toscani 2001] PRICE, A. M.; TOSCANI, S. S. *Implementação de linguagens de programação : compiladores*. [S.l.]: Sagra Luzzatto, 2001.

[RFC 819] RFC 819. *Domain naming convention for Internet user applications*. Disponível em: <http://www.faqs.org/rfcs/rfc819.html>.

[RFC 920] RFC 920. *Domain requirements*. Disponível em: <http://www.faqs.org/rfcs/rfc920.html>.

[Rogers et al. 2005] ROGERS, Y. et al. Ubi-learning integrates indoor and outdoor experiences. *Communications of the ACM*, ACM Press, v. 48, n. 1, p. 55–59, January 2005. Disponível em: <http://eprints.ecs.soton.ac.uk/11790/>.

[Román et al. 2002] ROMáN, M. et al. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, v. 1, p. 74–83, 2002.

[Saha e Mukherjee 2003] SAHA, D.; MUKHERJEE, A. Pervasive computing: a paradigm for the 21st century. *Computer*, v. 36, n. 3, p. 25–31, 2003. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1185214>.

[Satyanarayanan 1996] SATYANARAYANAN, M. Fundamental challenges in mobile computing. In: *In Fifteenth ACM Symposium on Principles of Distributed Computing*. [S.l.: s.n.], 1996. p. 1–7.

[Satyanarayanan 2001] SATYANARAYANAN, M. Pervasive computing: vision and challenges. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, v. 8, n. 4, p. 10–17, 2001. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=943998>.

[Sebesta 1999] SEBESTA, R. W. *Concepts of Programming Languages*. [S.l.]: Addison Wesley, 1999.

[Silva et al. 2009] SILVA, J. et al. Distribuição de conteúdo em ambientes cientes de trilhas. *WebMedia*, 2009.

[Syvanen et al. 2005] SYVANEN, A. et al. Supporting pervasive learning environments: adaptability and context awareness in mobile learning. *Wireless and Mobile Technologies in Education*, IEEE International Workshop, 2005.

[Thorn 1997] THORN, T. Programming languages for mobile code. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 29, n. 3, p. 213–239, September 1997. ISSN 0360-0300.

[Tucker 1998] TUCKER, A. Reading Java Class Files in C++. *C/C++ Users Journal*, p. 67–74, April 1998.

[U-BLOX site] U-BLOX site. Disponível em: <http://www.u-blox.com, Último acesso: Fevereiro de 2010.>.

[Ubisense site] UBISENSE site. Disponível em: <http://www.ubisense.net, Último acesso: Fevereiro de 2010.>.

[Ucla et al. 2001] UCLA, A. A. et al. *Fundamental Concepts of Dependability*. 2001.

[Vaughan-Nichols 2009] VAUGHAN-NICHOLS, S. J. Will mobile computing's future be location, location, location? *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 42, n. 2, p. 14–17, 2009. ISSN 0018-9162.

[Warren 1983] WARREN, D. H. D. An abstract prolog instruction set. *SRI International*, October 1983.

[Watts 2003] WATTS, D. J. *Six Degrees: The Science of a Connected Age*. [S.l.]: W. W. Norton & Company, 2003. Paperback. ISBN 0393325423.

[Weiser 1991] WEISER, M. The computer for the 21st century. 1991.

[Yamin e Augustin 2006] YAMIN, A. C.; AUGUSTIN, I. *Computação Pervasiva: Como programar aplicações. Tutorial*. 2006.

[Yau et al. 2003] YAU, S. S. et al. Smart classroom: Enhancing collaborative learning using pervasive computing technology. In: *In ASEE 2003 Annual Conference and Exposition*. [S.l.: s.n.], 2003. p. 13633–13642.

[Yourst 1998] YOURST, M. T. Inside Java Class Files. *Dr. Dobb's Journal*, p. 46–52, January 1998.

Apêndice A

Gramática da UbiLanguage

Arquivo: ubil.g

```
///#########################################################################
///######################### Inicio da gramatica ###########################
///#########################################################################

grammar ubil;

// starting point for parsing a ubil file
compilation_unit :
    (import_declaration)*
    (entity_definition)+
    EOF
    ;

import_declaration : 'import' id1=IDENTIFIER ('.' id2=IDENTIFIER )* ';' ;

entity_definition :
    'entity' IDENTIFIER
    (entity_options)?
    (supported_contexts)?
    (property_definition)*
    (method_definition|service_definition)+
    'end'
    ;

entity_options : '[' opt1=IDENTIFIER ( ',' opt2=IDENTIFIER )* ']' ;

supported_contexts : 'when' '(' expr ')' ;

property_definition : ('prop'|'property') type IDENTIFIER ';' ;

type returns : 'int' | 'real' | 'string' | 'bool' | 'element' | 'userdata' | 'table' | 'tuple' ;

method_definition :
    'method' ('public'|'private')? IDENTIFIER
    '(' (parameters_definition)? ')'
    (returns_definition)?
    (var_definition)*
```

```
    code_block
    'end'
    ;

service_definition :
    'service' IDENTIFIER
    '(' (parameters_definition)? ')'
    (returns_definition)?
    (var_definition)*
    code_block
    'end'
    ;

parameters_definition : type1=type id1=IDENTIFIER ( ',' type2=type id2=IDENTIFIER )* ;

returns_definition : '[' type1=type ( ',' type2=type )* ']' ;

var_definition : ('var'|'variable') type IDENTIFIER ('=' expr)? ';' ;

code_block : (statement)* ;

statement : (simple_statement ';') | block_statement ;

simple_statement : assignment_statement|return_statement|method_invocation|context_interation ;

block_statement : if_statement|iteration_statement ;

assignment_statement :
        var_assignment_statement   | context_assignment_statement | table_assignment_statement
    |   event_assignment_statement | expr
    ;

var_assignment_statement : Id1=IDENTIFIER (',' Id2=IDENTIFIER)* '=' expr ;

context_assignment_statement : context '.' IDENTIFIER '=' expr ;

table_assignment_statement : IDENTIFIER '[' expr ']' '=' expr ;

event_assignment_statement : context_join_event | element_event_assignment_statement ;

element_event_assignment_statement : element=IDENTIFIER '.' event=IDENTIFIER '+=' method=IDENTIFIER ;

context_join_event : context '.' data=IDENTIFIER '.' event=IDENTIFIER '+=' method=IDENTIFIER ;

context_interation :
      context_invocation
   |  data_context_interation
   |  data_context_event
   |  service_context_interation
   ;

context_invocation : '{' expr '}' '.' ('mjoin'|'mleave') '(' argument ')' ;

data_context_interation : '{' expr '}' '.' IDENTIFIER '(' (arg1=argument_list '=>' arg2=argument_list)? ')' ;

data_context_event : '{' expr '}' '.' event=IDENTIFIER '+=' method=IDENTIFIER ;

service_context_interation : '{' expr '}' '.' action=IDENTIFIER '(' (args=argument_list)? ')' ;

rcontext : '{' expr '}' '.' IDENTIFIER ( '(' (argument_list)? ')' )? ;

rtable
```

```
    :    IDENTIFIER '.' 'size' '(' ')'
    |    IDENTIFIER '[' expr ']'
    ;

rtuple : IDENTIFIER '.' ('keys'|'results') '[' expr ']' ;

return_statement : 'return' (expr (',' expr)* )? ;

if_statement
   : 'if' '(' expr ')'
      code_block
      ('else' code_block)? 'end'
   ;

iteration_statement :  for_statement|while_statement|repeat_statement|foreach_statement ;

for_statement
   :  'for' '(' assignment_statement ';' expr ';' assignment_statement ')'
      code_block
      'end'
   ;

while_statement
   :  'while' '(' expr ')'
      code_block
      'end'
   ;

repeat_statement
   :  'repeat'
      code_block
      'until' '(' expr ')'
   ;

foreach_statement
   :  'foreach' '(' IDENTIFIER 'in' rcontext ')'
      code_block
      'end'
   ;

method_invocation : local_method_invocation | element_method_invocation ;

local_method_invocation : methodId=IDENTIFIER '(' (argument_list)? ')' ;

element_method_invocation : elementId=IDENTIFIER '.' methodId=IDENTIFIER '(' (argument_list)? ')' ;

argument_list : argument (',' argument )* ;

argument : expr | literal | 'nil' ;

literal : IDENTIFIER | INTEGER_LITERAL | REAL_LITERAL | STRING_LITERAL | BOOLEAN_LITERAL ;

context_property : context '.' IDENTIFIER ;

context : 'identity' | 'location' | 'activity' | 'time' ;

element_property : IDENTIFIER '.' IDENTIFIER ;

// ---------------------------- Expressoes ---------------------------------

expr : expr_e ( 'or' expr_e )* ;
```

```
expr_e : expr_bit_ou ( 'and' expr_bit_ou )* ;

expr_bit_ou : expr_bit_xou ( '|' expr_bit_xou )* ;

expr_bit_xou : expr_bit_e ( '~' expr_bit_e )* ;

expr_bit_e : expr_igual ( '&' expr_igual )* ;

expr_igual : expr_relacional ( ( '==' | '<>' ) expr_relacional )* ;

expr_relacional : expr_ad ( ( '>' | '>=' | '<' | '<=' ) expr_ad )* ;

expr_ad : expr_multip ( ( '+' | '-' ) expr_multip )* ;

expr_multip : expr_unario ( ( '/' | '*' | '%' ) expr_unario )* ;

expr_unario : ('+' | '-' | 'not')? expr_incdec_prefixo ;

expr_incdec_prefixo : ('++'|'--')? expr_incdec_posfixo ;

expr_incdec_posfixo : expr_elemento ( '++' | '--' )? ;

expr_elemento
  : method_invocation
  | IDENTIFIER
  | literal
  | context_property
  | element_property
  | rcontext
  | rtable
  | rtuple
  | '(' expr ')'
  ;

// LEXER

INTEGER_LITERAL : ('0'..'9')+ ;

REAL_LITERAL : ('0'..'9')* '.' ('0'..'9')+ ;

CHARACTER_LITERAL : '\'' ( ESCAPE_SEQUENCE | ~('\''|'\\') ) '\'' ;

STRING_LITERAL : '"' ( ESCAPE_SEQUENCE | ~('\\'|'"') )* '"' ;

BOOLEAN_LITERAL : 'true' | 'false' ;

fragment ESCAPE_SEQUENCE : '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\') ;

IDENTIFIER : ('_'|LETTER) (LETTER|DIGIT|'_')* ;

fragment LETTER: 'a'..'z' ;

fragment DIGIT : '0'..'9' ;

WS : (' '|'\r'|'\t'|'\u000C'|'\n') ;

COMMENT : '/*' ( options {greedy=false;} : . )* '*/' ;

LINE_COMMENT : '//' ~('\n'|'\r')* '\r'? '\n' ;
```

Apêndice B

Código fonte dos experimentos

Este apêndice contém o código fonte dos cenários apresentados no capítulo 8.

B.1 Educação ubíqua

Arquivo: oportunidades_pedagogicas.ubil

```
import datetime;
import ncurses;

entity start
    prop int _line;
    prop int _obj_count;

    method constructor()
        ncurses.initscr();

        get_infos();
        show_learning_objects();

        ncurses.endwin();
    end

    method get_infos()
        var string object_name;
        var string object_type;
        var string object_publisher;
        var string object_year;

        // Solicita as informacoes sobre o aprendiz
        ncurses.clear();
        ncurses.mvwrite(1, 37, "OPORTUNIDADES PEDAGOGICAS");
        ncurses.mvwrite(3,  2, "NOME.................: "); identity.name             = ncurses.read();
        ncurses.mvwrite(4,  2, "INTERESSE GERAL......: "); identity.general_interest  = ncurses.read();
        ncurses.mvwrite(5,  2, "INTERESSE ESPECIFICO..: "); identity.specific_interest = ncurses.read();

        // Solicita os objetos de aprendizagem do aprendiz
```

```
    ncurses.mvwrite( 8, 39, "OBJETOS DE APRENDIZAGEM");
    ncurses.mvwrite(10, 12, "OBJETO");
    ncurses.mvwrite(10, 67, "TIPO");
    ncurses.mvwrite(10, 77, "QUEM_PUBLICOU");
    ncurses.mvwrite(10, 96, "ANO");
    _line = 11;

    while(object_name <> "fim")
        ncurses.mvwrite(_line, 12, ""); object_name = ncurses.read();

        if (object_name <> "fim")
            ncurses.mvwrite(_line, 67, ""); object_type      = ncurses.read();
            ncurses.mvwrite(_line, 77, ""); object_publisher = ncurses.read();
            ncurses.mvwrite(_line, 96, ""); object_year      = ncurses.read();

            add_object(object_name, object_type, object_publisher, object_year);
        end
        _line++;
    end
end

method show_learning_objects()
    var tuple learner;
    var table learner_list;
    var int   learner_num;

    location.symbolic = "start";

    while(true)
        ncurses.clear();
        {"/PIPCA"}.cpublish("LEARNER", identity.name =>
                identity.general_interest, identity.specific_interest, location.symbolic);

        ncurses.mvwrite(1, 37, "OPORTUNIDADES PEDAGOGICAS");
        ncurses.mvwrite(3,  2, "NOME.................:"); ncurses.mvwrite(3, 26, identity.name);
        ncurses.mvwrite(4,  2, "INTERESSE GERAL.....:"); ncurses.mvwrite(4, 26, identity.general_interest);
        ncurses.mvwrite(5,  2, "INTERESSE ESPECIFICO..:"); ncurses.mvwrite(5, 26, identity.specific_interest);
        ncurses.mvwrite(6,  2, "LOCALIZACAO SIMBOLICA.:"); ncurses.mvwrite(6, 26, location.symbolic);

        ncurses.mvwrite( 8, 39, "OBJETOS DE APRENDIZAGEM");
        ncurses.mvwrite(10,  2, "APRENDIZ");
        ncurses.mvwrite(10, 12, "OBJETO");
        ncurses.mvwrite(10, 67, "TIPO");
        ncurses.mvwrite(10, 77, "QUEM_PUBLICOU");
        ncurses.mvwrite(10, 96, "ANO");
        _line = 11;

        // Verifica por aprendizes com interesses similares
        learner_list = {"/PIPCA"}.clist();
        for(learner_num=1; learner_num<=learner_list.size();learner_num++)
            learner = learner_list[learner_num];
            if (learner.keys[1] == "LEARNER")
                if (learner.keys[2] <> identity.name and learner.results[1] == identity.general_interest
                        and learner.results[3] == location.symbolic)
                    // Quando encontra um aprendiz com interesses similares,
                    // lista os seus objetos de aprendizagem
                    show_learner_objects(learner.keys[2]);
                end
            end
        end

        ncurses.refresh();
```

```
            datetime.sleep(10);
        end
    end

    method add_object(string name, string type, string event, string year)
        _obj_count++;
        {"/PIPCA"}.cpublish("OBJECT", identity.name, _obj_count => name, type, event, year);
    end

    method show_learner_objects(string learner_name)
        var tuple obj;
        var table obj_list;
        var int   obj_number;

        // Lista os objetos de aprendizagem disponibilizados pelo aprendiz
        obj_list = {"/PIPCA"}.clist();
        for(obj_number=1; obj_number<=obj_list.size(); obj_number++)
            obj = obj_list[obj_number];
            if (obj.keys[1] == "OBJECT")
                if (obj.keys[2] == learner_name)
                    ncurses.mvwrite(_line, 2, learner_name);
                    ncurses.mvwrite(_line, 12, obj.results[1]);
                    ncurses.mvwrite(_line, 67, obj.results[2]);
                    ncurses.mvwrite(_line, 77, obj.results[3]);
                    ncurses.mvwrite(_line, 96, obj.results[4]);
                    _line++;
                end
            end
        end
    end
end
```

B.2 Comércio ubíquo

Arquivo: busca_ofertas.ubil

```
import ncurses;
import ncurses_menu;
import tools;

entity start
    method constructor()
        var tuple offer;
        var table offer_list;
        var int offer_number;
        var int count = 0;
        var table offer_code_in_menu;
        var table service_name_in_menu;

        ncurses.initscr();
        ncurses.mvwrite(1, 35, "BUSCA OFERTAS");
        ncurses.mvwrite(3,  2, "NOME...........: "); identity.name            = ncurses.read();
        ncurses.mvwrite(4,  2, "CATEGORIA......: "); identity.categoria       = ncurses.read();
        ncurses.mvwrite(5,  2, "COMPONENTE BASE: "); identity.componente_base = ncurses.read();
        ncurses.mvwrite(6,  2, "VALOR MAXIMO...: "); identity.valor_maximo    = tools.atof(ncurses.read());
```

```
        ncurses.mvwrite(7,  2, "NUMERO PEDIDO..:");

        ncurses.mvwrite(9, 30, "OFERTAS DE PRODUTOS");
        ncurses.mvwrite(10, 3, "LOJISTA                        OFERTA                                VALOR");

        // Monta um menu com as ofertas que atendem 100% das caracteristicas definidas pelo cliente
        ncurses_menu.clear();
        offer_list = {"/shopping/praca_alimentacao"}.clist();
        for(offer_number=1; offer_number<=offer_list.size(); offer_number++)
            offer = offer_list[offer_number];
            if (offer.results[2] == identity.categoria
                    and offer.results[3] == identity.componente_base
                    and offer.results[5] <= identity.valor_maximo)
                ncurses_menu.new_item(tools.pad(offer.keys[1], 30) + tools.pad(offer.keys[2],5) +
                        tools.pad(offer.results[1], 30) + tools.ftoa(offer.results[5]));
                count++;
                offer_code_in_menu[count]   = offer.keys[2];
                service_name_in_menu[count] = offer.results[6];
            end
        end

        if (count > 0)
                    // Cliente seleciona uma das ofertas
            offer_number = ncurses_menu.show(5, 80, 11, 1);

            // Executa o servico disponibilizado pelo lojista para realizar o pedido
            identity.checkout_number =
                    {"/shopping/praca_alimentacao"}.srun(service_name_in_menu[offer_number],
                    identity.name, offer_code_in_menu[offer_number]);

            // Exibe o numero do pedido
            ncurses.mvwrite(7, 19, identity.checkout_number);
        end

        ncurses.refresh();
        ncurses.getch();
        ncurses.endwin();
    end
end
```

Arquivo: ofertas_restaurante.ubil

```
entity restaurante_offers when (time.hour >= 11 and time.hour <= 14)
    // Ofertas para o horario do almoco
    method constructor()
        {"/shopping/praca_alimentacao"}.cpublish(
            "Restaurante da Olga", "1" => "massa a carbonara", "refeicao", "massa",
            "ovos, creme de leite, bacon, queijo parmesao, macarrao", 13.00,
            "restaurante_checkout");
        {"/shopping/praca_alimentacao"}.cpublish(
            "Restaurante da Olga", "2" =>
            "massa alho e oleo", "refeicao", "massa", "alho, azeite, macarrao", 11.00,
            "restaurante_checkout");
        {"/shopping/praca_alimentacao"}.cpublish(
            "Restaurante da Olga", "3" =>
            "picanha na chapa", "refeicao", "carne", "carne, arroz, salada", 21.00,
            "restaurante_checkout");
    end
```

```
end

entity restaurante_offers when (time.hour >= 18 and time.hour <= 22)
    // Ofertas para o horario da janta
    method constructor()
        {"/shopping/praca_alimentacao"}.cpublish(
            "Restaurante da Olga", "1" => "sopa creme ervilha", "refeicao", "sopa",
            "creme de erviha, leite", 9.00,
            "restaurante_checkout");
        {"/shopping/praca_alimentacao"}.cpublish(
            "Restaurante da Olga", "2" =>
            "panqueca a moda da casa", "refeicao", "massa", "queijo, presunto, oregano, azeitona, bacon", 9.00,
            "restaurante_checkout");
    end
end

entity start
    prop int last_order;

    method constructor()
        var element offers;

        // Publica o servico que deve ser utilizado para realizacao de pedidos
        {"/shopping/praca_alimentacao"}.spublish("restaurante_checkout");

        // Publica as ofertas
        offers = restaurante_offers.new();
        io.key_press();
    end

    service restaurante_checkout(string client_name, string offer_code) [int]
        io.writeln("[Restaurante] Recebido pedido de ", client_name,
                ", referente a oferta ", offer_code);
        last_order++;
        io.writeln("[Restaurante] Numero do pedido: ", last_order);

        return last_order;
    end
end
```

Arquivo: ofertas_lancheria.ubil

```
entity start
    prop int last_order;

    method constructor()
        // Publica o servico que deve ser utilizado para realizacao de pedidos
        {"/shopping/praca_alimentacao"}.spublish("lancheria_checkout");

        // Publica as ofertas
        {"/shopping/praca_alimentacao"}.cpublish(
            "Lancheria do Victorio", "1" =>
            "pizza presunto", "lanche", "massa", "presunto, queijo, oregano", 11.00,
            "lancheria_checkout");

        io.key_press();
    end
```

```
    service lancheria_checkout(string client_name, string offer_code) [int]
        io.writeln("[Lancheria] Recebido pedido de ", client_name,
                ", referente a oferta ", offer_code);
        last_order++;
        io.writeln("[Lancheria] Numero do pedido: ", last_order);

        return last_order;
    end
end
```

B.3 Redes sociais

Arquivo: parque.ubil

```
import datetime;

entity start
    prop int    _last_park_vacancy;
    prop int    _last_team_vacancy;
    prop string _soccer_field_reserve;
    prop string _soccer_field_play_time;
    prop string _bowl_field_reserve;

    method constructor()
        // Publica informacoes sobre o parque
        {"parque"}.cpublish("Roteiros de caminhadas" =>
                "Iniciando pelo Mini zoológico, percorra ...");
        {"parque"}.cpublish("Localizacao de servicos" =>
                "Banheiros: na entrada do parque. Bicicletário: ao lado do estacionamento.");

        // Publica os servicos disponiveis
        {"parque"}.spublish("reserva_vagas_estacionamento");
        {"parque"}.spublish("reserva_vagas_time_basquete");
        {"parque"}.spublish("reserva_quadra_futebol");
        {"parque"}.spublish("reserva_cancha_bocha");

        _soccer_field_play_time = "16hs";
        io.key_press();
    end

    service reserva_vagas_estacionamento(string person_name) [int]
        io.writeln("[Parque] Recebido pedido de reserva de vaga de estacionamento para ", person_name);

        if (_last_park_vacancy < 100)
            _last_park_vacancy++;
            io.writeln("[Parque] Vaga numero ", _last_park_vacancy, " reservada para ", person_name);
            return _last_park_vacancy;
        else
            io.writeln("[Parque] Estacionamento lotado ", person_name);
            return 0;
        end
    end

    service reserva_vagas_time_basquete(string person_name) [string]
        io.writeln("[Parque] Recebido pedido de reserva de vaga no time de basquete para ", person_name);
```

```
        if (_last_team_vacancy < 10)
            _last_team_vacancy++;
            io.writeln("[Parque] Vaga numero ", _last_team_vacancy, " reservada para ", person_name);
            return _soccer_field_play_time;
        else
            io.writeln("[Parque] Time de basquete lotado ", person_name);
            return "";
        end
    end

    service reserva_quadra_futebol(string penson_name) [bool]
        io.writeln("[Parque] Recebido pedido de reserva da quadra de futebol para ", person_name);

        if (_soccer_field_reserve == "")
            _soccer_field_reserve = person_name;
            io.writeln("[Parque] Quadra de futebol reservada para ", person_name);
            return true;
        else
            io.writeln("[Parque] Quadra de futebol já está reservada ", person_name);
            return false;
        end
    end

    service reserva_cancha_bocha(string penson_name) [bool]
        io.writeln("[Parque] Recebido pedido de reserva da cancha de bocha para ", person_name);

        if (_bowl_field_reserve == "")
            _bowl_field_reserve = person_name;
            io.writeln("[Parque] Cancha de bocha reservada para ", person_name);
            return true;
        else
            io.writeln("[Parque] Cancha de bocha já está reservada ", person_name);
            return false;
        end
    end
end
```

Arquivo: solicita_reserva_estacionamento.ubil

```
entity start
    method constructor()
        var int vacancy_number;
        var string username;

        io.write("Nome do usuario: "); username = io.readln();

        vacancy_number = {"parque"}.srun("reserva_vagas_estacionamento", username);

        if (vacancy_number == 0)
            io.writeln("Estacionamento lotado...");
        else
            io.writeln("Numero da vaga de estacionamento: ", vacancy_number);
        end

        io.key_press();
    end
end
```

Arquivo: solicita_reserva_time_basquete.ubil

```
entity start
    method constructor()
        var string play_time;
        var string username;

        io.write("Nome do usuario: "); username = io.readln();

        play_time = {"parque"}.srun("reserva_vagas_time_basquete", username);

        if (play_time == "")
            io.writeln("Nao existem mais vagas no time...");
        else
            io.writeln("Horario do jogo: ", play_time);
        end

        io.key_press();
    end
end
```

Arquivo: solicita_info_visita_assistida.ubil

```
import datetime;

entity start
    method constructor()
        var tuple  visita_assistida;
        var int    last_latitude;
        var int    last_longitude;

        identity.name      = "graciele";
        location.latitude  = 0;
        location.longitude = 0;

        // Exibe as informacoes sobre o servico
        visita_assistida = {"parque"}.cfind("Visita assistida");
        io.writeln("Informacoes sobre o servico Visita Assistida:");
        io.writeln("    Tempo de duracao   : ", visita_assistida.results[1]);
        io.writeln("    Descricao          : ", visita_assistida.results[2]);
        io.writeln("    Locais de interesse: ", visita_assistida.results[3]);
        io.writeln("");

        io.writeln("**** Informacoes sobre a fauna e flora ****");

        while(true)
            if (last_latitude <> location.latitude or last_longitude <> location.longitude)
                // Sempre que a localizacao do usuario eh alterada, executa o servico visita_assistida para
                                // saber se existem informacoes sobre a nova localizacao
                last_latitude  = location.latitude;
                last_longitude = location.longitude;
                io.writeln("Sobre este local (latitude=", last_latitude, ", longitude=", last_longitude, "): ",
```

```
                        {"parque"}.srun("visita_assistida", last_latitude, last_longitude));
            end

            datetime.sleep(2);
        end
    end
end
```

Arquivo: visita_assistida.ubil

```
entity start
    method constructor()
        // Publica o servico e as informacoes sobre ele
        {"parque"}.spublish("visita_assistida");
        {"parque"}.cpublish(
            "Visita assistida" => "1 hora",
            "Baseado na sua localização, são disponibilizadas informações sobre a fauna e a flora local.",
            "Mini zoológico, Plantas silvestres, Flores e Animais silvestres.");

        io.key_press();
    end

    service visita_assistida(int y, int x) [string]
        var table _y1_desc;
        var table _x1_desc;
        var table _y2_desc;
        var table _x2_desc;
        var table _desc;
        var int   _desc_count;
        var int   desc_number;

        io.writeln("[Visita assistida] Recebido pedido para coordenadas ", y, ", ", x);

        // Registra as areas de interesse

        // Area 1
        _desc_count++;
        _y1_desc[_desc_count] = 13;
        _x1_desc[_desc_count] = 04;
        _y2_desc[_desc_count] = 14;
        _x2_desc[_desc_count] = 12;
        _desc   [_desc_count] = "Mini zoo tem animais como girafa, urso, macaco prego, entre outros.";

        // Area 2
        _desc_count++;
        _y1_desc[_desc_count] = 13;
        _x1_desc[_desc_count] = 14;
        _y2_desc[_desc_count] = 14;
        _x2_desc[_desc_count] = 20;
        _desc   [_desc_count] =
            "Plantas como aroeira, cavalinha, pau-santo e tucum podem ser encontrados neste local.";

        // Area 3
        _desc_count++;
        _y1_desc[_desc_count] = 03;
        _x1_desc[_desc_count] = 14;
        _y2_desc[_desc_count] = 04;
        _x2_desc[_desc_count] = 20;
```

```
        _desc   [_desc_count] =
                "Estes animais silvestres vivem livremente aqui desde a construção do parque.";

        // Verifica se a localizacao fornecida faz parte de uma das areas de interesse.
        for(desc_number = 1; desc_number <= _desc_count; desc_number++)
            if (y >= _y1_desc[desc_number] and y <= _y2_desc[desc_number] and
                x >= _x1_desc[desc_number] and x <= _x2_desc[desc_number])
                // Caso afirmativo, retorna a informacao sobre a area
                return _desc[desc_number];
            end
        end
                // Nenhuma informacao sobre a localizacao atual
        return "Nenhuma informação sobre este local.";
    end
end
```

Apêndice C

Instruções da UbiVM

Este apêndice apresenta as instruções presentes na UbiVM. O apêndice contém a descrição do que cada instrução realiza, sua sintaxe, seu *opcode*, operandos necessárias na pilha para sua execução, resultados retornados na pilha, restrições e um exemplo de uso.

As instruções estão agrupadas em onze categorias. São elas:

- Uso geral: são instruções que armazenam ou retiram literais da pilha, transferindo-os para os parâmetros, propriedades, variáveis locais ou resultados de métodos e serviços;
- Aritméticas: são instruções aritméticas que retiram um ou mais literais da pilha, executam sua operação, e inserem o resultado na pilha novamente;
- Lógicas: são instruções lógicas que retiram um ou mais literais da pilha, executam sua operação, e inserem o resultado na pilha novamente;
- Relacionais: são instruções relacionais que retiram um ou mais literais da pilha, executam sua operação, e inserem o resultado na pilha novamente;
- Suporte a OO: são instruções que suportam as funcionalidades da OO;
- Controle do fluxo de execução: são instruções, condicionais ou incondicionais, que desviam o fluxo normal de execução da aplicação;
- Contextos: são instruções que interagem com os membros, conteúdos e serviços existentes nos contextos;
- Mobilidade: são instruções que realizam a mobilidade forte de código;
- Concorrência: são instruções que gerenciam os fluxos de execução existentes na aplicação;

•Manipulação de tabelas: são instruções que manipulam os dados das tabelas;

•Manipulação de tuplas: são instruções que manipulam as tuplas.

C.1 Instruções de uso geral

Instrução LDCONST (LoaD CONSTant)

Descrição: Carrega a constante *const* na pilha.
Sintaxe: ldconst const
Opcode: 0
Operandos: Nenhum.
Resultados: A constante *const*.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Carregando uma string !"
ldconst 1
lcall io.writeln
```

Instrução LDCONST_0 (LoaD CONSTant 0)

Descrição: Carrega a constante inteira *0* na pilha.
Sintaxe: ldconst_0
Opcode: 1
Operandos: Nenhum.
Resultados: A constante *0*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_0
ldconst_1
lcall io.writeln
```

Instrução LDCONST_1 (LoaD CONSTant 1)

Descrição: Carrega a constante inteira *1* na pilha.
Sintaxe: ldconst_1
Opcode: 2
Operandos: Nenhum.
Resultados: A constante *1*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_1
ldconst_1
lcall io.writeln
```

Instrução LDCONST_2 (LoaD CONSTant 2)

Descrição: Carrega a constante inteira *2* na pilha.
Sintaxe: ldconst_2
Opcode: 3
Operandos: Nenhum.
Resultados: A constante *2*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_2
ldconst_1
lcall io.writeln
```

Instrução LDCONST_3 (LoaD CONSTant 3)

Descrição: Carrega a constante inteira *3* na pilha.
Sintaxe: ldconst_3
Opcode: 4
Operandos: Nenhum.
Resultados: A constante *3*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_3
ldconst_1
lcall io.writeln
```

Instrução LDCONST_4 (LoaD CONSTant 4)

Descrição: Carrega a constante inteira *4* na pilha.
Sintaxe: ldconst_4
Opcode: 5
Operandos: Nenhum.
Resultados: A constante *4*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_4
ldconst_1
lcall io.writeln
```

Instrução LDCONST_5 (LoaD CONSTant 5)

Descrição: Carrega a constante inteira *5* na pilha.
Sintaxe: ldconst_5
Opcode: 6
Operandos: Nenhum.
Resultados: A constante *5*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_5
ldconst_1
lcall io.writeln
```

Instrução LDCONST_0_0 (LoaD CONSTant 0.0)

Descrição: Carrega a constante real *0.0* na pilha.
Sintaxe: ldconst_0_0
Opcode: 7
Operandos: Nenhum.
Resultados: A constante *0.0*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_0_0
ldconst_1
lcall io.writeln
```

Instrução LDCONST_1_0 (LoaD CONSTant 1.0)

Descrição: Carrega a constante real *1.0* na pilha.
Sintaxe: ldconst_1_0
Opcode: 8
Operandos: Nenhum.
Resultados: A constante *1.0*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_1_0
ldconst_1
lcall io.writeln
```

Instrução LDCONST_2_0 (LoaD CONSTant 2.0)

Descrição: Carrega a constante real *2.0* na pilha.
Sintaxe: ldconst_2_0
Opcode: 9
Operandos: Nenhum.
Resultados: A constante *2.0*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_2_0
ldconst_1
lcall io.writeln
```

Instrução LDCONST_3_0 (LoaD CONSTant 3.0)

Descrição: Carrega a constante real *3.0* na pilha.
Sintaxe: ldconst_3_0
Opcode: 10
Operandos: Nenhum.
Resultados: A constante *3.0*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_3_0
ldconst_1
lcall io.writeln
```

Instrução LDCONST_4_0 (LoaD CONSTant 4.0)

Descrição: Carrega a constante real *4.0* na pilha.
Sintaxe: ldconst_4_0
Opcode: 11
Operandos: Nenhum.
Resultados: A constante *4.0*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_4_0
ldconst_1
lcall io.writeln
```

Instrução LDCONST_5_0 (LoaD CONSTant 5.0)

Descrição: Carrega a constante real *5.0* na pilha.
Sintaxe: ldconst_5_0
Opcode: 12
Operandos: Nenhum.
Resultados: A constante *5.0*.
Restrições: Nenhuma.
Exemplo:

```
ldconst_5_0
ldconst_1
lcall io.writeln
```

Instrução LDVAR (Load VARiable)

Descrição: Carrega o conteúdo da variável *var1* na pilha.
Sintaxe: ldvar var1
Opcode: 13
Operandos: Nenhum.
Resultados: Conteúdo da variável.
Restrições: Nenhuma.
Exemplo:

```
.var int var1
ldconst 10
stvar var1
ldconst "Numero:"
ldvar var1
ldconst 2
lcall io.writeln
```

Instrução STVAR (STore VARiable)

Descrição: Armazena o literal na variável *var1*.
Sintaxe: stvar var1
Opcode: 14
Operandos: Literal.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
.var int var1
ldconst 10
stvar var1
ldconst "Numero:"
ldvar var1
ldconst 2
lcall io.writeln
```

Instrução LDPARAM (Load PARAMeter)

Descrição: Carrega o conteúdo do parâmetro *par1* na pilha.
Sintaxe: ldparam par1
Opcode: 15
Operandos: Nenhum.
Resultados: Conteúdo do parâmetro.
Restrições: Nenhuma.
Exemplo:

```
.par int par1
ldconst 10
stparam par1
ldconst "Numero:"
ldpar par1
ldconst 2
lcall io.writeln
```

Instrução STPARAM (STore PARAMeter)

Descrição: Armazena o literal no parâmetro *par1*.
Sintaxe: stpar par1
Opcode: 16
Operandos: Literal.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
.par int par1
ldconst 10
stpar par1
ldconst "Numero:"
ldpar par1
ldconst 2
lcall io.writeln
```

Instrução LDPROP (Load PROPerty)

Descrição: Carrega o conteúdo da propriedade *prop1* na pilha.
Sintaxe: ldprop prop1
Opcode: 17
Operandos: Nenhum.
Resultados: Conteúdo da propriedade.
Restrições: Nenhuma.
Exemplo:

```
.prop int prop1
ldconst 10
stprop prop1
ldconst "Numero:"
ldprop prop1
ldconst 2
lcall io.writeln
```

Instrução STPROP (STore PROPerty)

Descrição: Armazena o literal na propriedade *prop1*.
Sintaxe: stprop prop1
Opcode: 18
Operandos: Literal.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
.prop int prop1
ldconst 10
stprop prop1
ldconst "Numero:"
ldprop prop1
ldconst 2
lcall io.writeln
```

Instrução LDNIL (LoaD NIL)

Descrição: Carrega o literal *nil* na pilha.
Sintaxe: ldnil
Opcode: 19
Operandos: Nenhum.
Resultados: Literal *nil*.
Restrições: Nenhuma.
Exemplo:

```
ldnil
ldconst_1
lcall io.writeln
```

Instrução STNIL (STore NIL)

Descrição: Descarta o último literal da pilha.
Sintaxe: stnil
Opcode: 20
Operandos: Literal.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "literal que será descartado..."
stnil
```

Instrução LDRESULT (LoaD RESULT)

Descrição: Carrega na pilha o resultado de número *index* da sub-rotina.
Sintaxe: ldresult index
Opcode: 21
Operandos: Nenhum.
Resultados: Resultado da sub-rotina.
Restrições: Nenhuma.
Exemplo:

```
.result 0 int
ldconst 30
stresult 0
ldresult 0
ldconst_1
lcall io.writeln
```

Instrução STRESULT (STore RESULT)

Descrição: Armazena o literal como o resultado de número *index* da sub-rotina.
Sintaxe: stresult index
Opcode: 22
Operandos: Literal.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
.result 0 int
ldconst 30
stresult 0
```

C.2 Instruções aritméticas

Instrução ADD (ADDs)

Descrição: Realiza a operação "Op1 + Op2".
Sintaxe: add
Opcode: 23
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros, reais ou strings.
Exemplo:

```
ldconst 22
ldconst 33
add
ldconst_1
lcall io.writeln
```

Instrução SUB (SUBtract)

Descrição: Realiza a operação "Op1 - Op2".
Sintaxe: sub
Opcode: 24
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros ou reais.
Exemplo:

```
ldconst 15
ldconst 13
sub
ldconst_1
lcall io.writeln
```

Instrução MUL (MULtiply)

Descrição: Realiza a operação "Op1 * Op2".
Sintaxe: mul
Opcode: 25
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros ou reais.
Exemplo:

```
ldconst 15
ldconst 2
mul
ldconst_1
lcall io.writeln
```

Instrução DIV (DIVide)

Descrição: Realiza a operação "Op1 / Op2".
Sintaxe: div
Opcode: 26
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros ou reais.
Exemplo:

```
ldconst 15
ldconst 3
div
ldconst_1
lcall io.writeln
```

C.3 Instruções lógicas

Instrução AND (AND)

Descrição: Realiza a operação "Op1 and Op2".
Sintaxe: and
Opcode: 27
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais booleanos.
Exemplo:

```
ldconst true
ldconst false
and
ldconst 1
lcall io.writeln
```

Instrução OR (OR)

Descrição: Realiza a operação "Op1 or Op2".
Sintaxe: or
Opcode: 28
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais booleanos.
Exemplo:

```
ldconst true
ldconst false
or
ldconst 1
lcall io.writeln
```

Instrução NOT (NOT)

Descrição: Realiza a operação "not op".
Sintaxe: not
Opcode: 29
Operandos: Op.
Resultados: Resultado da operação.
Restrições: Somente para literais booleanos.
Exemplo:

```
ldconst true
not
ldconst 1
lcall io.writeln
```

Instrução XOR (eXclusive OR)

Descrição: Realiza a operação "Op1 xor Op2".
Sintaxe: xor
Opcode: 30
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais booleanos.
Exemplo:

```
ldconst true
ldconst false
xor
ldconst 1
lcall io.writeln
```

C.4 Instruções relacionais

Instrução GE (Greater or Equal)

Descrição: Realiza a operação "Op1 >= Op2".
Sintaxe: ge
Opcode: 31
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros, reais ou *strings*.
Exemplo:

```
ldconst 2
ldconst 7
ge
ldconst 1
lcall io.writeln
```

Instrução LE (Less or Equal)

Descrição: Realiza a operação "Op1 <= Op2".
Sintaxe: le
Opcode: 32
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros, reais ou *strings*.
Exemplo:

```
ldconst 2
ldconst 7
le
ldconst 1
lcall io.writeln
```

Instrução EQ (EQual)

Descrição: Realiza a operação "Op1 == Op2".
Sintaxe: eq
Opcode: 33
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros, reais ou *strings*.
Exemplo:

```
ldconst 2
ldconst 7
eq
ldconst 1
lcall io.writeln
```

Instrução NE (Not Equal)

Descrição: Realiza a operação "Op1 <> Op2".
Sintaxe: ne
Opcode: 34
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros, reais ou *strings*.
Exemplo:

```
ldconst 2
ldconst 7
ne
ldconst 1
lcall io.writeln
```

Instrução GT (Greater Than)

Descrição: Realiza a operação "Op1 > Op2".
Sintaxe: gt
Opcode: 35
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros, reais ou *strings*.
Exemplo:

```
ldconst 2
ldconst 7
gt
ldconst 1
lcall io.writeln
```

Instrução LT (Less Than)

Descrição: Realiza a operação "Op1 < Op2".
Sintaxe: lt
Opcode: 36
Operandos: Op1 e Op2.
Resultados: Resultado da operação.
Restrições: Somente para literais inteiros, reais ou *strings*.
Exemplo:

```
ldconst 2
ldconst 7
lt
ldconst 1
lcall io.writeln
```

C.5 Instruções para suporte a OO

Instrução NEWELEM (NEW ELEMent)

Descrição: Cria um novo elemento a partir da entidade *name*.
Sintaxe: newelem name
Opcode: 37
Operandos: Nenhum.
Resultados: Elemento criado.
Restrições: Nenhuma.
Exemplo:

```
newelem entity1
mcall show
```

Instrução DELELEM (DELete ELEMent)

Descrição: Destrói um elemento.
Sintaxe: delelem
Opcode: 38
Operandos: Elemento a ser destruído.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
.var element e
newelem entity1
stvar e
ldvar e
mcall show
ldvar e
delelem
```

Instrução LDSELF (LoaD SELF)

Descrição: Carrega o elemento atual em execução na pilha.
Sintaxe: ldself
Opcode: 39
Operandos: Nenhum.
Resultados: Elemento atual.
Restrições: Nenhuma.
Exemplo:

```
ldself
mcall show
```

C.6 Instruções para controle do fluxo de execução

Instrução LCALL (Library CALL)

Descrição: Executa a função *library_function* da biblioteca.
Sintaxe: lcall library_function
Opcode: 40
Operandos: Número de argumentos e os argumentos da função.
Resultados: Resultados da função (se pertinente).
Restrições: Nenhuma.
Exemplo:

```
ldconst "Oi !!!"
ldconst_1
lcall io.writeln
```

Instrução MCALL (Method CALL)

Descrição: Executa o método *method* do elemento, com seus argumentos.
Sintaxe: mcall method
Opcode: 41
Operandos: Elemento e os argumentos do método.
Resultados: Resultados do método (se pertinente).
Restrições: Nenhuma.
Exemplo:

```
ldconst 10
ldconst 20
ldself
mcall soma
```

Instrução EXIT (EXIT)

Descrição: Finaliza a execução do aplicativo informando o estado *status* para o SO.
Sintaxe: exit
Opcode: 42
Operandos: Status.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Hello world !!!"
ldconst_1
lcall io.writeln
ldconst 4
exit
ldconst "Eu nao vou ser executado !!!"
ldconst_1
lcall io.writeln
```

Instrução HALT (HALT)

Descrição: Interrompe a execução do aplicativo informando o estado *status* para o SO.
Sintaxe: halt
Opcode: 43
Operandos: Status.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "A UbiVM vai ser interrompida..."
lconst_1
lcall io.writeln
ldconst 3
halt
```

Instrução JMP (JuMP)

Descrição: Desvia o fluxo de execução para o *label*.
Sintaxe: jmp label
Opcode: 44
Operandos: Nenhum.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
:inicio     ldconst "Estou em loop !!!"
            ldconst_1
            lcall io.writeln
            jmp inicio
```

Instrução IFNOT (IF NOT)

Descrição: Desvia o fluxo de execução para o *label* caso *Jump* seja *false*.
Sintaxe: ifnot label
Opcode: 45
Operandos: Jump.
Resultados: Nenhum.
Restrições: Somente para literais booleanos.
Exemplo:

```
            ldconst_2
            ldconst_5
            le
            ifnot label_nao
            ldconst "le"
            ldconst_1
            lcall io.writeln
            jmp fim
:label_nao  ldconst "not le"
            ldconst_1
            lcall io.writeln
:fim
```

Instrução IF (IF)

Descrição: Desvia o fluxo de execução para o *label* caso *Jump* seja *true*.
Sintaxe: if label
Opcode: 46
Operandos: Jump.
Resultados: Nenhum.
Restrições: Somente para literais booleanos.
Exemplo:

```
            ldconst_2
            ldconst_5
            le
            if label_sim
            ldconst "not le"
            ldconst_1
            lcall io.writeln
            jmp fim
:label_sim  ldconst "le"
            ldconst_1
            lcall io.writeln
:fim
```

Instrução RET (RETurn)

Descrição: Encerra a execução da sub-rotina atualmente em execução.
Sintaxe: ret
Opcode: 47
Operandos: Nenhum.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Vou retornar desta sub-rotina"
ldconst_1
lcall io.writeln
ret
```

Instrução NOP (Null OPcode)

Descrição: Não faz nada, e somente serve para ajustes diretamente em um arquivo uvm.
Sintaxe: nop
Opcode: 48
Operandos: Nenhum.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Hello world !!!"
nop
ldconst_1
lcall io.writeln
```

C.7 Instruções que manipulam contextos

Instrução MJOIN (Member JOIN)

Descrição: Realiza o *join* do elemento atual no contexto *context_ name* com a identificação *identification*.
Sintaxe: mjoin
Opcode: 49
Operandos: context_name e identification.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
ldconst "Minha ID"
mjoin
```

Instrução MLEAVE (Member LEAVE)

Descrição: Realiza o *leave* do elemento atual do contexto *context_name*.
Sintaxe: mleave
Opcode: 50
Operandos: context_name.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
mleave
```

Instrução MLIST (Members LIST)

Descrição: Lista os membros do contexto.
Sintaxe: mlist
Opcode: 51
Operandos: Nome do contexto.
Resultados: Lista com informações sobre os membros encontrados.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Membros: "
ldconst "MeuContexto"
mlist
ldconst 2
lcall io.writeln
```

Instrução CPUBLISH (Content PUBLISH)

Descrição: Publica o conteúdo no contexto determinado com a chave e resultado informados.
Sintaxe: cpublish
Opcode: 52
Operandos: Número de informações da chave, número de informações do resultado, as chaves, os resultados e o nome do contexto.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
ldconst "Numero 1"
ldconst "54321"
ldconst_1
ldconst_1
cpublish
```

Instrução CGET (Content GET)

Descrição: Bloqueia sua execução até que o conteúdo, com a chave informada, possa ser obtido e retirado do contexto.
Sintaxe: cget
Opcode: 53
Operandos: Número de informações da chave, a chave e o nome do contexto.
Resultados: Dado obtido.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Informacoes inseridas: "
ldconst "MeuContexto"
ldconst "Numero 1"
ldconst 1
cget
ldconst 2
lcall   "io.writeln"
```

Instrução CGETNB (Content GET Non Block)

Descrição: Obtém e retira do contexto o conteúdo com a chave informada; caso o conteúdo não esteja disponível, não bloqueia sua execução e insere *false* na pilha.
Sintaxe: cgetnb
Opcode: 54
Operandos: Número de informações da chave, a chave e o nome do contexto.
Resultados: *Flag* indicando se o conteúdo foi obtido e o conteúdo (caso $flag == true$).
Restrições: Nenhuma.
Exemplo:

```
ldconst "Informacoes inseridas: "
ldconst "MeuContexto"
ldconst "Numero 1"
ldconst 1
cgetnb
ldconst 2
lcall   "io.writeln"
```

Instrução CFIND (Content FIND)

Descrição: Bloqueia sua execução até que o conteúdo, com a chave informada, possa ser obtido do contexto.
Sintaxe: cfind
Opcode: 55
Operandos: Número de informações da chave, a chave e o nome do contexto.
Resultados: Conteúdo obtido.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Informacoes inseridas: "
ldconst "MeuContexto"
ldconst "Numero 1"
ldconst 1
cfind
ldconst 2
lcall   "io.writeln"
```

Instrução CFINDNB (Content FIND Non Block)

Descrição: Obtém do contexto o conteúdo com a chave informada; caso o conteúdo não esteja disponível, não bloqueia sua execução e insere *false* na pilha.
Sintaxe: cfindnb
Opcode: 56
Operandos: Número de informações da chave, a chave e o nome do contexto.
Resultados: *Flag* indicando se o conteúdo foi obtido e o conteúdo (caso *flag*==*true*).
Restrições: Nenhuma.
Exemplo:

```
ldconst "Informacoes inseridas: "
ldconst "MeuContexto"
ldconst "Numero 1"
ldconst 1
cfindnb
ldconst 2
lcall   "io.writeln"
```

Instrução CREM (Content REMove)

Descrição: Bloqueia sua execução até que o conteúdo, com a chave informada, possa ser removido do contexto.
Sintaxe: crem
Opcode: 57
Operandos: Número de informações da chave, a chave e o nome do contexto.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
ldconst "Numero 1"
ldconst 1
crem
```

Instrução CREMNB (Content REMove Non Block)

Descrição: Remove do contexto o conteúdo com a chave informada; caso o conteúdo não esteja disponível, não bloqueia sua execução e insere *false* na pilha.
Sintaxe: cremnb
Opcode: 58
Operandos: Número de informações da chave, a chave e o nome do contexto.
Resultados: *Flag* indicando se o conteúdo foi obtido e o conteúdo (caso *flag*==*true*).
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
ldconst "Numero 1"
ldconst 1
cremnb
```

Instrução CLIST (Contents LIST)

Descrição: Lista os conteúdos do contexto.
Sintaxe: clist
Opcode: 59
Operandos: Nome do contexto.
Resultados: Lista com informações sobre os conteúdos encontrados.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Conteudos: "
ldconst "MeuContexto"
clist
ldconst 2
lcall io.writeln
```

Instrução SPUBLISH (Service PUBLISH)

Descrição: Publica o serviço *service_name* no contexto *context_name*.
Sintaxe: spublish
Opcode: 60
Operandos: context_name e service_name.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
ldconst "MeuServico"
spublish
```

Instrução SFIND (Service FIND)

Descrição: Bloqueia sua execução até que o serviço possa ser encontrado no contexto.
Sintaxe: sfind
Opcode: 61
Operandos: Nome do serviço e nome do contexto.
Resultados: Informações sobre o serviço encontrado.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Informacoes sobre o servico:"
ldconst "MeuContexto"
ldconst "MeuServico"
sfind
ldconst 2
lcall   "io.writeln"
```

Instrução SFINDNB (Service FIND Non Block)

Descrição: Obtém do contexto informações sobre o serviço; caso o serviço não esteja disponível, não bloqueia sua execução e insere *false* na pilha.
Sintaxe: sfindnb
Opcode: 62
Operandos: Nome do serviço e nome do contexto.
Resultados: *Flag* indicando se o serviço foi encontrado e as informações sobre o serviço (caso *flag*==*true*).
Restrições: Nenhuma.
Exemplo:

```
ldconst "Informacoes sobre o serviço: "
ldconst "MeuContexto"
ldconst "MeuServico"
ldconst 1
sfindnb
ldconst 2
lcall   "io.writeln"
```

Instrução SRUN (Service RUN)

Descrição: Bloqueia sua execução até que o serviço service_name possa ser executado no contexto context_name.
Sintaxe: srun
Opcode: 63
Operandos: Número de argumentos do serviço, os argumentos, o nome do serviço (service_name) e o nome do contexto (context_name).
Resultados: Resultados do serviço (se pertinente).
Restrições: Nenhuma.
Exemplo:

```
.var 0 element s
newelem "servico"
stvar s
ldconst "Soma: "
ldconst "context"
ldconst "soma"
ldconst 10
ldconst 20
ldconst 2
srun
ldconst 2
lcall "io.writeln"
```

Instrução SRUNNB (Service RUN Non Block)

Descrição: Executa o serviço do contexto; caso o serviço não esteja disponível, não bloqueia sua execução e insere *false* na pilha.
Sintaxe: srunnb
Opcode: 64
Operandos: Número de argumentos do serviço, os argumentos, o nome do serviço (service_name) e o nome do contexto (context_name).
Resultados: *Flag* indicando se o serviço foi executado e seus valores de retorno, caso flag==true.
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
ldconst "MeuServico"
srunnb
```

Instrução SREM (Service REMove)

Descrição: Bloqueia sua execução até que o serviço service_name possa ser removido do contexto context_name.
Sintaxe: srem
Opcode: 65
Operandos: service_name e context_name.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
ldconst "MeuServico"
srem
```

Instrução SREMNB (Service REMove Non Block)

Descrição: Remove o serviço do contexto; caso o serviço não esteja disponível, não bloqueia sua execução e insere *false* na pilha.
Sintaxe: sremnb
Opcode: 66
Operandos: Nome do serviço e nome do contexto.
Resultados: *Flag* indicando se o serviço foi removido.
Restrições: Nenhuma.
Exemplo:

```
ldconst "MeuContexto"
ldconst "MeuServico"
sremnb
```

Instrução SLIST (Services LIST)

Descrição: Lista os serviços do contexto.
Sintaxe: slist
Opcode: 67
Operandos: Nome do contexto.
Resultados: Lista com informações sobre os serviços encontrados.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Servicos: "
ldconst "MeuContexto"
slist
ldconst 2
lcall io.writeln
```

Instrução LDCONTEXTI (LoaD CONTEXT Information)

Descrição: Carrega na pilha o conteúdo da informação *info* do contexto.
Sintaxe: ldcontexti info
Opcode: 68
Operandos: Nenhum.
Resultados: Conteúdo da informação.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Alex"
stcontexti "identity.name"
ldconst "Meu nome: "
ldcontexti "identity.name"
ldconst_2
lcall io.writeln
```

Instrução STCONTEXTI (STore CONTEXT Information)

Descrição: Altera o conteúdo da informação *info* do contexto.
Sintaxe: stcontexti info
Opcode: 69
Operandos: Conteúdo da informação.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "Alex"
stcontexti "identity.name"
ldconst "Meu nome: "
ldcontexti "identity.name"
ldconst_2
lcall io.writeln
```

C.8 Instruções para mobilidade

Instrução MOVE (MOVE)

Descrição: Realiza a mobilidade de código do aplicativo.
Sintaxe: move
Opcode: 70
Operandos: Destino do aplicativo.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
ldconst "meu_desktop"
move
```

C.9 Instruções para concorrência

Instrução CMCALL (Concurrent Method CALL)

Descrição: Executa o método *method* do elemento, com seus argumentos, de forma concorrente.
Sintaxe: cmcall method
Opcode: 71
Operandos: Elemento e os argumentos do método.
Resultados: Identificação do fluxo de execução do método concorrente.
Restrições: Nenhuma.
Exemplo:

```
ldconst 10
ldconst 20
ldself
cmcall soma
cmwait
ldconst_1
lcall io.writeln
```

Instrução CMWAIT (Concurrent Method WAIT)

Descrição: Aguarda o término do método concorrente.
Sintaxe: cmwait
Opcode: 72
Operandos: Identificação do método concorrente (retornado por CMCALL).
Resultados: Resultados do método (se pertinente).
Restrições: Nenhuma.
Exemplo:

```
ldconst "Resultado da soma de 10 e 20: "
ldconst 10
ldconst 20
ldself
cmcall soma
cmwait
ldconst_2
lcall io.writeln
```

C.10 Instruções para manipulação de tabelas

Instrução STTAB (STore TABle)

Descrição: Armazena a informação *info* na posição *index* da tabela *tab*.
Sintaxe: sttab tab
Opcode: 73
Operandos: Info e index.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
.var 0 table tab
ldconst 1
ldconst 100
sttab tab
ldconst "tab[1]="
ldconst 1
ldtab tab
ldconst 2
lcall "io.writeln"
```

Instrução LDTAB (LoaD TABle)

Descrição: Carrega na pilha a informação da posição *index* da tabela *tab*.
Sintaxe: ldtab tab
Opcode: 74
Operandos: Index.
Resultados: Info.
Restrições: Nenhuma.
Exemplo:

```
.var 0 table tab
ldconst 1
ldconst 100
sttab tab
ldconst "tab[1]="
ldconst 1
ldtab tab
ldconst 2
lcall "io.writeln"
```

Instrução TABSIZE (TABle SIZE)

Descrição: Carrega na pilha o número de informações da tabela *tab*.
Sintaxe: tabsize tab
Opcode: 75
Operandos: Nenhum.
Resultados: Número de informações.
Restrições: Nenhuma.
Exemplo:

```
.var 0 table tab
ldconst 1
ldconst 100
sttab tab
ldconst "Numero de informações: "
tabsize tab
ldconst 2
lcall "io.writeln"
```

C.11 Instruções para manipulação de tuplas

Instrução STTUPLEK (STore TUPLE Key)

Descrição: Armazena a informação *info* em *tuple.key*[*index*].
Sintaxe: sttuplek tuple
Opcode: 76
Operandos: Index e info.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
.var tuple tupla
ldconst 100
ldconst 1
sttuplek tupla
ldconst "Valor da chave 1 da tupla: "
ldconst 1
ldtuplek tupla
ldconst_2
lcall "io.writeln"
```

Instrução LDTUPLEK (LoaD TUPLE Key)

Descrição: Obtém a informação *info* de *tuple.key*[*index*].
Sintaxe: ldtuplek tuple
Opcode: 77
Operandos: Index.
Resultados: Info.
Restrições: Nenhuma.
Exemplo:

```
.var tuple tupla
ldconst 100
ldconst 1
sttuplek tupla
ldconst "Valor da chave 1 da tupla: "
ldconst 1
ldtuplek tupla
ldconst_2
lcall "io.writeln"
```

Instrução STTUPLER (STore TUPLE Result)

Descrição: Armazena a informação *info* em *tuple.result*[*index*].
Sintaxe: sttupler tuple
Opcode: 78
Operandos: Index e info.
Resultados: Nenhum.
Restrições: Nenhuma.
Exemplo:

```
.var tuple tupla
ldconst 100
ldconst 1
sttupler tupla
ldconst "Valor do resultado 1 da tupla: "
ldconst 1
ldtupler tupla
ldconst_2
lcall "io.writeln"
```

Instrução LDTUPLER (LoaD TUPLE Result)

Descrição: Obtém a informação *info* de *tuple.result*[*index*].
Sintaxe: ldtupler tuple
Opcode: 79
Operandos: Index.
Resultados: Info.
Restrições: Nenhuma.
Exemplo:

```
.var tuple tupla
ldconst 100
ldconst 1
sttupler tupla
ldconst "Valor do resultado 1 da tupla: "
ldconst 1
ldtupler tupla
ldconst_2
lcall "io.writeln"
```

Apêndice D

Formato do Arquivo UVM

Um aplicativo compilado para ser executado pela UbiVM é representado utilizando-se um formato binário independente de hardware e sistema operacional, tipicamente armazenado em um arquivo, conhecido como o formato de arquivo UVM. Este arquivo contém a descrição das entidades existentes no aplicativo, definindo suas propriedades, métodos e constantes necessárias durante sua execução.

Este apêndice apresenta o formato deste arquivo e as possíveis verificações que podem ser realizadas ao carregar este arquivo.

D.1 Características básicas

O arquivo UVM contém a representação binária de um programa em UbiL. Este arquivo define precisamente a *Constant Pool* (onde são armazenadas as constantes utilizadas pelos *opcodes*) e a representação das entidades (onde são armazenadas as propriedades e os métodos).

Este arquivo é peça fundamental na independência de hardware e sistema operacional da UbiVM. Ele possui estruturas simples que facilitam sua geração por outras ferramentas e a implementação de novas versões da UbiVM em outros ambientes. Como possui um formato de arquivo binário padronizado, outros compiladores podem gerar este arquivo e usufruir dos recursos presentes na UbiVM.

Em vários pontos do arquivo faz-se necessário indicar o tipo do dado. Os tipos existentes (tabela D.1) são baseados nos tipos existentes na UbiL (tabela 5.7). O tipo faz-se necessário na definição das constantes, propriedades, parâmetros, retornos das sub-rotinas e variáveis.

Para a definição deste arquivo foi utilizado o *endianness "Little Endian"* [Intel 2004, Adiga 2007].

Tabela D.1: Tipos de dados existentes no arquivo UVM

Tipo	**Valor**
String	S
Integer	I
Real	R
Boolean	B
Element	E
Table	L
Tuple	T

D.2 Estrutura do arquivo

A figura D.1 contém a estrutura geral do arquivo. O arquivo é organizado em três seções (estruturas): *File header*, que contém informações de identificação do arquivo; *Constant pool*, que contém as constantes utilizadas no arquivo e *Entity pool*, que contém a descrição das entidades.

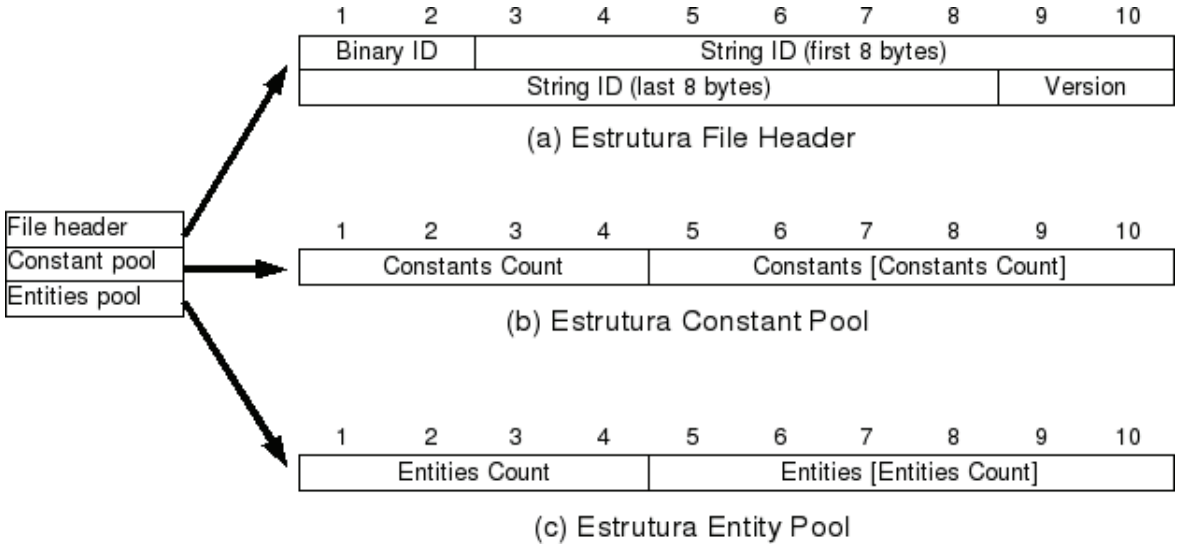


Figura D.1: Estrutura geral do arquivo

A estrutura ***File header*** (figura D.1(a)) identifica o arquivo. Contém a assinatura binária que identifica o arquivo (*Binary ID*), assinatura alfanumérica (*String ID*) e o número da versão do arquivo (*version*) visando compatibilidade com futuras versões.

A estrutura ***Constant pool*** é uma imagem das constantes existentes em um programa UbiL, sendo similar à tabela de símbolos de um compilador tradicional [Aho, Sethi e Ullman 1988, Lindholm e Yellin 1999]. Constante é qualquer *token* de um programa que não faça parte diretamente do léxico da

linguagem como mensagens aos usuários, nomes de variáveis, nomes de entidades, nomes de métodos, números e strings.

A *Constant pool* (figura D.1(b)) armazena duas informações: o número de entradas no *array Constants* (*Constants count*) e o array que descreve cada uma destas constantes (*Constants*).

Cada entrada no *array Constants* é uma estrutura *Constant definition* (figura D.2) que contém as informações relativas a uma constante: o tipo da constante (*type*), o tamanho de cada constante (*constant size*) e o valor da constante (*value*).

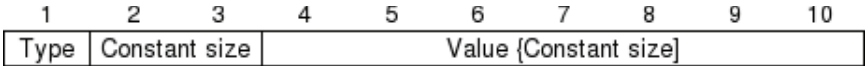


Figura D.2: Estrutura *Constant definition*

A estrutura ***Entity pool*** (figura D.1(c)) contém a descrição das entidades. Esta estrutura contém duas informações: o número de entradas no *array Entity Pool* (*Entities count*) e o *array* que armazena a descrição de cada entidade (*Entities*).

Cada entrada no *array Entities* é uma estrutura *Entity description* (figura D.3) que contém as informações relativas a uma entidade: índice da constante que contém o nome da entidade (*Index name*), número de propriedades (*Properties Count*), número de métodos (*Methods Count*), *array* contendo estas propriedades (*Properties*) e o *array* contendo estes métodos (*Methods*).

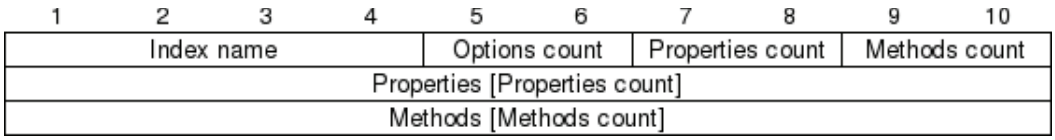


Figura D.3: Estrutura *Entity description*

Cada entrada do *array Properties* é uma estrutura *Property description* (figura D.4) e contém as informações relativas a uma propriedade: visibilidade da propriedade (*visibility*), tipo (*type*) e o índice da constante que contém o nome da propriedade (*index name*).

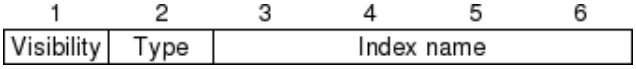


Figura D.4: Estrutura *Property description*

Methods armazena os métodos existentes em uma entidade através da estrutura *Method description*. Esta estrutura (figura D.5) contém o índice da

constante que armazena o nome do método (*Index name*), o número de parâmetros (*Parameters count*) existentes no *array Parameters*, o número de resultados (*Results count*) existentes no *array Results*, o número de variáveis (*Variables count*) existentes no *array Variables* e o número de instruções (*Instructions count*) existentes no *array Instructions*.

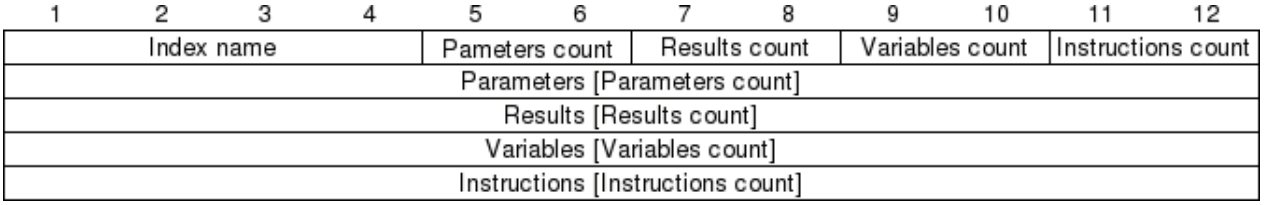


Figura D.5: Estrutura *Method description*

Parameters armazena cada parâmetro em uma estrutura *Parameter definition* (figura D.6). Esta estrutura contém o tipo (*type*) e o índice da constante (*index name*) que identifica o parâmetro.

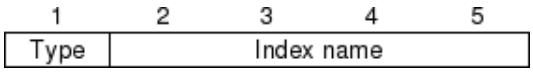


Figura D.6: Estrutura *Parameter definition*

Results armazena estruturas *Result definiton* (figura D.7) contendo o tipo (*type*) de cada resultado.

Figura D.7: Estrutura *Result definiton*

Variables armazena estruturas *Variable definition* (figura D.8) contendo o tipo (*type*) e o índice da constante (*index name*) que identifica a variável.

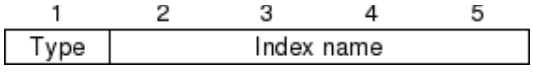


Figura D.8: Estrutura *Variable definition*

Instructions armazena estruturas *Instruction definition* (figura D.9) contendo a instrução (*opcode*) e o seu argumento (*argument*). O argumento é opcional e existe em algumas instruções apenas.

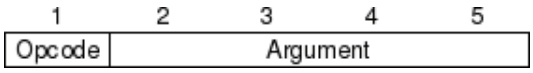


Figura D.9: Estrutura *Instruction definition*

D.3 Verificações no arquivo

Quando um arquivo é carregado, deve ser verificado se este arquivo tem o formato básico de um arquivo. Os primeiros quatro *bytes* devem conter a assinatura correta (*0xBEBEF0F0*), o arquivo não pode estar truncado ou ter *bytes* extras no final, e a *constant pool* não deve conter qualquer informação não reconhecida.

Uma análise do fluxo de dados em cada método deve ser realizada, assegurando que:

- métodos são invocados com o número apropriado de argumentos;
- todos os *opcodes* tem o tipo apropriado de argumentos na pilha;
- os argumentos para cada invocação de método é compatível com a assinatura deste método;
- não irá ocorrer *overflow* ou *underflow* na pilha.

As verificações nas instruções do *array* de código são as seguintes:

- os índices de propriedades, parâmetros, variáveis e resultados são válidos;
- o destino das instruções de salto (*if, ifnot, jmp*) é um *opcode* válido de uma instrução;
- cada operando que indique um índice nos elementos da *Constant pool* deve ser um índice válido nesta tabela;
- as referências para a *Constant pool* devem ser uma entrada para um tipo apropriado (conforme o *opcode*);
- o código não deve terminar com uma instrução incompleta;
- toda sub-rotina deve ter, ao final de seu código, a instrução *ret*, com exceção para os construtores da entidade *Start*, que devem ter a instrução *exit* no final do seu código;
- não deve sobrar ou faltar argumentos na pilha durante a execução das sub-rotinas;
- todas as funções necessárias das bibliotecas do SO devem existir.