UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ANDRÉ LUÍS PITOMBEIRA

UM GUIDELINE PARA O DESENVOLVIMENTO DE SOFTWARE UBÍQUO ORIENTADO A TAREFAS

QUIXADÁ
2011

ANDRÉ LUÍS PITOMBEIRA

UM GUIDELINE PARA O DESENVOLVIMENTO DE SOFTWARE UBÍQUO ORIENTADO A TAREFAS

Trabalho de Conclusão de Curso submetido à Coordenação do Curso de Graduação em Sistemas de Informação da Universidade Federal do Ceará como requisito parcial para obtenção do grau de Bacharel.

Área de concentração: computação

Orientador Prof. MSc. Lincoln Souza Rocha

QUIXADÁ
2011

P76g Pitombeira, André Luís.
Um guideline para o desenvolvimento de software ubíquo orientado a tarefas / André Luís Pitombeira. – Quixadá, 2011.

42.: il.; 31 cm.
Cópia de computador (printout(s)).

Orientador: Prof.MSc. Lincoln Souza Rocha
Monografia (graduação em Sistemas de Informação) – Universidade Federal do Ceará, Campus Quixadá, Quixadá, Ceará, 2011.

1. Padrões de software 2.Software ubíquo 3.Desenvolvimento de software. 4. Modelo de desenvolvimento orientado a tarefas I. Rocha, Lincoln Souza (orient.) II. Universidade Federal do Ceará – Curso de Bacharelado em Sistemas de Informação III. Título

CDD 005.12

ANDRÉ LUÍS PITOMBEIRA

UM GUIDELINE PARA O DESENVOLVIMENTO DE SOFTWARE UBÍQUO ORIENTADO A TAREFAS

Trabalho de Conclusão de Curso submetido à Coordenação do Curso de Graduação em Sistemas de Informação da Universidade Federal do Ceará como requisito parcial para obtenção do grau de Bacharel.

Área de concentração: computação

Aprovado em: _____ / _____ / 2011.

BANCA EXAMINADORA

____
Prof. MSc. Lincoln Souza Rocha (Orientador)
Universidade Federal do Ceará-UFC

____
Prof. MSc. Camilo Camilo Almendra
Universidade Federal do Ceará-UFC

____
Prof. MSc. Vitor Almeida dos Santos
Universidade Federal do Ceará-UFC

Aos meus pais...

AGRADECIMENTOS

Aos meus pais, irmãos e a toda minha família que, com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa de minha vida. Ao professor Lincoln Souza Rocha pela paciência na orientação e incentivo que tornaram possível a conclusão desta monografia. Ao professor e tutor do PET Davi Romero de Vasconcelos, pelo convívio, pelo apoio, pela compreensão e pela amizade. A todos os professores da UFC, que foram tão importantes na minha vida acadêmica e no desenvolvimento desta monografia. Ao prefeito de Russas-CE Raimundo Cordeiro pelo apoio logístico. Aos amigos e colegas, pelo incentivo e pelo apoio constantes.

"As tecnologias mais profundas e duradouras são aquelas que desaparecem."
(Mark Weiser)

UM GUIDELINE PARA O DESENVOLVIMENTO DE SOFTWARE UBÍQUO ORIENTADO A TAREFAS

RESUMO

O processo de desenvolvimento de software é um conjunto de atividades que levam à produção padronizada e sistemática de software. A adoção de um processo de software tende a diminuir a quantidade de erros de projeto, evitando gastos com retrabalho e aumentando a qualidade geral do software produzido. Contudo, mesmo assumindo um papel importante na atividade produtiva de software, foram encontrados poucos trabalhos tendo com objetivo padronizar o desenvolvimento de software ubíquo.. Dessa forma, o presente trabalho surgiu da necessidade de sistematizar o processo de desenvolvimento de software ubíquo. Para isso, um *guideline* para o desenvolvimento de software ubíquo orientado a tarefas é proposto e uma aplicação exemplo é modelada usando o *guideline*. Adicionalmente, com objetivo de iniciar a avaliação da proposta, o *guideline* proposto foi apresentado a um painel de especialista, do GREat/UFC, onde foram feitas críticas e sugestões de melhorias, as quais foram incorporadas na versão final. Com o desenvolvimento de uma aplicação, foi constatado que é possível projetar um software ubíquo a partir do conceito de tarefas, permitindo ao desenvolvedor lidar com um maior nível de abstração.

Palavras chave: Processo de Software, Software Ubíquo, Modelo de Desenvolvimento Orientado a Tarefas.

A GUIDELINE FOR THE DEVELOPMENT OF UBIQUITOUS TASK ORIENTED SOFTWARE

ABSTRACT

The process of software development is a set of activities that lead to the standardized and systematic production of software. Adopting a software development process tends to diminish the quantity of project errors, avoiding the costs of rework and increasing the general quality of the software produced. However, although it takes on an important role in software production activities, few efforts aiming at standardization of the development of ubiquitous software can be found in literature. Consequently, this work arose from the need to systemize the development process of ubiquitous software. Thus, a guideline is proposed for the development of task-oriented ubiquitous software, and an example of an application is modeled according to this guideline. Additionally, in order to initiate the evaluation of the proposal, the proposed guideline was presented to a panel of GREat/UFC specialists, who presented feedback and suggestions for improvements that were incorporated to the final version.

Key words: Software Process, Ubiquitous Software, Task-oriented Development Model.

LISTA DE ILUSTRAÇÕES

Figura 1 – Etapas do Processo de Desenvolvimento de Software.. 14
Figura 2 – Visão geral da computação ubíqua 17
Figura 3 – Computação Orientada a Tarefas... 22
Figura 4 – Estrutura Geral do Guideline 29
Figura 5 – Análise de Atividades... 30
Figura 6 – Projeto de Tarefas 31
Figura 7 – Estacionar 35
Figura 8 – Modelo de Contexto... 36

Tabela 1 – Dimensões da Computação Ubíqua (Adaptado de (Lyyttnem; Yoo, 2002))......... 17
Tabela 2 – Requisitos do Software Ubíquo.. 26
Tabela 3 – Contexto de Ativação... 35
Tabela 4 – Lista de Serviços Necessários no Estacionamento Ubíquo 36
Tabela 5 - Atores.. 36
Tabela 6 – Fluxo Base... 37
Tabela 7 – Contexto de Adaptação.. 37
Tabela 8 – Fluxo Adaptativo 37

SUMÁRIO

1 INTRODUÇÃO 11

2 FUNDAMENTAÇÃO TEÓRICA 14

2.1 O processo de desenvolvimento de software tradicional 14
2.2 Software Ubíquo 15
2.2.1 Computação Ubíqua 15
2.2.2 Contexto e Sensibilidade ao Contexto 18
2.2.2.1 Características do software sensível ao contexto 19
2.2.2.2 Requisitos para a construção de software sensível ao contexto 20
2.3 Modelo de desenvolvimento orientado a tarefas 21

3 GUIDELINE PARA O DESENVOLVIMENTO DE SOFTWARE UBÍQUO ORIENTADO A TAREFAS 25

3.2 Fundamentação do Guideline 26
3.3 Guideline 29

4 APLICAÇÃO EXEMPLO 34

5 CONSIDERAÇÕES FINAIS 39

1 INTRODUÇÃO

O processo de desenvolvimento de software é um conjunto de atividades que levam à produção de software. Estes processos são complexos e, como todos os processos intelectuais e criativos, dependem do julgamento humano (Sommerville, 2007). A complexidade corresponde à sobreposição das complexidades relativas ao desenvolvimento dos seus diversos componentes: software, hardware, procedimentos, etc. Isto se reflete no alto número de projetos de software que não chegam ao fim, ou que extrapolam recursos de tempo e de dinheiro alocados (Bezerra, 2007).

Segundo um estudo feito pelo *STADISH Group* sobre projetos de desenvolvimento de software (Chaos Report, 2009), podemos ver os principais problemas no desenvolvimento de software.

Porcentagem de Projetos que terminam dentro do prazo estimado: 32%;

Porcentagem de Projetos que são descontinuados dentro do prazo estimado: 24%;

Porcentagem de Projetos acima do custo esperado: 44%.

As tentativas de lidar com essa complexidade e minimizar os problemas, inerentes ao desenvolvimento de software, envolvem a definição de processos de desenvolvimento de software. O processo deve envolver todas as atividades necessárias para definir, implementar, testar e manter um produto de software (Bezerra, 2007).

Contudo, nem todos os benefícios que a engenharia de software trouxe para o desenvolvimento de software convencional podem ser totalmente aplicados no domínio do desenvolvimento de software ubíquo (aquele desenvolvido sobre os preceitos da Computação Ubíqua) (Weiser, 1991), que como tal, deve prover ao usuário proatividade e invisibilidade na execução de tarefas.

O desenvolvimento de software ubíquo é uma atividade árdua, porque além de serem aplicações mais complexas, não existe uma metodologia padrão para o seu desenvolvimento (Choi, 2008). O software ubíquo possui diversas características que tornam seu processo de desenvolvimento mais complicado, como por exemplo, a heterogeneidade de dispositivos, diferentes formas de comunicação e utilização de contexto.

Não obstante esses desafios, a demanda por software ubíquo tem apresentado um grande crescimento nos últimos anos. Esse fato tem despertado o interesse tanto da indústria

como da academia (Choi, 2008). Todavia, existem poucas pesquisas que mostram como aplicar os princípios da engenharia de software no desenvolvimento deste tipo de software. A literatura apresenta alguns trabalhos isolados envolvendo disciplinas específicas da engenharia de software (i.e., elicitação de requisitos, modelagem de contexto, programação e testes) para o desenvolvimento de software ubíquo, conforme definido por (Choi, 2008), (Bulcão Neto, 2006) e (Vieira, 2009). Contudo, esses trabalhos não proveem informações suficientes para o desenvolvimento sistemático deste tipo de software na prática (Choi, 2008).

Além da complexidade inerente ao desenvolvimento de software ubíquo, a escassez de técnicas que descrevam como aplicar os princípios da engenharia de software no desenvolvimento de aplicações para esse domínio torna o desenvolvimento ainda mais complexo, menos sistemático e consequentemente pouco produtivo.

Nesse cenário, a investigação de alguma forma de padronizar e lidar com a complexidade do desenvolvimento de software ubíquo torna-se um trabalho desafiador e, sobretudo, relevante, podendo representar um passo à frente rumo à criação de um processo de desenvolvimento para este domínio de aplicação.

Diante disto, para diminuir a complexidade no desenvolvimento de software ubíquo, consideramos neste trabalho o modelo de desenvolvimento orientado a tarefas, que permite ao desenvolvedor abstrair a complexidade associada ao desenvolvimento de software ubíquo, pois o foco é dado à tarefa, o que torna o processo de desenvolvimento menos complexo e mais intuitivo (Loke, 2009).

A sistematização do desenvolvimento é feita através de uma sequência de passos que deve ser seguida pelos desenvolvedores que desejem desenvolver software ubíquo. Estes passos estão dispostos em um *guideline* que fornece todas as atividades, de forma macro, que devem ser seguidas para desenvolver um software ubíquo orientado a tarefas.

O presente trabalho, então, se propõe a definir um *guideline* para o desenvolvimento de software ubíquo, a partir do modelo de desenvolvimento orientado a tarefas. Espera-se com isto, sistematizar o desenvolvimento de software ubíquo e definir quais atividades devem ser realizadas ao longo do desenvolvimento do software.

A metodologia do trabalho consistiu de um levantamento bibliográfico das principais características do software ubíquo e dos requisitos que estes devem atender. Utilizando a abstração que o modelo de desenvolvimento orientado a tarefas fornece, são definidas atividades que permitam satisfazer esses requisitos. O *guideline* foi apresentado a um painel

de especialistas do GREat (Grupo de Pesquisas em Redes de Computadores, Engenharia de Software e Sistemas) da Universidade Federal do Ceará e foi desenvolvida uma aplicação para exemplificar a utilização do *guideline.*

2 FUNDAMENTAÇÃO TEÓRICA

2.1 O processo de desenvolvimento de software tradicional

Conforme definido anteriormente, um processo de software é um conjunto de atividades que leva à produção de um produto de software. Para (Sommerville, 2007), as atividades fundamentais de um processo de software podem ser visualizadas na Figura 01.

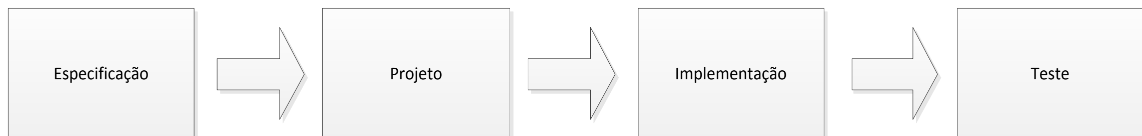

Figura 1 – Etapas do Processo de Desenvolvimento de Software

A Figura 1 nos permitir fazer uma abstração do processo de desenvolvimento de software, ao mostrar as etapas de desenvolvimento como um fluxo de atividades que devem ser seguidas.

Especificação: é o processo para compreender e definir quais serviços são necessários e identificar as restrições de operação e de desenvolvimento do sistema:

Estudo de Viabilidade: é feita uma avaliação para verificar se as necessidades dos usuários identificadas podem ser satisfeitas por meio de tecnologias atuais de software e hardware;

Elicitação e Análise de Requisitos: é o processo de derivação de requisitos de sistemas através da observação de sistemas existentes, discussões com usuários potenciais e compradores, análise de tarefas, dentre outros;

Especificação de Requisitos: atividade de traduzir as informações coletadas durante a atividade de análise em um documento que define os requisitos do sistema;

Validação de Requisitos: essa atividade verifica os requisitos em relação ao realismo, consistência e abrangência.

Projeto e Implementação: o estágio de implementação do desenvolvimento de software é o processo de conversão de uma especificação de sistema em executável. O projeto se software é a descrição da estrutura do software a ser implementada, dos dados que são parte do sistema, das interfaces entre os componentes do sistema e, às vezes, dos algoritmos usados:

Projeto de Arquitetura: os subsistemas constituintes do sistema e os seus relacionamentos são identificados e documentados;

Especificação Abstrata: para cada subsistema são produzidas uma especificação abstrata dos serviços sob as quais ele deve operar;

Projeto de Interface: para cada subsistema é projetada e documentada a interface com outros subsistemas;

Projeto de Componentes: os serviços são alocados aos componentes e as interfaces desses componentes são projetadas;

Projeto de Estrutura de Dados: as estruturas de dados usadas na implementação do sistema são projetadas detalhadamente e especificadas;

Projeto de Algoritmo: os algoritmos usados para fornecer os serviços são projetados detalhadamente e especificados.

Teste: A validação de software ou mais genericamente, verificação e validação (V & V) destina-se a mostrar que um sistema está em conformidade com sua especificação e que atende às expectativas do cliente que está adquirindo o sistema:

Teste de Componente: os componentes individuais são testados para garantir que operem corretamente;

Teste de Sistema: os componentes são interligados para compor o sistema;

Teste de Aceitação: este é o estágio final do processo de teste, antes que o sistema seja aceito para o uso operacional.

2.2 Software Ubíquo

2.2.1 Computação Ubíqua

A história da computação pode ser dividida em função da forma como os usuários interagiam com os computadores. Em uma fase anterior, tínhamos os mainframes que se caracterizavam pela forma de interação "vários usuários por máquina". Na fase atual, temos os microcomputadores, em que a interação é da forma "um usuário por máquina". A fase futura será marcada pelo surgimento de um novo paradigma de interação, a computação ubíqua, que possibilitará a interação de "um usuário para várias máquinas" (Rocha, 2007).

A computação ubíqua, idealizada pelo visionário *Mark Weiser* (Weiser, 1991), tem como finalidade fazer com que o uso dos computadores se torne mais amigável, disponibilizando para o usuário o acesso a informação e computação a todo instante e em qualquer lugar. A ideia central da computação ubíqua é fazer com que a relação entre pessoas e computadores seja marcada pela forma proativa com que os computadores atuam na resolução de problemas e no fornecimento de informação relevante e contextualizada, de maneira transparente, para seus usuários (Rocha, 2007).

De uma maneira geral, poderíamos entender a computação ubíqua como sendo uma computação distribuída, realizada por uma diversidade de dispositivos computacionais inseridos no ambiente que atuam de forma discreta e imperceptível. Os dispositivos podem estar embutidos em objetos de baixa ou nenhuma mobilidade (i.e., aparelhos de TV, lâmpadas e ar condicionado) ou com um alto grau de mobilidade (e.g., PDAs, *smartphones* e veículos automotivos) (Rocha, 2007).

Os princípios fundamentais da computação ubíqua são:

- **Descentralização:** vários dispositivos interagindo, cada um com sua função específica, em um ambiente comum, onde os sistemas computacionais são quem decidem qual o mais adequado para realizar uma determinada tarefa;
- **Diversificação:** diversidade de dispositivos interagindo entre si, onde cada dispositivo possui funcionalidades específicas, que o tornam mais adequado para execução de uma determinada tarefa;
- **Conectividade:** os dispositivos podem se locomover de forma transparente entre redes heterogêneas, que devem possuir padrões robustos para englobar os mais diversos dispositivos;
- **Simplicidade:** as tarefas executadas pelos dispositivos devem ser feitas da forma mais simples possível.

Podemos observar nestes princípios que a computação ubíqua está intimamente relacionada com a computação móvel e a computação pervasiva. A computação móvel é caracterizada pela mobilidade dos dispositivos computacionais interconectados por rede, enquanto a computação pervasiva é vista como um conjunto de dispositivos computacionais dispersos no ambiente que oferecem serviços ao usuário em qualquer lugar e a qualquer hora. Assim, a computação ubíqua surge como uma forma de integrar a mobilidade com a

capacidade dos dispositivos estarem embarcados no ambiente. Na Tabela 01 apresenta um quadro com as dimensões da computação ubíqua proposto por (Lyyttnem; Yoo, 2002).

Tabela 1 – Dimensões da Computação Ubíqua (Adaptado de (Lyyttnem; Yoo, 2002)).

	Computação Pervasiva	Computação Móvel	Computação Ubíqua
Mobilidade	Baixo	Alto	Alto
Grau de Embarcamento	Alto	Baixo	Alto

Podemos ver como estas três áreas estão relacionadas na figura abaixo. A Figura 2 mostra a computação ubíqua como sendo a interseção entre a computação móvel e a computação pervasiva, pois apresenta as características de ambos os modelos de computação.

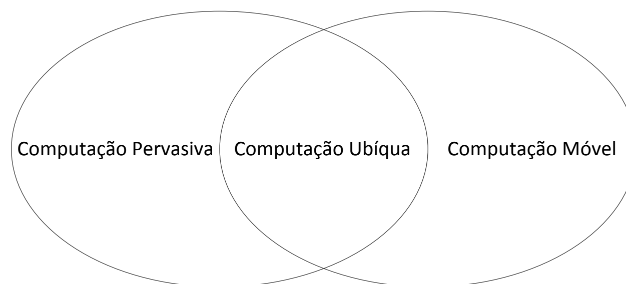

Figura 2 – Visão geral da computação ubíqua

Os ambientes ubíquos podem ser caracterizados como quaisquer espaços físicos (casa, colégio, hospital, empresa) com diversos dispositivos computacionais capazes de realizar alguma tarefa para os frequentadores (usuários) destes ambientes. Tal capacidade necessita que os dispositivos estejam em sintonia com os outros dispositivos e os frequentadores do lugar, o que torna as aplicações ubíquas altamente dependentes do contexto (Rocha, 2007).

O contexto é fundamental para que as aplicações ubíquas atinjam seu propósito, permitindo amplificar a interação humana com serviços prestados por dispositivos computacionais, que se adaptam às circunstâncias em que estão sendo usados. Desta forma, a computação ubíqua comporta um modelo que possibilita usuários, serviços e recursos descobrirem outros usuários, serviços e recursos, integrando-os de uma forma mais útil (Coutaz et. al, 2005).

Ainda, segundo Coutaz et al (2005) o contexto, ao prover um mapeamento e uma visão estruturada do mundo em que o sistema opera, contribui para que sejam atingidos os

objetivos da computação ubíqua, eliminando do usuário a necessidade de configurações manuais para a realização de algum serviço provido por um dispositivo computacional.

2.2.2 Contexto e Sensibilidade ao Contexto

As pessoas, em geral, conseguem se comunicar umas com as outras devido a diversos fatores, dentre as quais, podemos destacar a linguagem natural e o entendimento implícito do contexto. Todavia, quando a interação é da forma humano – computador a comunicação se torna falha, pois o computador não consegue entender as sutilezas da linguagem natural, tampouco o contexto implícito. No entanto, esta situação poderia ser contornada se fosse permitido aos computadores tirarem proveito das informações de contexto, o que tornaria a interação mais rica e possibilitaria que fosse feito um serviço computacional mais útil (Dey, 2001).

A computação sensível ao contexto tem como finalidade estudar o emprego de informações que são utilizadas para caracterizar uma situação de interação entre usuários e computadores, com o intuito de fornecer serviços adaptados aos usuários e aplicações. As informações de contexto podem ser obtidas de duas formas: explícita, o usuário intencionalmente fornece a informação para o sistema; implícita, a informação é obtida sem que o usuário se der conta ou desvie o foco da sua atenção (Bulcão Neto, 2006).

As primeiras definições de contexto não eram claras, em geral, utilizavam-se exemplos para descrever o que seria contexto, como por exemplo: “contexto é a localização e identificação de pessoas e objetos próximos, e as alterações a esses objetos.” Com isto, o processo de identificar se uma determinada informação, não listada na definição seria contexto ou não, se tornava complicado, pois não se podia utilizar a definição para resolver a situação (Dey, 2001).

Ainda nas primeiras definições de contexto, uma forma utilizada para descrever contexto foi prover-lhe sinônimos, descrevendo-o como o ambiente ou uma situação. Segundo Schlit *et al* (1994, *apud* Dey, 2001) “os aspectos importantes do contexto são: onde você está, quem você é e quais recursos estão nas proximidades.”. Pascoe (1998, *apud* Dey, 2001) “define o contexto para o subconjunto de estados físicos e conceituais de interesse para uma determinada entidade”. Todavia, assim como as definições “por exemplo”, as definições que usam sinônimos para descrever contexto são muito difíceis de serem colocadas em prática, por serem muito específicas.

Segundo (Bulcão Neto, 2006) uma definição amplamente aceita na comunidade de computação ciente de contexto é a de informação de contexto como sendo:

> "Contexto é qualquer informação que pode ser usada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar, ou objeto que é considerado relevante para a interação de um usuário e uma aplicação, incluindo o próprio usuário e a aplicação". (Dey, 2001).

As primeiras definições sobre computação sensível ao contexto surgiram em meados da década de 90 e apontavam a sensibilidade ao contexto como sendo um software que se adapta de acordo com o seu local de utilização, a coleção de pessoas próximas e objetos, bem como as alterações a esses objetos ao longo do tempo (Dey, 2001).

Conforme define (Dey, 2001) "Um sistema é sensível ao contexto se este usa contexto para prover informações relevantes e/ou serviços ao usuário, onde a relevância depende da tarefa do usuário".

2.2.2.1 Características do software sensível ao contexto

Pascoe (1998, apud Bulcão Neto, 2006) define uma taxonomia contendo as características essenciais que um software sensível ao contexto deve conter:

Percepção contextual: Consiste em capturar informações contextuais e apresentá-las ao usuário.

Adaptação contextual: Habilidade que permite a aplicação adaptar-se de acordo com o contexto.

Descoberta de recursos contextuais: Localizar e explorar recursos que tenham relevância para o usuário.

Expansão contextual: Associa informações de contexto à situação em que se encontra o usuário.

(Dey, 2000) cria uma categorização, que permite delimitar quais são os aspectos básicos que um software sensível ao contexto deve conter e defende que é possível ter o conhecimento sobre quais tipos de características de contexto devem ser levadas em conta no processo de desenvolvimento de softwares sensíveis ao contexto:

Fornecer informação e serviço ao usuário: Apresentar o contexto como forma de informação para o usuário.

Execução de um serviço para um usuário: O software executa um serviço para o usuário de forma automática de acordo com o contexto do usuário.

Rotular as informações do contexto para uma posterior recuperação: As informações de contexto são rotuladas e podem servir como índices para uma recuperação posterior.

2.2.2.2 Requisitos para a construção de software sensível ao contexto

Segundo (Vieira, 2009) existem muitos desafios ao se projetar uma aplicação sensível ao contexto, dentre os quais se podem destacar: (i) caracterização de elementos contextuais e sua representação em um modelo semântico; (ii) aquisição de informações contextuais através de fontes heterogêneas; (iii) processamento e interpretação das informações adquiridas; (iv) disseminação e compartilhamento da informação contextual obtida entre diferentes aplicações; (v) adaptação da aplicação a variações do contexto observada.

Para (Bulcão Neto, 2006), sistemas sensíveis ao contexto necessitam, para serem desenvolvidos, de um conjunto de requisitos que sirvam como orientação durante o seu processo de desenvolvimento. (Dey, 2000) define um conjunto de requisitos que podem ser utilizados na construção de sistemas sensíveis ao contexto:

Especificação de informações de contexto: Especifica-se os tipos de informações de contexto que uma aplicação pode necessitar. Deve-se utilizar um mecanismo que permita especificar as informações de contexto e uma linguagem que permita as informações de contexto serem representadas como simples ou fragmentadas, se existe uma relação entre estas informações e se são interpretadas ou não.

Captura e acesso: Não há uma forma padrão para adquirir e manipular informações de contexto. Em geral, as informações de contexto são manipuladas de forma ad hoc, sem se preocupar com generalização e reúso. No entanto, deve-se definir como estas informações serão adquiridas (através de sensores, GPS, usuário, etc).

Comunicação distribuída e transparente: Os ambientes ubíquos, cada vez mais, fazem uso de diferentes dispositivos computacionais, como por exemplo, os sensores. Desta forma, a comunicação entre a aplicação e os dispositivos de captura de informações de contexto deve ser feito de forma transparente, para que o desenvolvedor não tenha que se preocupar em implementar uma padrão de comunicação específico para cada dispositivo.

Interpretação de informações de contexto: Dado que podem existir várias camadas envolvidas no processo de aquisição do contexto, essas camadas devem ser transparentes para o desenvolvedor. Para tanto, as informações de contexto são interpretadas antes de serem utilizadas pelas aplicações. A interpretação pode ser feita tanto pela análise dessas informações de forma independente ou como a combinação dessa informação com outras registradas no passado ou no presente.

Execução independente da aplicação: É importante separar os processos de aquisição e de utilização das informações de contexto, de forma que as aplicações possam utilizar essas informações sem se preocuparem com a forma que estas foram adquiridas. Os componentes que executam a captura de informações de contexto devem ser executados de forma independente.

Portanto, o desenvolvimento de um sistema sensível ao contexto deve ser visto sob duas diferentes perspectivas. Uma primeira, dependente de um domínio de aplicação específico (especificação e uso do contexto) e uma segunda independente de domínio ou aplicação (gerenciamento do contexto). Diferentes domínios ou aplicações demandam por diferentes informações de contexto e tratamento para ser utilizado por um sistema. Diferentemente, o gerenciamento pode ser feito de forma modularizada e tratado independentemente, visto que fornece as funcionalidades de adquirir, processar, disseminar e armazenar informações de contexto (Vieira, 2009).

2.3 Modelo de desenvolvimento orientado a tarefas

Atualmente temos milhares de serviços web disponíveis que fornecem interfaces programáveis para dispositivos computacionais, aparelhos, mecanismos de buscas e objetos diários. Porém, existe uma a lacuna entre os serviços disponíveis e as tarefas que o usuário que os usuários desejam realizar com estes serviços. A tarefa é uma função ou objetivo, enquanto um serviço é uma execução desta tarefa. As tarefas estão associadas com o que o usuário quer realizar e os serviços com a capacidade do ambiente para completar as tarefas (Loke, 2009).

A computação baseada em tarefas é uma tecnologia que foi projetada tendo como finalidade preencher a lacuna existente entre aquilo que o usuário quer fazer e o que os serviços oferecem. Podemos ver a computação orientada a tarefas como um framework, que permite aos usuários finais acompanharem complexas tarefas em ambientes ricos em aplicações, dispositivos computacionais e serviços (Masuoka et al, 2005).

A computação orientada a tarefa possibilita várias formas de interação com o ambiente. Seus usuários podem acompanhar facilmente uma variedade de tarefas usando diferentes fontes, tais como: o computador do usuário ou PDA, dispositivos computacionais dispersos no ambiente e serviços web. Além disso, a interação do usuário é feita através de uma interface baseada sobre a linguagem humana e a experiência é como uma sentença em linguagem natural do tipo: “Ligar a Tv”, “Acender a luz”, “Ligar o projetor”. O usuário deve dizer ao sistema para executar a sentença (Masuoka et al, 2005).

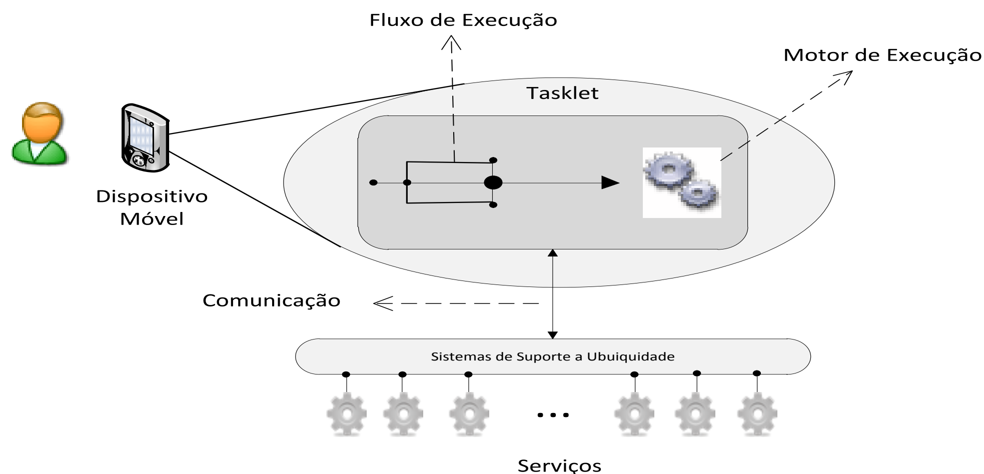

Figura 3 – Computação Orientada a Tarefas

A Figura 3 apresenta o modelo de computação orientada a tarefa. Temos um usuário que deseja executar uma atividade através de um dispositivo móvel. A atividade é mapeada em uma Tasklet (i.e, aplicação de suporte automatizado à tarefa), que possui um fluxo de execução, em que é representada uma sequência de interações com os serviços existentes no ambiente, através de um sistema de suporte a ubiquidade, que pode ser visto como uma camada de middleware que abstrai questões de projeto de software ubíquo tais como, mobilidade, distribuição e sensibilidade ao contexto, permitindo que o desenvolvedor foque sua atenção e esforços somente nos requisitos funcionais do software a ser construído.

No modelo de computação orientado a tarefas, o usuário sempre é o foco principal. O usuário tem autonomia e participação no sistema, que por sua vez, provê o usuário com um alto grau de abstração da tarefa do ambiente para uma fácil manipulação destas pelos próprios usuários. Além disso, o usuário é que detém autoridade sobre o sistema, podendo decidir quando e o que vai ser executado (Masuoka et al, 2005).

A computação orientada a tarefas usa sistema baseado em serviços, que consiste em um mapeamento entre tarefas e serviços. Porém, este mapeamento é complexo de ser feito, devido ao problema de representar a tarefa e o serviço, e o link entre os dois. O usuário não lida diretamente com os serviços, mas preferencialmente especificam tarefas de alto nível para o sistema, que lida com os serviços de baixo nível (Loke, 2009).

Segundo (Loke, 2009), Os desenvolvedores devem considerar seis aspectos quando projetam um software para computação orientada a tarefas:

Formalismo: O sistema pode representar tarefas em dois modos - usando frases sem um formalismo rico e usando programas explícitos - como representação de tarefas compostas.

Frases de Abstração: O usuário especifica uma tarefa (o que), sem ter que se preocupar com detalhes de encontrar, configurar ou invocar aplicações (o como). Exemplos: executar um vídeo, exibir uma apresentação em slides, enviar uma mensagem para um usuário, etc.

Modelo Mental: O propósito de empregar abstração de tarefas é ajudar usuários a interagir com o sistema para pensar em termos de tarefas que eles querem fazer melhor do que as aplicações disponíveis. O que requer um modelo mental simples e intuitivo para abstração de tarefas. O sistema captura tarefas comuns em lugares particulares com artefatos particulares. Usuários adaptam um modelo mental simples para as aplicações comuns entenderem comandos em dispositivos, aparelhos e objetos localizados no espaço que o usuário se encontrar, por exemplo, uma sala de visita.

Mapear tarefas para serviços: Dada a camada subjacente de serviços sobre as quais se abstraem as tarefas, existe um mecanismo para mapear tarefas especificadas por usuários para os serviços. O sistema realiza o mapeamento dessas tarefas para os serviços subjacentes ao solicitar que o usuário forneça uma tarefa específica, então é aplicado um elaborado mecanismo de busca para descobrir os serviços relevantes, compô-los e invocá-los para executar a tarefa. A invocação de serviços pode tomar como parâmetro o contexto do usuário para já identificar o subconjunto dos serviços que atendem aquela informação de contexto.

Execução e Gerenciamento de Tarefas: Na maioria dos sistemas, o usuário especifica a tarefa e o sistema determina como executá-la. Porém, um sistema pode também deixar o usuário especificar o suporte de preferência para as tarefas, tais como a qualidade de tolerância do serviço e software preferido, que o sistema, então, usa na execução da tarefa.

Atualmente, os sistemas tipicamente não informam o progresso da tarefa para o usuário, não obstante, seja um fator importante para usabilidade. É um desafio saber o que apresentar ao usuário no caso da tarefa falhar, porque usuários não técnicos podem não ser aptos a tomar ações corretivas.

Identificar as tarefas que um sistema deve suportar: Quando criamos um sistema para suporte a tarefas na sala de estar de uma pequena casa, por exemplo, o desenvolvedor deve determinar as tarefas típicas que o sistema deve suportar. As tarefas podem ser aplicações específicas, mas dependerão do que o usuário faz em uma sala de estar, bem como os artefatos (aparelhos e dispositivos) presentes. Idealmente, deve ser um conjunto mínimo de tarefas que um lugar suporta, e usuários devem ser aptos a executar essas tarefas em um lugar suportado por um sistema computacional baseado em tarefas.

Pluralidade de Expressão de Tarefas: Um sistema associado com uma ou mais tarefa pode expressar tarefas emitidas no contexto de um lugar, que resolve uma tarefa especificada pelo usuário dentro de uma coleção de invocação de serviços.

3 GUIDELINE PARA O DESENVOLVIMENTO DE SOFTWARE UBÍQUO ORIENTADO A TAREFAS

No levantamento bibliográfico pode-se perceber que a comunidade acadêmica está se movimentando para definir um processo a ser seguido no desenvolvimento de software ubíquo. Contudo, apesar do esforço considerável de definir um conjunto de atividades em cada etapa de desenvolvimento que contemplem os requisitos para o desenvolvimento de software ubíquo, as propostas existentes tendem a desviar o desenvolvedor dos objetivos do software que é a automatização das atividades cotidianas dos usuários. Diante disso, nesse trabalho é proposto um *guideline* que tenta apontar caminhos para gerenciar a complexidade do desenvolvimento de software ubíquo, ao permitir que o desenvolvedor foque sua atenção e esforços nas tarefas do usuário e nos seus requisitos funcionais.

O *guideline* consta de uma sequência de passos que deve ser seguida para o desenvolvimento de software ubíquo usando o modelo de desenvolvimento orientado a tarefas. Cada passo constitui uma etapa do processo de desenvolvimento que deve ser seguido. Não são levados em consideração aspectos de custos de desenvolvimento, metodologias e processos específicos. O *guideline* pode ser adaptado para quaisquer modelos de desenvolvimento de software, desde que siga as recomendações apontadas.

Neste *guideline* utiliza-se o desenvolvimento **com** reuso e **para** reuso. Reúso de software é o processo de criar sistemas de software a partir de software existente, ao invés de construí-los a partir do zero (Krueger, 1992). O desenvolvimento com reúso consiste em desenvolver o software reutilizando artefatos existentes, enquanto o desenvolvimento para reúso consiste em desenvolver artefatos de software que possam ser reutilizados posteriormente no desenvolvimento de um novo software (Almeida *et al* (2010)).

No desenvolvimento **com** reúso, levamos em conta o desenvolvedor que vai desenvolver um software ubíquo e conta com uma infraestrutura de serviços já definidas, assim, ele não precisará mais se preocupar com isso e pode desenvolver seu software reutilizando todos os serviços disponíveis. Por outro lado, no desenvolvimento **para** reúso, o desenvolvedor não conta com uma infraestrutura de serviço definida, então, ele deve considerar isto ao projetar o software, pois estes serviços também devem ser desenvolvidos junto ao software e depois podem ser disponibilizados para reúso.

O guideline foi validado por um painel de especialistas realizado no GREat (Grupo de Pesquisa em Redes de Computadores, Engenharia de Software e Sistema) da Universidade

Federal do Ceará que forneceram informações valiosas sobre quais atividades deveriam ser consideradas, de modo que tornasse o desenvolvimento o mais simples possível e que fosse de fácil compreensão. Além disso, foram fornecidos relatos de desenvolvimento de software ubíquo, que tanto ajudaram a desenvolver o guideline como a aplicação de exemplo.

3.2 Fundamentação do Guideline

A partir de um levantamento bibliográfico realizado por (Rocha, 2011) podemos definir, baseado nas características de software ubíquo, um conjunto de requisitos considerados importantes para o desenvolvimento de software ubíquo.

Tabela 2 – Requisitos do Software Ubíquo

Requisitos	Descrição
Descentralização	Responsabilidades distribuídas entre vários dispositivos presentes no ambiente.
Diversidade e Heterogeneidade	Dispositivos com propósitos específicos e heterogêneos.
Conectividade e Interoperabilidade	Dispositivos movem-se junto com o usuário de forma transparente, através de várias e heterogêneas redes sem fio.
Mobilidade	Mobilidade de Usuário: O usuário pode mudar de dispositivo de acesso mantendo sua identificação e estado de interação; Mobilidade de Dispositivo: Mobilidade física dos dispositivos; Mobilidade de Código: Migração do estado de execução, dos dados e do próprio código executável do software entre os dispositivos computacionais presentes no ambiente.
Sensibilidade ao Contexto	Capacidade do dispositivo perceber informações contextuais e adaptar seu comportamento e reagindo adequadamente modificando sua estrutura e seu comportamento através da execução de estratégias de adaptação apropriadas.
Adaptabilidade	O sistema reage de maneira apropriada às mudanças no seu ambiente de execução.
Autonomia	O sistema deve ser capaz de se auto gerenciar.
Interoperação Espontânea	Os elementos (hardware e software) que compõem o sistema precisam ser projetados para estarem aptos a coordenar suas atividades com outros elementos previamente desconhecidos.
Invisibilidade	Os dispositivos computacionais devem possibilitar as pessoas sua utilização de maneira transparente e casual. A invisibilidade pode ser tanto de natureza física (relacionada com o grau

	de miniaturização e embarcamento dos dispositivos computacionais no ambiente), como de natureza mental (a forma como os usuários interagem e percebem os computadores no seu ambiente).
Segurança e Privacidade	O sistema deve prover segurança e a privacidade dos usuários para que os recursos do ambiente não sejam usados indevidamente e as mensagens sejam capturadas por pessoas não autorizadas.

A Tabela 2 nos permite visualizar a grande quantidade de requisitos não funcionais que um software ubíquo deve atender. Na maioria das vezes, estes requisitos não vem sendo abordados de uma forma sistemática. O que acaba gerando um problema no desenvolvimento do software, pois alguns requisitos, como a sensibilidade ao contexto e adaptação, influenciam todo o processo de desenvolvimento.

A sensibilidade ao contexto e adaptação precisam ser incorporadas ao processo de desenvolvimento. Podemos ver nos trabalhos de (Bulcão Neto, 2006), (Vieira, 2009), (Choi 2008) e (Henricksen; Indulska,2006) formas diferentes de tratar a sensibilidade ao contexto e adaptação. Entretanto, todos estes trabalhos têm atividades específicas em comum, dentro das atividades macro do processo de desenvolvimento de software convencional, que tratam de forma sistemática estes dois requisitos.

Outra forma para lidar com estes requisitos é utilizando o modelo de desenvolvimento orientado a tarefas. Este modelo possibilita definir uma forma de estruturar e modularizar o processo de desenvolvimento. O que ajuda a gerenciar a complexidade do desenvolvimento do software ubíquo.

Diante disso, foi considerado para formar a base do guideline o modelo de desenvolvimento orientado a tarefas e os requisitos de software ubíquo, sobretudo, a sensibilidade ao contexto e adaptação, que influenciam diretamente no processo de desenvolvimento.

Utilizamos essa base para propor um painel de especialistas que foi realizado no GREat/UFC. Foram apresentadas as principais características que um software ubíquo deve conter, bem como as principais características do software sensível ao contexto, tudo sendo visto sob o ponto de vista do modelo de desenvolvimento orientado a tarefa.

O painel de especialistas nos permitiu constatar que a complexidade de desenvolver software ubíquo é muito grande. Os especialistas deram muitos exemplos de situações excepcionais que podem ocorrer no desenvolvimento. Recomendaram propor algo que abstraísse o máximo de complexidade possível. Uma abordagem que focasse somente na tarefa do usuário e considerasse as atividades de análise e projeto.

Dentre as recomendações dos especialistas que serviram para auxiliar o desenvolvimento do *guideline*, podemos destacar um conjunto de sugestões que foram feitas:

Identificar as Atividades do Usuário: Discutiu-se sobre quais atividades do usuário seriam consideradas em um domínio específico. Foram apresentados exemplos das atividades que um usuário poderia realizar em uma sala de aula.

Mapear Atividades em Tarefas: Considerar os recursos computacionais do ambiente para saber se uma atividade pode ser mapeada em uma tarefa.

Identificar Recursos Computacionais Existentes: A forma que o desenvolvedor faria o levantamento dos recursos computacionais no ambiente. Iniciou-se uma discussão neste tópico sobre a importância de ser feito um sistema que desse suporte automatizado a esta atividade.

Identificar o Fluxo de Execução da Tarefa: Identificar os passos necessários para a execução da tarefa.

Identificar o que Dispara as Tarefas: Identificar quais informações de contexto disparam uma tarefa.

Identificar Contexto de Adaptação: Identificar quais informações de contexto funcionam como gatilhos para o sistema se auto adaptar.

Identificar Fluxos Alternativos: Identificar a sequência de passos que o sistema deve seguir ao ser verificado uma informação de contexto que faz o sistema se auto adaptar.

O painel de especialista foi fundamental para ajudar a fechar o escopo do trabalho e a definir as atividades que iriam ser consideradas. A partir das recomendações foi feito um diagrama que permite visualizar como estas atividades estão dispostas no *guideline* e o relacionamento entre elas. Foi utilizado o diagrama de atividades da UML, que nos permite observar o fluxo de controle de uma atividade para outra.

O *guideline* foi enviado para os especialistas que puderam constatar que suas observações e sugestões foram incorporadas, bem como sugeriram algumas mudanças no

texto e nas figuras. Recomendaram que fosse especificado no texto que este *guideline* não leva em consideração aspectos de custos de desenvolvimento, metodologias e processos específicos. Ressaltar que o *guideline* pode ser adaptado para os modelos de desenvolvimento de software, mas seguindo as recomendações apontadas.

3.3 Guideline

Visão Geral: Definimos nesta etapa uma estrutura geral do guideline, mostrada na Figura 4, ao propormos uma fase de análise de atividades e projeto de tarefas. As atividades referem-se ao que os usuários fazem no dia a dia e as tarefas são as descrições prescritivas dessas atividades. O desenvolvedor especificará na fase de análise as atividades a serem consideradas e na fase de projeto ele fará o mapeamento dessas atividades em tarefas. Ao final da etapa de análise e projeto para cada atividade o desenvolvedor verificará se ainda existem atividades a serem consideradas, em caso afirmativo, ele passará para a análise e projeto de outra atividade, caso contrário, irá para uma nova etapa de desenvolvimento.

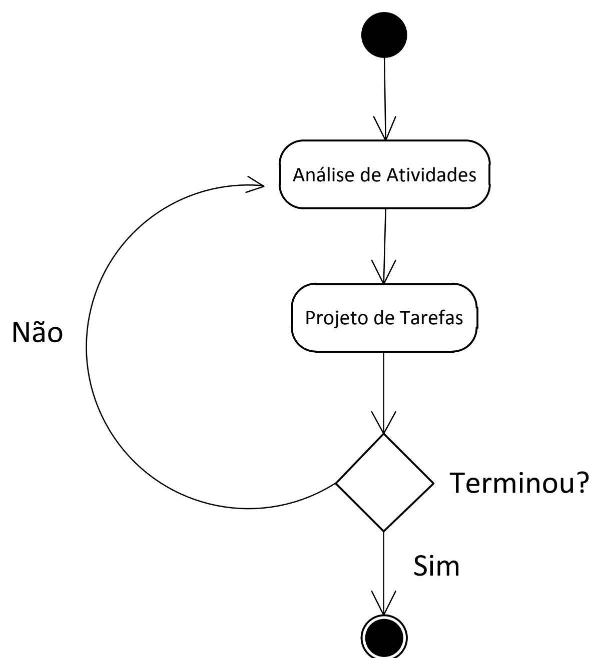

Figura 4 – Estrutura Geral do Guideline

1 – Análise de Atividades: Identificação e levantamento das Atividades do Usuário. O desenvolvedor deverá fazer um levantamento das atividades comuns relacionadas a um domínio de interesse específico que um usuário pode realizar. As atividades representam àquilo que o usuário pretende fazer em um domínio específico. Exemplo: No domínio de um shopping um usuário pode estacionar um carro, assistir um filme, fazer compras, etc.

2 – Projeto de Tarefas: Mapeamento das atividades do usuário em tarefas que podem ser automatizadas. As atividades que não podem ser automatizadas não são consideradas. Utiliza-se uma linguagem de especificação para decompor a tarefa em uma sequência de passos, onde cada passo representa uma tarefa primitiva. Uma tarefa primitiva é uma unidade de ação que pode ser executada por um serviço ou mais serviços.

3 – Terminou?: Verifica se ainda existem mais atividades para serem analisadas. No caso afirmativo, deve-se ir para a etapa de projeto, caso contrário, continua na etapa de análise de atividades.

Detalhamento da Análise: Nesta etapa será feito o detalhamento das atividades levantadas na fase de análise de atividades. O desenvolvedor deverá especificar as atividades em mais detalhes para identificar se a atividade pode ser mapeada em uma tarefa.

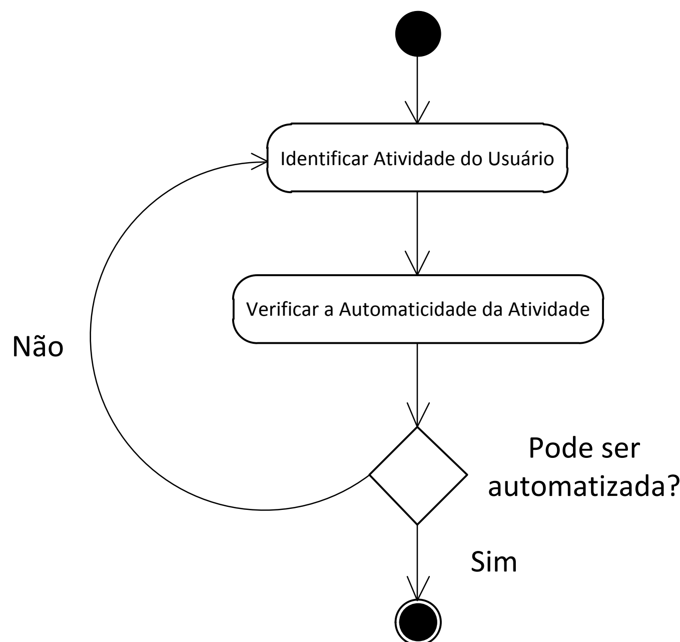

Figura 5 – Análise de Atividades

1 – Identificar Atividades do Usuário: Identificar as atividades que o usuário deseja realizar. O desenvolvedor deve fazer um levantamento das atividades que um usuário pode realizar ao estar em um determinado ambiente. O desenvolvedor pode partir das atividades comuns que são realizadas no ambiente considerado e, se for o acaso, vai-se acrescentando novas atividades que sejam necessárias.

2 – Verificar a Automaticidade da Atividade: Verifica se a atividade pode ser automatizada em uma tarefa. O desenvolvedor deve considerar as características da atividade

e verificar se existem no ambiente recursos computacionais que possam dar suporte para a realização dessa atividade. Neste passo, o desenvolvedor deve considerar se todos os passos para a realização da atividade podem ter suporte automatizado no ambiente.

3 – Pode ser automatizada?: No caso da atividade ser automatizável, deve-se passar para a fase de projeto da tarefa, caso contrário deve-se passar para uma próxima atividade.

Detalhamento do Projeto: Nesta fase o desenvolvedor projetará a Tasklet, que é o nome que adotamos neste trabalho de uma atividade que pode ser automatizada. Definiremos todos os aspectos que estão relacionados com sua execução e que satisfazem os requisitos do software ubíquo. Baseamos esta etapa nas características do modelo de desenvolvimento orientado a tarefas exposto na Figura 03 e no desenvolvimento com reuso e para reuso.

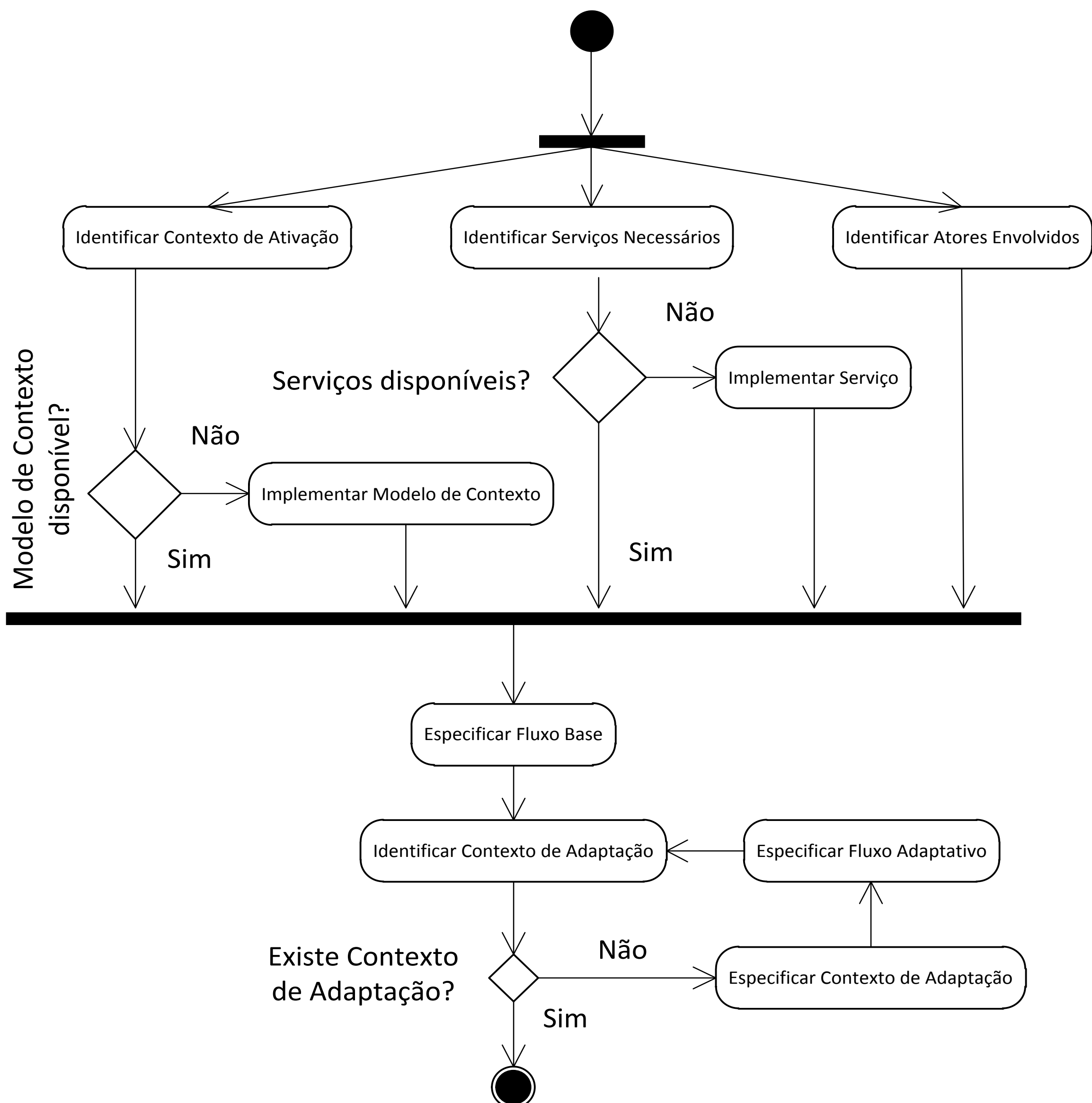

Figura 6 – Projeto de Tarefas

1 – Identificar Contexto de Ativação: Identificar o contexto necessário para caracterizar que o usuário está envolvido em uma atividade. O desenvolvedor deve elencar as informações contextuais que caracterizam a instanciação da tarefa. Neste caso a tarefa pode ser recomendada ao usuário e/ou iniciada automaticamente pelo sistema.

2 – Identificar Serviços Necessários: Identificar os serviços necessários para a execução da tarefa e que sejam essenciais para o conjunto das tarefas primitivas. O desenvolvedor deve identificar todos os serviços necessários para a execução de uma tarefa. Para isto a tarefa deve ser quebrada em conjunto de tarefas primitivas, que representam a unidade mínima de ação que pode ser suportada por um ou mais serviços. Os serviços representam alguma funcionalidade do negócio, em que a funcionalidade depende do domínio do software. O desenvolvedor elencará todos os serviços necessários para que a tarefa seja executada.

3 – Identificar Atores Envolvidos: Identificar os atores envolvidos na tarefa. Os atores representam as entidades que irão interagir com o sistema e que influenciam seu comportamento. O desenvolvedor deve elencar os usuários e os recursos computacionais que irão interagir no ambiente no momento de execução da tarefa.

4 – Modelo de Contexto Disponível?: Verifica se existe um modelo contextual com as informações necessárias para a execução da tarefa. O desenvolvedor irá verificar se já tem disponível um modelo que represente as informações contextuais necessárias para a execução da tarefa. O modelo de contexto representa as informações em um domínio ou aplicação e como essas informações se relacionam com o comportamento do sistema.

5 – Serviços Disponíveis?: Verificar no ambiente quais são os serviços disponíveis necessários para a realização da tarefa. O desenvolvedor deve nesse passo fazer um levantamento dos recursos computacionais existentes e verificar se estes são suficientes para que a tarefa seja executada. No caso afirmativo o desenvolvedor apenas reutiliza o serviço, caso contrário, é necessário o desenvolvimento do serviço.

6 – Implementar Modelo de Contexto: Especificar e projetar as informações de contexto necessárias para a execução da tarefa em um modelo contextual. O desenvolvedor deve fazer a modelagem das informações contextuais necessárias para a execução da tarefa. Nesta etapa o desenvolvedor irá considerar como as informações contextuais se relacionam entre si e com o software.

7 – Implementar Serviço: Desenvolver os serviços necessários para a execução da tarefa. Nesta etapa o desenvolvedor deverá desenvolver os serviços necessários para a tarefa, onde cada serviço representa uma funcionalidade abstrata do domínio da aplicação. As funcionalidades dos serviços deveram ser disponibilizadas por meio da sua interface. Além disso, dependendo da natureza e complexidade da tarefa que o serviço venha a atender, o desenvolvedor pode fazer uma composição de serviço, e atribuir para cada serviço uma tarefa menor.

8 – Especificar Fluxo Base: Especifica o fluxo de execução da tarefa. Deve ser especificada uma sequência de interações da tarefa com os serviços existentes. Neste passo o desenvolvedor divide a tarefa em um conjunto de passos que devem ser seguidos para que a tarefa seja executada. A tarefa é dividida em tarefas primitivas, que são a unidade mínima de ação que pode ser suportada por recursos computacionais.

9 – Identificar Contexto de Adaptação: Identificar quais informações de contexto funcionam como gatilhos para o software se auto adaptar. O desenvolvedor deve especificar quais informações de contexto influenciam o comportamento do software e fazem-no se adaptar para atender a tarefa do usuário.

10 – Existe Contexto de Adaptação?: Verifica se existe um modelo contextual com as informações de contexto necessárias para a adaptação do software. O desenvolvedor irá verificar se já tem disponível um modelo que represente as informações contextuais necessárias para o software se auto adaptar.

11 – Especificar Contexto de Adaptação: Especificar os diferentes tipos de contexto que influenciam no fluxo base da tarefa. O desenvolvedor fará o levantamento das informações contextuais que influenciam o comportamento do software, sobretudo, as excepcionais, que devem ser mapeadas pelo desenvolvedor nesta etapa.

12 – Especificar Fluxo Adaptativo: Especificar os fluxos alternativos que a tarefa pode assumir ao obter informações do contexto de adaptação. O desenvolvedor deve nesta etapa especificar quais ações o software deve tomar ao serem observadas variações no ambiente de execução. O desenvolvedor deve criar um fluxo alternativo para cada informação contextual. O fluxo alternativo representa uma sequência de passos que o software deve seguir ao ser observado alguma informação contextual que leve o software a uma situação particular e exigem sua adaptação.

4 APLICAÇÃO EXEMPLO

No mundo real, as pessoas realizam diversas atividades todos os dias. As atividades podem ser as mais variadas possíveis e em diferentes domínios. Vivemos em uma sociedade frenética, onde as coisas sempre acontecem de forma muito rápida e simultaneamente.

Uma situação típica que acontece rotineiramente com as pessoas é a atividade de estacionar um carro. Todo mundo que tem carro passa por isso várias vezes ao dia. O condutor/veículo ao adentrar em um estacionamento, procura por uma vaga para estacionar seu carro, e preferencialmente, uma vaga que esteja mais próxima de onde ele quer ficar. Percebemos que esta atividade pode tomar um tempo considerável do usuário e como a computação ubíqua visa auxiliar as pessoas nas suas atividades diárias, podemos pensar em um estacionamento ubíquo que auxilie o usuário a estacionar um carro, de forma que seja gasto o menor tempo possível.

Diante disto, consideraremos para nossa aplicação de exemplo este estacionamento ubíquo, que tem como atividade prover ao usuário suporte automatizado para ajudá-lo a estacionar um carro.

Iremos mostrar, usando o *guideline*, como um desenvolvedor pode desenvolver um software ubíquo que auxilie condutor/veículo a estacionar, abstraindo os detalhes de baixo nível e focando-se apenas na tarefa do usuário e nos requisitos funcionais que o software deve atender.

Análise de Atividades : Levantamento das Atividades do Usuário

1 – Análise das Atividades: Um estacionamento ubíquo tem como atividade principal estacionar o carro. A atividade consiste em um condutor/veículo achar uma vaga livre no estacionamento para guardar o carro.

2 – Projeto de Tarefas: A atividade do usuário representada pela atividade estacionar um carro, será mapeada no ambiente do estacionamento ubíquo, em uma tarefa que será chamada de ParkingTask.

Detalhamento da Análise: Análise das atividades

1 – Identificar as Atividades do usuário: Identificamos que a atividade que um condutor/veículo pode realizar em um estacionamento ubíquo é estacionar o carro. Utilizando o diagrama de casos de uso da UML fazemos a representação da atividade estacionar.

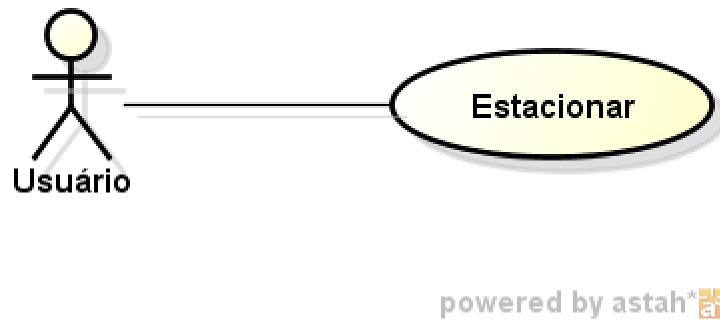

Figura 7 – Estacionar

2 - Verificar a automaticidade da tarefa: A atividade estacionar um carro pode ser automatizada, pois tem como ser projetado uma infraestrura que possa dar apoio a esta atividade. O estacionamento pode ser equipado com sensores de localização que podem informar a posição do usuário, agentes computacionais dispersos no ambiente que podem informar as vagas livres e serviços web que podem oferecer serviços de suporte ao negócio, como por exemplo, fazer a bilhetagem.

Detalhamento do Projeto: Projeto da Tarefa

1 – Identificar Contexto de Ativação: O software ubíquo que fornece suporte a atividade de estacionar um carro, precisa saber antes de iniciar sua execução, se o usuário se encontra nas dependências do estacionamento. Na Tabela 3 fornecemos as informações de contexto que indicam que a tarefa pode ser iniciada.

Tabela 3 – Contexto de Ativação

Contexto	Descrição
Localização do Veículo/Condutor	Utilizada para informar ao sistema que o usuário se encontra nas dependências do estacionamento.

2 – Identificar Serviços necessários: Considerando as características da tarefa, constatamos que um estacionamento ubíquo que provê suporte automatizado para a realização da tarefa estacionar um carro, necessita dos serviços mostrados na Tabela 4.

Tabela 4 – Lista de Serviços Necessários no Estacionamento Ubíquo

Serviços	Descrição
IDService	Serviço que provê a identificação do veículo
FreeSpaceService	Serviço para a indicação das vagas livres no estacionamento
TicketingService	Serviço para realizar a bilhetagem do estacionamento

3 - Identificar Atores Envolvidos: Os atores envolvidos no sistema podem ser visualizados na Tabela 5.

Tabela 5 - Atores

Ator	Descrição
Condutor/Veículo	Usuário que deseja estacionar
Agentes Computacionais	Sensores dispersos no estacionamento que indicam as vagas livres.
Outros Condutor/Veículo	Outros usuários que estão simultaneamente querendo estacionar.

4 – Verificar disponibilidade de Modelo de Contexto: Analisando o contexto necessário para atividade estacionar um carro chegamos neste modelo de contexto. Utilizamos o diagrama de classes da UML para identificarmos as entidades e contexto envolvidos nesta atividade, mostrando como eles se relacionam entre si. Utilizamos como base para a modelagem o perfil da UML proposto por (Vieira, 2009).

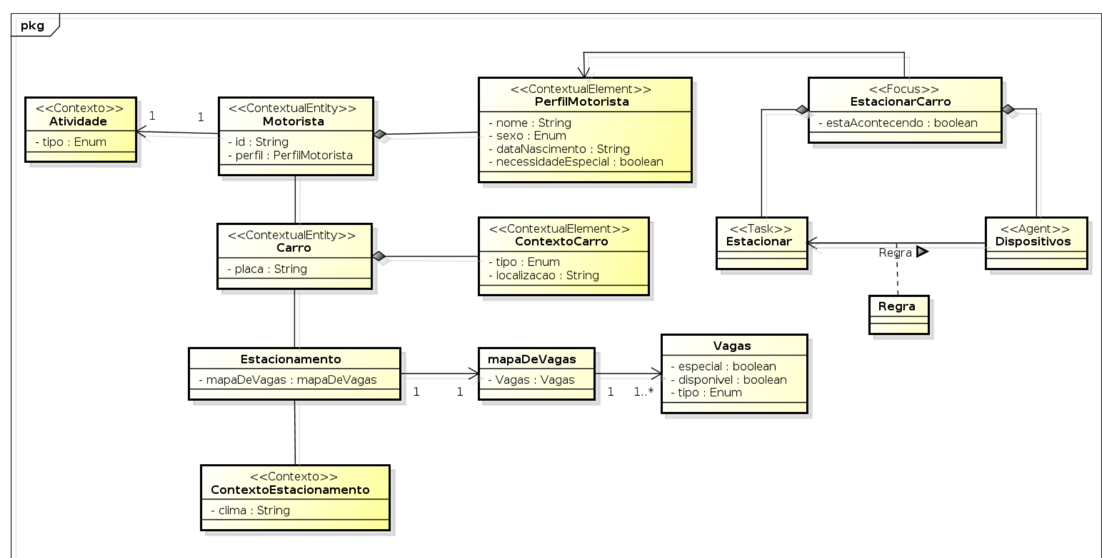

Figura 8 – Modelo de Contexto

5 – Implementar Modelo de Contexto: Para implementar o modelo de contexto foi preciso fazer uma análise dos elementos contextuais e entidades envolvidos nesta atividade. Criamos um diagrama de classes da UML que mostra os relacionamentos entre as entidades e os elementos contextuais.

6 – Implementar Serviço: Consideramos nesta atividade o desenvolvimento com reúso, portanto, partimos do pressuposto que já existe os serviços disponíveis.

7 – Especificar Fluxo Base:

Tabela 6 – Fluxo Base

Tarefas primitivas	Descrição
Realizar a identificação do condutor/veículo	O serviço IDService faz a identificação do veículo.
Sugerir uma trajetória até uma vaga livre	O serviço FreeSpaceService indica uma vaga livre no estacionamento
Iniciar a bilhetagem após o veículo ter sido efetivamente estacionado	O serviço TicketingService faz a bilhetagem

8 - Identificar Contexto de Adaptação:

Tabela 7 – Contexto de Adaptação

Contexto	Descrição
Estacionamento é pago ou gratuito	Verifica se o estacionamento é pago ou gratuito.
Localização do condutor/veículo	Indicar a vaga mais próxima da localização que se encontra o veículo/conduto
Vagas ocupadas	Indicar as vagas que não estão ocupadas ou que acabaram de ser liberadas
Outros veículos que estão simultaneamente querendo estacionar	Outros condutores que estejam querendo estacionar simultaneamente podem ocupar a vaga destinada a outro condutor.

10 – Especificar Fluxo Adaptativo:

Tabela 8 – Fluxo Adaptativo

Contexto	Fluxo Adaptativo
Outros veículos que estão simultaneamente querendo estacionar	O veículo ao se dirigir para a vaga livre indicada, outro condutor ocupa aquela vaga. A Tasklet

	deve indicar uma nova vaga livre observando sua localização
Estacionamento é pago ou gratuito	O condutor ao adentrar em um estacionamento gratuito o serviço de bilhetagem não deve ser cobrado dele.

5 CONSIDERAÇÕES FINAIS

Nossa tentativa de propor um guideline para o desenvolvimento de software ubíquo partiu da necessidade de uma forma de sistematização para o desenvolvimento deste tipo de software, que, de certo modo, vem sendo deixada um pouco de lado, conforme podemos observar nos trabalhos listados no nosso referencial bibliográfico. A maioria dos pesquisadores e profissionais da área costuma se preocupar apenas com a parte técnica do software ubíquo, deixando o desenvolvimento ser feito de uma forma pouco sistemática e produtiva.

A princípio, ao considerarmos desenvolver este trabalho, pensamos em algo maior, iríamos propor um processo de desenvolvimento para software ubíquo. No entanto, percebemos que isto seria um trabalho que demandaria uma maior pesquisa exigindo uma maior quantidade de tempo, o que não seria viável para a conclusão deste trabalho.

Diante disso, ao invés de considerarmos o processo de desenvolvimento como um todo, consideramos apenas as atividades de análise e projeto. Esta escolha foi motivada tanto pela banca que qualificou o projeto deste trabalho no ano de 2010.2, como pelo painel de especialistas realizado no GREat/UFC.

O painel de especialista contribuiu bastante para a realização deste trabalho. No dia do encontro tivemos uma calorosa discussão sobre os aspectos que devem ser considerados no desenvolvimento de software ubíquo, tivemos relatos de desenvolvimento, sugestões de atividades que deveriam ser incluídas no nosso guideline e muitas considerações sobre o modelo de desenvolvimento orientado a tarefas.

O *guideline* tentou sintetizar as características que observamos no referencial bibliográfico, as quais um software ubíquo deve possuir, e as discussões realizadas no painel de especialistas. Para acomodar tudo isso, o modelo de desenvolvimento orientado a tarefas foi adotado para abstrair determinados aspectos do desenvolvimento de software ubíquo.

A abstração que propomos nas atividades do *guideline* tem como intuito tentar minimizar a complexidade no desenvolvimento de software ubíquo. Queremos que o desenvolvedor foque somente nos requisitos funcionais do software e na atividade do usuário, para que assim, consiga-se projetar um software que atenda as características da ubiquidade.

O guideline apresenta uma abstração dos passos que devem ser considerados para desenvolver um software ubíquo, o que permite sistematizar o desenvolvimento, definir quais etapas serão executadas e as atividades mais importantes a serem feitas.

Podemos constatar neste trabalho, ao modelarmos uma aplicação que oferece suporte automatizado a atividade de estacionar um carro em um estacionamento ubíquo, que é possível projetar um software ubíquo, sem ter que nos preocuparmos com detalhes de baixo nível. Ao usarmos o modelo de desenvolvimento orientado a tarefas permitimos que o desenvolvedor focasse apenas no desenvolvimento do software.

Portanto, esperamos com este trabalho ter contribuído com um passo rumo a um processo de desenvolvimento para software ubíquo. Esperamos que os desenvolvedores que projetam software ubíquo, possam se beneficiar do nosso trabalho e utilizar o nosso guideline como uma forma de sistematização no desenvolvimento do software, de forma que possibilite ao desenvolvedor ganhos em qualidade, tempo e recursos

REFERÊNCIAS

ALMEIDA, Eduardo Santana de. *et al*. **C.R.U.I.S.E:** Component Reuse in Software Engineering. Recife: Cesar E-books, 2010.BEZERRA, E. **Princípio de Análise e Projeto de Sistemas com UML.** 2.ed. Rio de Janeiro: CAMPUS, 2007.

BULCÃO NETO, R.F. **Um processo de software e um modelo ontológico para apoio ao desenvolvimento de aplicações sensíveis ao contexto.** 2006. Tese (Doutorado em Ciência da Computação) - Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, São Carlos, 2006.

BULCÃO NETO, R.F., Kudo, T.N., and Pimentel, M.G.C. (2006). **POCAp: A software process for context-aware computing**. Prc. Of the IEEE/WIC/ACM International Conference on Inteligent Agent Technology, Hong Kong, China, 2006.

STADISH GROUP. The CHAOS Report(2010). Documento disponível em http://www1.standishgroup.com/newsroom/chaos_2009.php. Acessado no dia 05/05/2011.

CHOI, Jongmyung., MOON, Hyun-Joo. **Software Engineering Issues in Developing a Context – aware Exhibition Guide System.** Software Engineering , Artificial Inteligence, Networking and Parallel and Distributed Computing, Phuket, 2008.

DEY, A.K. (2000). **Providing architectural support for building context-aware applications.** PhD thesis, College of Computing, Georgia Institute of Technology.

DEY, A.k. (2001). **Understanding and using context.** Personal and Ubiquitous Computing, 5(1):4-7.

COUTAZ, J. et al. **Context is Key.** Communications of the ACM. March 2005/vol. 48. No. 3.

HENRICKSEN, K., INDULSKA, J. (2006**). Developing Context-Aware Pervasive Computing Applications: Models and Approach, Pervasive and Mobile Computing Journal**, v. 2, n. 1, pp. 37-64.

KRUEGER, C.W. **Software Reuse**, In: ACM Computing Surveys, Vol. 24, No. 02, June, 1992, pp. 131-183.

LOCKE, W., SANG. **Building Taskable Spaces Over Ubiquitous Service**. IEEE Pervasive Computing. 2009.

MASUOKA, R. et al. **On Building Task Computing.** Agent Link News. n. 18, August, 2005.

SOMMERVILE, I. **Engenharia de Software.** 8.ed. São Paulo: Addison Wesley, 2007.

SOUSA, J. P et al. **Task-Based Adaptation for Ubiquitous Computing.** IEEE Transactions on Systems, Man, and Cybernetics—Part c: Applications and Reviews, vol. 36, n. 3, MAY, 2006

ROCHA, L. S. AdaptiveRME AspectCompose: **Um Middleware Adaptativo e um Processo de Composição Orientado a Aspectos para o Desenvolvimento de Software Móvel e Ubíquo.** Dissertação (Mestrado em Ciência da Computação) – Departamento de Computação, Universidade Federal do Ceará, Fortaleza, 2007.

VIEIRA, V., TEDESCO, Patricia., SALGADO, Ana Carolina. **A process for the Design of Context-Sensitive Systems.** 13th Conference on Computer Supported Cooperative Work in Design, Chile, 2009.

WANG, ZHENYU, GARLAN, DAVID. **Task-Driven Computing.** Technical Report CMU-CS-00-154. Department of Computer Science, Carnegie Mellon University. Pittsburgh, PA. May, 2000.

WEISER, M. (1991). **The Computer for the 21st Century.** Scientific American. v. 265, n. 3, p.94-104, fev. 1991.