Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica

Departamento de Engenharia de Computação e Automação Industrial

Especificação Unificada de Sistemas de Software no Paradigma Orientado a Objetos

Tese de Doutorado

Autor : Joinvile Batista Junior

Orientador : Prof. Dr. Mario Jino

Janeiro de 1995

Tese apresentada à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas, como parte dos requisitos exigidos para obtenção do título de Doutor em Engenharia Elétrica.

"Bendito seja Deus que não rejeita a minha oração, nem afasta de mim a sua graça." (Salmos 27- 6)

Dedico este trabalho à memória de meus pais, cuja fonte de inspiração permanece viva.

Agradecimentos

Ao professor Mário Jino pela orientação, pelo apoio, pela confiança e boa vontade.

Ao coordenador do projeto CHILL Márcio Machado Pereira pela oportunidade que me proporcionou de participar do desenvolvimento de uma ferramenta de software no paradigma orientado a objetos e pelo apoio que tenho recebido na realização do meu trabalho.

Ao colega do projeto CHILL Gustavo Leite de Mendonça Chaves pela definição de todos os requisitos e da concepção das classes de objetos do núcleo do sistema de software, do qual foram utilizadas partes para aplicação da metodologia proposta, e pelas críticas a algumas das heurísticas propostas neste trabalho.

Resumo

O objetivo deste trabalho é o de estabelecer premissas para avaliação de metodologias de análise e projeto orientadas a objetos, baseadas na forma de representação e no poder heurístico de tais metodologias; uma metodologia alternativa que satisfaça as premissas estabelecidas é também proposta.

Metodologias bem conhecidas da década de 90 são descritas e analisadas segundo as premissas estabelecidas. Abordagens de reuso de software são resumidas com o objetivo de caracterizar suas vantagens e suas fraquezas.

A metodologia proposta, denominada Especificação Unificada, baseia-se em uma representação única a partir da qual são extraídas visões correspondentes às representações comumente geradas durante a especificação de requisitos e a análise. Heurísticas são também propostas com o objetivo de guiar o desenvolvimento de sistemas de software e torná-los mais adequados para ações de manutenção corretiva e evolutiva.

Abstract

The purpose of this work is to establish criteria for the evaluation of object-oriented analysis and design methodologies, based on representation models and the heuristic power of methodologies; an alternative methodology satisfying such criteria is also proposed.

Well known methodologies from the early nineties are described and analyzed according to the proposed criteria. Software reuse approaches are summarized to point out their strengths and limitations.

The methodology proposed here, called "Especificação Unificada", is based on a single representation from which views corresponding to representations usually produced during the requirements specification and analysis phases can be retrieved. Heuristics are also proposed to guide the development of software systems and to make them more adequate for actions of evolutive and corrective maintenance.

Índice

Lista de Figuras

1. Introdução

1.1 Motivação

O perfil da informática na década de 90 deverá ser caracterizado por extensa interligação entre sistemas distribuídos, com integração de dados, som e imagem. O suporte ao reuso em domínios bem estabelecidos (tais como interfaces gráficas e estruturas de dados) tende a ser padronizado, mas o reuso de software aplicativo continua sendo um desafio para aumentar a produtividade e qualidade dos sistemas. Neste cenário de demanda crescente de software cada vez mais complexo, ações de manutenção corretiva e evolutiva tornam-se cada vez mais frequentes.

As linguagens de programação orientadas a objetos representam uma evolução no sentido de construir software mais robusto e confiável, viabilizando:

- o encapsulamento de informação em componentes autônomos (acessados via protocolos representados pelos seus métodos);
- a especialização de componentes fatorados e a instanciação de componentes genéricos, concorrendo para diminuir duplicações e inconsistências;
- a composição de componentes para criar camadas de serviços, elevando o nível de programação em determinados domínios.

A engenharia de software depara-se com o grande desafio de suportar a passagem da produção artesanal de software para o processo industrial. Houve uma proliferação e um conseqüente amadurecimento de metodologias de análise e projeto orientadas a objetos no início desta década. Contudo, permanece a questão fundamental: até que ponto essas metodologias têm conseguido guiar a decomposição de sistemas de software para torná-los mais adaptados a ações de manutenção corretiva e evolutiva?

Metodologias de análise e projeto de software têm sido questionadas quanto ao modelo que propõem para o mapeamento do domínio do problema como uma cooperação de objetos [Hoydalsvik93] [Constantine92]. As propostas iniciais vinculavam o paradigma orientado a objetos às vantagens decorrentes do mapeamento do mundo real, com ênfase no modelo estático de objetos. Propostas posteriores têm resgatado a importância do modelo dinâmico e até mesmo do modelo funcional (que constituem a base da análise estruturada).

A motivação deste trabalho está baseada na seguinte proposta:

- estabelecer premissas de avaliação de metodologias de análise e projeto orientadas a objetos baseadas nos seus dois aspectos fundamentais: a forma de representação e o conjunto de heurísticas para a decomposição de sistemas de software;
- avaliar reconhecidas metodologias orientadas a objetos desta década segundo tais premissas;
- caracterizar as vantagens e as fraquezas das diferentes categorias de abordagens de reuso;
- e, principalmente, propor uma metodologia que contemple as premissas estabelecidas e se apresente como uma alternativa para reuso de software aplicativo.

A metodologia proposta neste trabalho é denominada Especificação Unificada por basear-se em uma representação única, para requisitos, análise e projeto, a partir da qual são recuperadas visões correspondentes aos níveis de abstração da especificação de requisitos e da análise de um sistema de software. Grande ênfase é dada à vinculação dos requisitos, tanto sob o aspecto de guiar a decomposição do sistema como o de permitir o seu rastreamento para subsidiar ações de manutenção corretiva e evolutiva.

Heurísticas contemplando a decomposição de sistemas de software são propostas, baseadas na identificação dos serviços prestados pelo sistema e na manipulação da informação a eles associada.

1.2 Organização

O Capítulo 2 conceitua reusabilidade de software, caracteriza as vantagens e as fraquezas de cada uma das categorias de abordagem de reuso e conclui analisando as características que uma metodologia de análise e projeto deve incorporar para suportar reuso de software aplicativo.

O Capítulo 3 aborda os seguintes aspectos:

- caracteriza o paradigma orientado a objetos como uma evolução dos paradigmas anteriores;
- estabelece premissas de avaliação de metodologias de análise e projeto;
- apresenta descrições esquemáticas de reconhecidas metodologias desta década;
- compara as metodologias apresentadas em função das premissas estabelecidas;
- propõe a metodologia Especificação Unificada sob o ponto de vista das premissas.

O Capítulo 4 ilustra a linguagem de representação da metodologia proposta, através de exemplos, abordando: especificação de requisitos (abrangendo interface com usuários), classes e serviços.

O Capítulo 5 apresenta o conjunto de heurísticas associado à metodologia proposta.

O Capítulo 6 ilustra a recuperação de visões a partir da Especificação Unificada com um exemplo retirado de uma ferramenta de software desenvolvida segundo o paradigma orientado a objetos.

O Capítulo 7 estabelece o contexto da Especificação Unificada:

- analisando os conceitos incorporados de outras metodologias, os conceitos propostos com correspondências em outras metodologias e as suas contribuições;
- analisando-a em função das premissas estabelecidas nos capítulos anteriores;
- caracterizando a sua abrangência;
- comentando uma experiência de sua utilização.

O Capítulo 8 conclui a proposta apresentando:

- uma revisão dos resultados alcançados;
- e sugestões para trabalhos futuros. □

2. Reusabilidade de Software

2.1 Conceituação

Reusabilidade é um princípio geral de engenharia que objetiva evitar duplicação e capturar características comuns entre tarefas similares [Wegner89].

O alvo da reusabilidade de software é toda informação gerada nos processos de desenvolvimento de software [Freeman83], tais como:

- fragmento de código: pedaço de programa ou de uma especificação executável;
- estruturas lógicas: projeto interno do sistema (módulos, dados e relacionamentos);
- arquiteturas funcionais: projeto externo do sistema na forma de sistemas genéricos;
- conhecimento externo: área de aplicação e conhecimento do processo de desenvolvimento (ciclos de vida, testes e qualidade);
- conhecimento a nível ambiental: conhecimento de utilização e transferência de tecnologia.

A reusabilidade de software envolve determinados compromissos [Biggerstaff87]:

- generalidade de aplicação versus ganho: a generalidade concorre para aumentar o potencial de reuso; no entanto, concorre também para aumentar a complexidade e a ineficiência dos componentes;
- tamanho dos componentes versus potencial de reuso: à medida que o componente cresce a economia em reusá-lo é maior; no entanto, o componente se torna mais específico, diminuindo o seu potencial de reuso;
- número de componentes versus custo de criação: a reusabilidade se torna mais efetiva à medida que a disponibilidade de componentes de software aumenta; no entanto, cresce o custo de geração da população de componentes.

Neste capítulo, a maioria dos conceitos utilizados são originários de um *survey* sobre reusabilidade de software [Krueger92], cuja referência considere-se implícita, para evitar sua repetição desnecessariamente. Referências a outros trabalhos serão explicitadas.

Abstração tem sido usada para gerenciar a complexidade do software. Segundo Wegner, abstração e reuso são dois lados da mesma moeda:

- toda abstração descreve uma coleção de entidades reusáveis;
- toda coleção relacionada de entidades reusáveis determina uma abstração.

Uma abstração sobre um artefato de software suprime os detalhes que não são importantes e enfatiza o que é importante. Toda abstração de software tem dois níveis: especificação e realização. Quando abstrações são organizadas em camadas, a especificação de um dado nível é a realização do seu nível imediatamente superior.

Na concepção de uma abstração o seu criador arbitra o particionamento da abstração nas seguintes partes:

- parte escondida: engloba os detalhes da realização não visíveis na especificação;
- partes visíveis na especificação que inclui:
  - parte fixa: representa as características invariantes;
  - parte variável: representa as características variantes, mapeando em uma coleção de possíveis realizações.

Raramente especificações semânticas são derivadas dos seus princípios básicos. Certa quantidade de especificação de abstração é assumida como domínio comum. Em domínios bem estabelecidos, um nome pode ser suficiente para definir a semântica de uma abstração (por exemplo a referência ao nome *stack* no domínio de tipos abstratos de dados é suficiente para definir a semântica da abstração correspondente).

Abstração exerce um papel central e limitante em relação a outras facetas de reuso, tais como:

- seleção: artefatos reusáveis devem ser abstrações concisas que possam ser eficientemente localizadas, entendidas, comparadas e selecionadas de uma coleção;
- especificação: um artefato reusável generalizado corresponde a uma abstração com parte variável, na qual a especialização corresponde à escolha de uma possível realização;
- integração: o entendimento de um artefato de software, a partir de uma abstração que suprima detalhes, é crucial para a integração do artefato.

Distância cognitiva pode ser definida como a quantidade de esforço intelectual que deve ser dispendida por projetistas de software para levar um sistema de software de um estágio para o outro. Abordagens de reuso serão mais efetivas à medida que diminuirem a distância cognitiva entre um conceito inicial e sua implementação executável, o que pode ser conseguido a partir das seguintes características:

- as partes visíveis (fixa e variável) devem ser sucintas e expressivas;
- a parte escondida deve ser maximizada;
- o mapeamento da especificação para a realização deve ser automatizado.

A tecnologia ideal de reuso é aquela que suporta as seguintes características:

- facilidade para selecionar, especializar e integrar especificações que satisfazem requisitos informais (associados aos conceitos iniciais sobre o problema);
- tradução automática para um sistema executável.

Para que o reuso seja efetivo os projetistas devem estar familiarizados com as abstrações envolvidas ou ter tempo para estudá-las. Esta premissa justifica o sucesso de reuso em domínios bem estabelecidos (bibliotecas matemáticas, tipos abstratos de dados e interfaces gráficas) ou em domínios cujos usuários estão familiarizados com os conceitos (usuários de geradores de aplicação em domínios específicos).

2.2 Abordagens de Reuso

Abordagens de reuso podem ser agrupadas em sete grandes categorias:

- recuperação de código (e de projeto);
- componentes baseados em código fonte;
- esquemas de software;
- geradores de aplicação;
- *VHLLs* (*very high level language*)
- sistemas transformacionais
- arquiteturas de software

O objetivo desta seção não é descrever cada uma dessas categorias de abordagem de reuso, mas sim caracterizar suas vantagens e deficiências.

2.2.1 Recuperação de Código

Recuperação de código corresponde à cópia de um bloco contíguo de código fonte de um sistema existente. Recuperação de projeto pode ser feita a partir da mesma fonte, mas omitindo detalhes internos.

Em função da não caracterização de partes escondidas, não ocorre a distinção entre os níveis de abstração correspondentes à especialização e realização, trazendo como consequência a necessidade de atividades manuais para localização e modificação de código. A realização destas atividades está associada a uma grande distância cognitiva, equivalente à implementação de código. Na prática a efetividade desta forma de reuso é muito restrita devido à sua informalidade, o que pode ser observado em função das seguintes características:

- altamente dependente do conhecimento do projetista;
- não há uma forma sistemática de partilhar fragmentos de código entre vários programadores;
- especialização e integração são ineficientes devido à usual necessidade de entendimento e edição de código, o que pode introduzir erros prejudicando a estabilidade do código.

2.2.2 Componentes Baseados em Código Fonte

Reuso de componentes baseados em código fonte é muito bem sucedido em domínios bem estabelecidos, nos quais o nome (seno, multiplicação de matrizes, *stack*, listas, *string*) identifica o componente.

A especialização de componentes ocorre basicamente de três formas:

- por edição de código;
- por instanciação de componentes genéricos;
- por herança de uma classe de objetos.

Na edição de código, o esforço requerido para entender o componente pode comprometer o ganho do reuso e a edição pode invalidar a correção do componente. A distância cognitiva é portanto bem menor na especialização por instanciação ou por herança.

O maior desafio para implementar grandes bibliotecas de componentes é definir abstrações concisas para os componentes, uma vez que o usuário não deve precisar examinar código para procurar ou escolher um componente. O implementador da biblioteca deve prover especificações que descrevam sucintamente o comportamento do componente associadas a esquemas de classificação e de recuperação.

Bibliotecas de componentes de propósito geral permanecem utópicas, uma vez que as características de implementação e compromissos para estruturas de dados e computações são muito variáveis.

2.2.3 Esquemas de Software

Esquemas de software enfatizam o reuso de algoritmos abstratos e estruturas de dados em vez de código fonte. O maior desafio é achar abstrações adequadas e representações para os esquemas. Diferentes algoritmos e estruturas de dados têm diferentes propriedades e esta diversidade precisa ser expressa nos esquemas.

Várias formas têm sido utilizadas para representar esquemas de software:

- *Plan Calculus*, utilizado em *Programmer's Apprentice* (Rich, Waters), combina idéias de *flowcharts*, abstração de dados, formalismos lógicos e transformações de programas;
- PARIS (Katz, Richter, The) utiliza asserções lógicas para expressar a semântica das especificações, restrições de instanciação e de validação da utilização dos esquemas;
- LIL (Goghen) utiliza teorias axiomáticas com grau variável de formalidade associadas a parâmetros dos esquemas e visões para descrever como instanciações satisfazem as teorias;
- Software Templates (Volpano, Kieburtz) utiliza notação funcional.

Em *Plan Calculus*, *flowcharts* (que tem isomorfismos com formalismos lógicos) são utilizados para descrever aspectos algorítmicos (fluxo de controle e de dados) e anotados com pré e pós condições (de forma a capturar aspectos declarativos). Caixas vazias, utilizadas nos *flowcharts*, representam a parte variável da especificação, podendo ser preenchidas com esquemas aninhados.

Em PARIS, a especificação constitue-se de:

- descrição semântica formal - o que o esquema faz;
- asserções para instanciação - restrições sobre as partes variáveis do esquema;
- asserções para validar o uso - pré e pós-condições para o código fonte gerado automaticamente.

O usuário de PARIS fornece um conjunto de requisitos formais, e é auxiliado na seleção do esquema com um suporte sofisticado baseado nas pré-condições e pós-condições dos requisitos e

da população de esquemas disponíveis. Infelizmente o usuário interage a nível de especificação, mas não tem a vantagem das abstrações de alto nível, uma vez que não pode simplesmente especificar um nome (como *queue*): precisa fornecer uma especificação lógica elaborada da semântica (de *queue*). Essa abstração de baixo nível é necessária para o provador de teoremas (utilizado em PARIS), mas não é ideal para seres humanos. Além disso, achar um esquema que satisfaça os requisitos implica em provar sua consistência lógica com os requisitos, o que em geral (para qualquer tipo de requisito) está além da tecnologia disponível para provadores de teoremas. Segundo Katz et all, "não é tarefa trivial prover um sistema com interface amigável para o usuário e ao mesmo tempo produzir entrada correta e eficiente para o mecanismo do provador de teoremas".

Esquemas são especializados de duas formas:

- substituição de construções da linguagem por fragmentos de código, especificações ou esquemas aninhados nas partes parametrizadas dos esquemas (como em PARIS);
- escolha a partir de uma enumeração pré-definida de opções (como em PARIS e LIL).

Esquemas reduzem a distância cognitiva se comparados a componentes de código fonte, uma vez que o trabalho do usuário consiste em especializar algoritmos e estruturas de dados. Infelizmente não existem muitas abstrações universais em sistemas de software (como funções matemáticas e os tipos abstratos de dados já conhecidos). O desafio, ao utilizar formalismos lógicos e linguagens de especificação, é achar formalismos naturais, sucintos e expressivos.

2.2.4 Geradores de Aplicação

Geradores de aplicação traduzem especificação de entrada (abstrações de um domínio muito restrito) para programas executáveis (em contraste com esquemas onde apenas um módulo ou um subsistema é produzido). Seus artefatos reusáveis são: arquiteturas, subsistemas, estruturas de dados e algoritmos muito específicos. São apropriados quando:

- muitos sistemas de software similares são escritos;
- um sistema de software é modificado e reescrito muitas vezes durante o seu tempo de vida;
- muitos protótipos do sistema são necessários para convergir para um produto estável.

Diferentes formas de representação podem ser utilizadas pelo usuário:

- linguagens de especificação textual (linguagens orientadas à aplicação e linguagens de quarta geração);
- *templates;*
- diagramas gráficos;
- diálogos dirigidos por menus iterativos;
- interfaces orientadas a estruturas.

A especialização é a tarefa básica do usuário de um gerador de aplicações. Nos geradores de aplicações orientados a negócios, as aplicações são intensivas em dados:

- gerenciamento de base de dados: especialização de tipos, campos e relações;

- geração de relatórios textuais: descrição através de linguagem declarativa baseada em recuperação de dados associativos;
- geradores de relatórios gráficos: entrada similar, produzindo saídas gráficas (histogramas e gráficos de pizzas);
- manipuladores de bases de dados: restrições de controle de acesso, entre itens e sobre dados particulares.

Outros exemplos de geradores podem ser citados, para ilustrar o tipo de especialização feita pelos usuários:

- geradores de sistemas especialistas que incorporam métodos comuns de solução de problemas: o usuário deve determinar hipóteses que explicam os sintomas, identificando dados para diferenciá-las;
- geradores de analisadores léxicos (como Lex): entrada descreve os *tokens* legais através de expressões regulares;
- geradores de *parsers* (como Yacc): entrada descreve as produções da linguagem (às quais podem ser associadas ações, descritas na linguagem C, para checar semântica estática, geração de código, etc) a ser analisada em uma gramática livre de contexto.

A distância cognitiva entre a especificação informal e as abstrações específicas da aplicação (entrada para o gerador de aplicação) é muito pequena. Os usuários só constroem ou modificam especificações de requisitos, uma vez que a análise, projeto e geração de código são incorporados pelo gerador de aplicações. Não há necessidade de testes: o usuário só precisa validar a especificação de entrada em relação às intenções de projeto. Geradores de aplicação incorporam entendimento da semântica do seu domínio de aplicação livrando o usuário de derivar a semântica do domínio repetidamente a partir dos princípios básicos.

2.2.5 *Very High Level Languages* (*VHLLs*)

VHLLs são conhecidas como linguagens de especificação executáveis. Aceitam código fonte na entrada e usam compiladores, da mesma forma que as *HLLs* (*High Level Languages*). No entanto, da mesma forma que o código fonte escrito em uma linguagem *HLL* pode ser uma ordem de magnitude mais sucinto que as suas implementações em linguagem *assembly*, código fonte escrito em uma linguagem *VHLL* pode ser também uma ordem de magnitude mais sucinto do que as suas implementações em linguagem *HLL*.

Quanto mais geral a tecnologia de reuso, mais difícil implementar um sistema com essa tecnologia. *VHLLs* provêem sintaxe e semântica para expressar computação de propósito geral, enquanto geradores de aplicação, embora incorporem somente conhecimento restrito a um domínio muito específico, são capazes de aceitar especificações de requisitos como entrada. *VHLLs* são portanto, mais abrangentes que geradores de aplicações, porém, menos sucintas e menos poderosas.

O objetivo básico das *VHLLs* não é a eficiência na execução mas sim, na implementação e modificação de programas. *VHLLs* usam abstrações matemáticas, baseadas por exemplo em teoria dos conjuntos, raramente utilizadas no desenvolvimento de software convencional. Seus artefatos reusáveis correspondem a padrões reusáveis em *HLLs*. Três linguagens são frequentemente citadas como exemplos de *VHLLs*: *SETL*, *PAISLey* e *MODEL*.

SETL é baseada na teoria dos conjuntos. Seus tipos de dados são:

- atômicos:inteiros, reais ou booleanos; ou
- compostos: conjuntos (*sets*) e tuplas.

Conjuntos (coleção não ordenada finita) e tuplas (seqüência ordenada finita) são formados pela enumeração de seus membros ou por formadores de conjuntos da forma:

- {<expressão> : x1, . . . xn | <condição>}.

Operações sobre conjuntos incluem: união, interseção, diferença, subconjuntos (*powersets*) e iteradores. Mapas (*maps*) são providos como funções da teoria dos conjuntos (ou seja, F(x) leva x a um único valor). As operações comuns aplicadas a tipos abstratos de dados podem ser implementadas trivialmente nesta linguagem. *SETL* é considerada uma linguagem de largo espectro por prover construções VHLL (originárias da teoria dos conjuntos) e HLL (tais como seleção e iteração). Também por motivação de eficiência, os usuários podem especializar construções *defaults* adotados pelo compilador durante o tempo de execução.

PAISLey é composta por abstrações da programação funcional baseada em conjuntos e de processos assíncronos comunicantes. Os aspectos baseados em conjuntos são similares a *SETL*. Usuários podem definir uma coleção de processos assíncronos que se comunicam via funções de troca (*exchange functions*), que se constituem em uma extensão à programação funcional que permite modelar paralelismo na linguagem. Adicionalmente restrições de tempo real permitem avaliação estática e dinâmica da execução a partir da associação às funções de: limites superiores e inferiores, distribuições e valores randômicos.

O ambiente de execução de *PAISLey* permite a execução de programas incompletos. Ao avaliar uma função incompleta, o ambiente de execução pode escolher uma das seguintes alternativas:

- no modo *defaut*: retornar um valor *default*;
- no modo aleatório: retornar um valor aleatório;
- no modo interativo: retornar um valor interativo.

MODEL é uma linguagem declarativa baseada na solução simultânea de uma coleção de equações de restrição (contém declarações de dados e de equações de consistência). Seu compilador realiza análise de fluxo de dados para determinar se a consistência das equações pode ser unicamente satisfeita e determina a ordem das computações (fluxo de controle). O compilador checa adicionalmente se todas as variáveis usadas foram definidas e se não há circularidade nas restrições.

Linguagens têm forte influência na solução do problema:

- SETL e PAISLey encorajam o usuário a pensar em termos de conjuntos de objetos e de funções sobre estes conjuntos;
- MODEL encoraja o usuário a pensar em termos de restrições entre objetos estáticos.

A melhor linguagem para uma dada aplicação é a que provê a menor distância cognitiva dos conceitos iniciais à implementação executável.

A equipe que desenvolveu a linguagem *SETL* utilizou-a para prototipar um compilador Ada. Não foi demonstrado, no entanto, se teoria dos conjuntos é adequada para a população geral dos projetistas de software na construção de um largo espectro de aplicações.

2.2.6 Sistemas Transformacionais

Sistemas transformacionais operam em duas fases:

- primeira fase: descrição do comportamento semântico do sistema usando uma linguagem de nível mais alto que as *VHLLs* puras (*SETL, PAISLey, MODEL*) mas com desempenho pobre na tecnologia de compiladores;
- segunda fase: transformações são aplicadas, sem alterar o comportamento semântico, para aumentar a eficiência guiadas iterativamente pelo usuário (a intervenção humana é necessária porque a seleção automática de algoritmos e estruturas de dados está além da tecnologia dos compiladores).

A fase de transformação pode utilizar tipos distintos de artefatos de reuso: protótipos, histórias do desenvolvimento e transformações.

Protótipos são utilizados em sistemas transformacionais com dois objetivos:

- como uma especificação formal do sistema; e
- como base para a geração de uma implementação executável eficiente a partir da aplicação de transformações.

Histórias do desenvolvimento correspondem a seqüências de transformações aplicadas durante o desenvolvimento de um sistema. *PADDLE* (Balzer) é um sistema transformacional que armazena as transformações utilizadas durante o desenvolvimento para permitir a reutilização das mesmas durante a manutenção corretiva ou evolutiva. Para solucionar limitações do sistema PADDLE, que armazena seqüências de transformações sem explicitar as decisões de projeto que as motivaram (dificultando o entendimento e conseqüentemente as modificações necessárias), foi concebido o sistema transformacional Glitter (Fickas) que armazena a seqüência de transformações na forma de regras (cujo nível mais alto de abstração facilita o entendimento e o reuso).

A manutenção corretiva e evolutiva são realizadas sobre a especificação (na qual a informação é localizada e independente) e não sobre a implementação (na qual, por motivos de eficiência, a informação se encontra dispersa e interdependente).

Transformações são mapeamentos de padrões sintáticos de código em padrões mais eficientes mas funcionalmente equivalentes. Uma transformação é composta das sequintes partes:

- padrão de casamento (*pattern matching*): *template* de código que define o padrão a ser pesquisado;
- condições de aplicabilidade: restrições semânticas sobre a aplicação da transformação;
- padrão de substituição: padrão de código que substitui o padrão anterior (localizado a partir do padrão de casamento).

A distância cognitiva é muito pequena na primeira fase mas é maior na segunda fase devido à necessidade de intervenção humana.

2.2.7 Arquiteturas de Software

Arquiteturas de software são *frameworks* e subsistemas que capturam a estrutura global de um sistema de software. Exemplos de arquiteturas de software incluem:

- subsistemas de base de dados que permitem a especialização e reuso em diferentes aplicações;
- *frameworks* para compiladores nos quais diferentes analisadores léxicos, *parsers* e geradores de código podem ser inseridos;
- arquiteturas adaptáveis para interface com usuários.

Arquiteturas de software são análogas a esquemas de larga escala com a diferença de que o seu foco de atenção está na interação entre subsistemas e não em estruturas de dados e algoritmos. Também são análogas a geradores de aplicação com a diferença que, ao invés de adotarem uma arquitetura implícita, permitem especializações e composições de diferentes arquiteturas.

O sistema Draco [Neighbors84] [Freeman87] pode ser utilizado como um gerador de arquiteturas de software. O sistema Draco é baseado na definição de linguagens de domínio, cujos objetos e operações são o resultado de uma análise de domínio. A especificação de um domínio no sistema Draco é composta de cinco partes:

- um analisador sintático: define a sintaxe externa da linguagem de domínio (em notação BNF acrescida de mecanismos de controle tais como recuperação de erros e *backtracking*) e forma interna prefixa (árvore com nome de atributo e dados em cada nó);
- formatador: descreve como produzir a sintaxe externa referente aos fragmentos de programa escritos na linguagem de domínio (uma vez que o sistema pode precisar interagir com o usuário acerca de partes incompletas do programa em desenvolvimento);
- transformações: regras de otimização entre objetos e operações do domínio na linguagem do domínio (por exemplo, representação de quadrado de um número com uma multiplicação);
- componentes: especificam a semântica do domínio para cada um dos objetos e operações do domínio aos quais associam diferentes implementações (por exemplo, exponenciação refinidada através de deslocamento binário ou através de expansão de Taylor);

- *procedures*: usadas quando o conhecimento para realizar alguma transformação específica é de origem algorítmica.

A solução de um problema referente a um domínio, conhecido pelo sistema Draco, requer a descrição do problema na linguagem de domínio já existente, resultando na reutilização da descrição do domínio previamente fornecida ao sistema Draco. A solução de um problema pertencente a um domínio similar aos conhecidos pelo sistema Draco envolve a definição de um novo domínio com a reutilização das descrições de seus domínios similares existentes no sistema. A especificação de domínios similares a partir dos domínios previamente especificados conduz a uma hierarquia de domínios disponíveis no sistema.

A geração de um novo domínio ocorre a partir da geração de uma linguagem para este domínio reutilizando as linguagens dos domínios existentes.

Arquiteturas de software têm baixa distância cognitiva, uma vez que arquiteturas correspondem a abstrações de alto nível cujo mapeamento para a implementação é automatizado (ou quase totalmente automatizado).

2.3 O Papel de Reuso no Desenvolvimento e Manutenção de Software

Reusabilidade de software tem sido aplicada de forma restrita:

- bibliotecas de componentes em domínios bem estabelecidos (por funções matemáticas e tipos abstratos de dados);
- geradores de aplicação em domínios específicos (como planilhas de cálculo).

A pequena abrangência das tecnologias de reuso tem sido um empecilho à utilização de reuso em larga escala. Esta afirmação pode ser constatada em função da utilização restrita das tecnologias de reuso, tais como:

- bibliotecas de componentes de propósito geral continuam sendo utópicas: devido a compromissos muito diversos entre estruturas de dados e computações e devido à falta de abstrações precisas para componentes que suportem entendimento e seleção de componentes;
- geradores de aplicação são bastante poderosos por utilizarem abstrações de domínios e gerarem a implementação automaticamente; contudo, têm uma cobertura de domínio muito restrita;
- asserções lógicas (utilizadas por esquemas de software, VHLLs e sistemas transformacionais) não são úteis para grandes sistemas devido à complexidade das expressões.

As tecnologias que requerem como entrada a derivação de especificações semânticas a partir dos seus princípios básicos (esquemas de software, *VHLLs* e sistemas transformacionais) não têm se popularizado devido à dificuldade de especificar os requisitos de entrada. Esta afirmação pode ser justificada em função das seguintes observações:

- uma especificação lógica elaborada, que corresponde a abstrações de baixo nível nas quais o usuário não pode simplesmente recorrer a nomes, é adequada para provadores de teoremas mas não para seres humanos;
- abstrações matemáticas são raras no desenvolvimento de sofware convencional: não têm se mostrado acessíveis à população geral dos projetistas de software na construção de um largo espectro de aplicações.

Geradores de aplicação têm uma interface amigável, ao incorporar entendimento da semântica do domínio, livrando o usuário de derivá-lo repetidamente dos princípios básicos; infelizmente, não se aplicam ao desenvolvimento de software de propósito geral.

A melhor linguagem é aquela cuja distância cognitiva entre os conceitos iniciais e implementação é a menor possível. A utilização do poder de abstração dos nomes como entrada para tecnologias de reuso é uma alternativa significativa para diminuir a distância conceitual. Pessoas ao se referirem a conceitos bem estabelecidos fazem-no designando-os através de nomes, sem necessitar especificar suas características para serem entendidas (pode-se imaginar o ônus de referir-se ao conceito *stack* sempre através de sua especificação em vez de simplesmente utilizar o nome que o especifica). O sucesso no uso de componentes é maior em domínios bem estabelecidos nos quais o nome é suficiente para recuperar e entender o componente. A Figura 1 ilustra o papel desempenhado pela utilização de nomes para especificar conceitos.

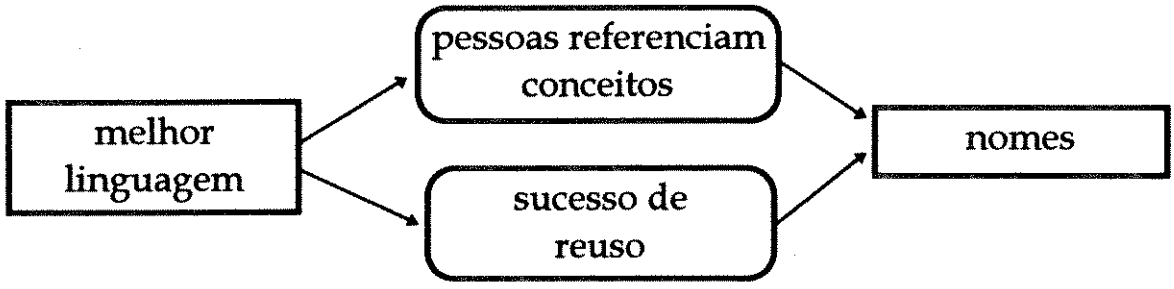

Figura 1: Utilização de Nomes para Especificar Conceitos

Na manutenção corretiva e evolutiva de software aplicativo a recuperação de código e de projeto é mandatória. Uma vez que esta técnica de reuso é justamente aquela na qual a distância cognitiva é maior, a concepção de software reusável e o suporte ao entendimento dos requisitos que o originaram (fundamental para ações de manutenção corretiva e evolutiva) constituem desafios significativos.

A utilização do paradigma orientado a objetos, embora seja de grande auxílio na construção de classes de objetos primitivos e fatorados por generalidade, herança e composição de classes de objetos primitivos, tem suas limitações e seus *trade-offs*.

Reuso deve estar incorporado na metodologia de desenvolvimento de software. A linguagem de representação deve facilitar a associação de nomes dos conceitos no domínio de aplicação aos nomes das classes de objetos, seus atributos, relacionamentos e métodos.

A vinculação dos requisitos à decomposição de software é fundamental para o suporte ao entendimento do sistema e à avaliação do impacto da alteração de requisitos por decorrência de ações de manutenção corretiva e evolutiva. Uma forma sistemática de partilhar código é dispor de suporte automatizado ao entendimento de seus vínculos com os requisitos de projeto. A caracterização de partes escondidas (através de mecanismos adicionais ao mecanismo de restrição de visibilidade suportado pelo paradigma orientado a objetos) pode ser ampliada a partir da recuperação de visões de níveis distintos de abstração. Esta recuperação permite a obtenção da especificação cuja informação localizada e independente é mais adequada ao entendimento do que a usual informação dispersa e interdependente da implementação.

3. Metodologias de Análise e Projeto Orientadas a Objeto

3.1 A evolução das Metodologias de Análise e Projeto

Metodologias de análise e projeto de sistemas de software têm por objetivo guiar o desenvolvimento e a manutenção corretiva e evolutiva de sistemas que manipulam informação que, em geral, é recebida, transformada, armazenada e fornecida ao meio externo.

Os modelos de ciclo de vida de desenvolvimento de software têm evoluído de uma abordagem sequencial (modelo em cascata), na qual as fases de desenvolvimento (especificação de requisitos, análise, projeto e implementação) são serializadas, para abordagens que propõem a sobreposição e a realimentação entre as fases (modelo espiral e modelo de fonte). A utilização de sobreposição e realimentação entre as fases é mais adequada para o paradigma orientado a objetos, para o qual é amplamente aceita a necessidade de utilização alternada de estratégias *top-down* e *bottom-up* na concepção das classes de objetos.

Sistemas de software podem ser conceituados como compostos de uma parte reativa (que troca informação com o ambiente externo) e de uma parte não reativa (que não interage com o ambiente externo). Sistemas de informação têm, em geral, uma parte reativa bastante significativa, enquanto existem sistemas nos quais a parte reativa é nula (por exemplo, compiladores).

Pode-se observar que, historicamente, conceitos associados a um novo paradigma são inicialmente incorporados às linguagens de programação, seguidos pelo aparecimento de metodologias de projeto e finalmente pelas metodologias de análise. O encapsulamento de informação (e de seus procedimentos afins) nas linguagens de programação orientadas a objetos é o resultado de uma evolução no suporte a diferentes paradigmas [Stroustrup88]:

- programação baseada em algoritmos: ênfase na utilização de dados globais para compartilhamento entre vários procedimentos (exemplo: Pascal);
- programação baseada em módulos: encapsulamento de dados e procedimentos em módulos com sua exportação e importação seletiva entre os módulos (exemplo: Modula 2);
- programação baseada em tipos abstratos de dados: dados (atributos) e procedimentos (métodos) são encapsulados em tipos abstratos de dados, onde os atributos são dados privados do tipo abstrato que os contém, com acesso externo somente via métodos do mesmo tipo abstrato, segundo o conceito de informação escondida [Parnas72] (exemplo: Ada);
- programação baseada em classes de objetos: atributos e métodos comuns podem ser herdados por classes especializadas, com a opção de se redefinir métodos herdados (exemplo: C++).

Classes de objetos descrevem um grupo de objetos com propriedades similares e com comportamentos, relacionamentos e semântica comuns [Rumbaugh91]. Classes de objetos podem ser vistas como módulos mais primitivos, de forma que um sistema de software pode ser projetado e implementado como uma cooperação de objetos no paradigma orientado a objetos. Do ponto de

vista de reuso de classes de objetos e de métodos, além da fatoração decorrente do uso de herança, linguagens de programação orientadas a objetos suportam em geral:

- a parametrização de tipos (*templates*), que simplifica a definição de *containers* nos tipos abstratos de dados (como listas ligadas, cujo tipo de conteúdo é definido por simples instanciação de tipo);
- a distinção de métodos com mesmo nome através de sua assinatura (nome mais lista de parâmetros);
- a sobreposição de operadores que viabiliza a definição de novos tipos abstratos como *built-ins* da linguagem (como a redefinição dos operadores de soma e subtração para números complexos).

A evolução das metodologias de análise e projeto tem pelo menos quatro marcos significativos, caracterizados pelas seguintes abordagens de desenvolvimento:

- orientação a fluxo de dados;
- orientação à estrutura de dados;
- refinamento da orientação a fluxo de dados através da identificação de eventos;
- orientação a objetos.

Dentre as metodologias orientadas a fluxo de dados, a metodologia de análise estruturada [DeMarco79] propõe a decomposição funcional *top-down* de um sistema utilizando DFDs (diagramas de fluxo de dados), no qual um processo inicial é sucessivamente decomposto em processos, fluxos de dados e armazenadores de dados, até atingir processos primitivos. Dentre as metodologias de projeto, a metodologia de projeto estruturado [Yourdon79] introduz os conceitos de coesão e acoplamento e divide o fluxo de informação (representado nos DFDs) em aferente (fluxos de entrada), centro de transformação e eferente (fluxo de saída). Seus pontos fracos mais significativos são: a quebra da continuidade representacional decorrente do mapeamento dos DFDs em diagramas de estruturas e o fato de não levar em conta a estrutura de dados na decomposição do sistema.

Nas metodologias orientadas à estrutura de dados [Jackson75] [Warnier74] os procedimentos do sistema de software são obtidos a partir de um mapeamento da estrutura de dados do sistema; isto as aproxima das metodologias orientadas a objetos, tendo como diferença marcante a não utilização dos conceitos de herança e de informação escondida (fundamentais na orientação a objetos).

Baseada na idéia de que sistemas são mecanismos de resposta a estímulos, a análise essencial [McMenamin84] propõe a composição funcional *bottom-up* de processos a partir da identificação de eventos, em substituição à decomposição funcional *top-down* (das metodologias orientadas a fluxo de dados). Em algumas classes de problemas (particularmente os sistemas reativos) a identificação dos eventos a serem tratados reduz o sistema a um conjunto de serviços simples. Mas existem classes de problemas para as quais a identificação apenas as reduz a um conjunto de serviços complexos ou simplesmente não se aplica (como nos sistemas *batch*, onde o sistema se

constitui em um único serviço complexo). Nestes casos, a decomposição do sistema deve continuar, necessitando-se, portanto, de heurísticas adicionais.

Na busca de preservar a capacitação existente em análise e projeto estruturado, a análise estruturada (ou uma adaptação dela) foi sugerida como *front-end* para o projeto orientado a objetos [Alabios88]. No entanto, a análise orientada a objetos é considerada o *front-end* mais adequado para evitar a descaracterização do projeto com noções algorítmicas pré-concebidas [Booch91].

A motivação inicial do paradigma orientado a objetos é o mapeamento das entidades do mundo real nas classes de objetos que compõem o sistema, o que não é nada evidente em muitas aplicações cujos domínios são constituídos de entidades mais abstratas (por exemplo um depurador simbólico para linguagem de programação). Contrariando a ortodoxia de que as classes de objetos devem ser extraídas do mundo real como abstrações estáticas às quais comportamento dinâmico pode ser associado [Constantine92], alguns metodologistas [Jacobson92] [Rubin92] têm proposto a descrição do comportamento dinâmico do sistema como base para análise e projeto orientados a objetos, em uma busca clara de inspiração na Análise Essencial. As metodologias de análise e projeto orientadas a objetos devem ser encaradas como uma evolução dos conceitos e heurísticas propostos nas metodologias, aplicados às abordagens anteriores.

É importante observar que os conceitos de evento e informação são essenciais do ponto de vista das metodologias de análise e projeto orientadas a objetos, uma vez que as heurísticas de decomposição de sistemas de software propostas por essas metodologias baseiam-se essencialmente no tratamento de eventos e no tratamento de informação.

3.2 Premissas para uma Metodologia de Análise e Projeto Orientada a Objetos

A busca da transição da produção de software artesanal para um processo industrial, amparado por técnicas de engenharia de software, é o grande desafio à pesquisa na área. Alguns requisitos neste sentido têm sido encontrados na literatura:

- continuidade representacional: utilização das mesmas construções notacionais e sintáticas durante o ciclo completo de desenvolvimento, evitando a necessidade de transformar notações com conseqüente perda de informação [Rumbaugh94a];
- localidade de mudanças: alterações decorrentes de manutenção devem ser restritas, evitando a sua proliferação em parte substancial da implementação do sistema [Lubars92];
- facilidade de evolução em função da alteração de requisitos: o desenvolvimento de sistemas é um processo de progressivas mudanças, à medida que alterações ou novos requisitos são continuamente impostos a quaisquer produtos [Jacobson92];
- rastreabilidade entre modelos: especificações de requisitos, projeto e implementação, deveriam ser representadas e mantidas de forma compatível, de tal forma que alterações em um dado modelo (estático, dinâmico ou funcional) possam ser rastreadas nos demais [Lubars92];

- reusabilidade em todos os níveis de projeto: para aumentar significativamente a produtividade, reuso deve ser um ingrediente natural, ocorrendo nos mais diversos e diferentes níveis [Jacobson92].

Muitos sistemas que incorporam software têm uma longa vida útil (como centrais telefônicas, que chegam a ficar algumas décadas em funcionamento). A manutenção corretiva desses sistemas pode ocorrer em várias fases do desenvolvimento: teste isolado de módulos executáveis (no qual o resto do sistema é simulado), teste de integração em protótipo, teste de campo, e falhas detetadas a partir da liberação do sistema para utilização. A manutenção evolutiva é praticamente mandatória, em função da evolução tecnológica e da mudança de cenário de utilização durante o longo tempo de vida do sistema. A especificação gerada durante o início do desenvolvimento deve cumprir alguns papéis fundamentais durante a manutenção:

- manter correspondência permanente com a implementação;
- servir de base para qualquer caracterização de defeito encontrado ou evolução proposta, de forma a determinar a decisão de manutenção mais adequada (no contexto geral do sistema), evitando remendos mal planejados, com conseqüências desastrosas;
- servir como fonte de componentes, em todos os níveis de abstração, para ações de manutenção corretiva e evolutiva.

Na área de representação do conhecimento, uma forma de representação pode ser avaliada em função de sua adequação representacional (quanta informação sobre o objeto representado pode ser expressa com a representação) e de seu poder heurístico (capacidade de inferência a partir da representação) [Frakes88]. A característica fundamental de uma metodologia de análise e projeto orientada a objetos é a sua capacidade de guiar a decomposição de um sistema em uma representação adequada para o seu desenvolvimento e manutenção. Sob esta nova perspectiva, os fatores enunciados acima são redefinidos da seguinte forma:

- adequação representacional: capacidade de representar adequadamente (de forma concisa, consistente e não ambígua) os requisitos do sistema bem como o vínculo entre os artefatos de software gerados pela decomposição do sistema em todos os níveis de abstração envolvidos, da especificação de requisitos à implementação;
- poder heurístico: capacidade, associada ao conjunto de heurísticas definidas pela metodologia, de guiar a decomposição do sistema em artefatos de software que atendam os requisitos do sistema e suportem reusabilidade e propagação restrita de alterações devidas à manutenção corretiva ou evolutiva.

A premissa de adequação representacional implica:

- representação que capture os requisitos do sistema de forma concisa, não ambígua e consistente;
- manutenção do vínculo entre os artefatos gerados em todos os níveis de abstração.

A manutenção do vínculo entre os requisitos e os artefatos gerados, que torna possível o suporte ao rastreamento dos artefatos afetados pela alteração de requisitos, depende basicamente das seguintes condições:

- continuidade representacional;
- decomposição do sistema a partir dos seus requisitos, associando cada decomposição ao requisito que a justifica.

Do ponto de vista de poder heurístico, metodologias devem prover a capacidade de guiar a decomposição do sistema em artefatos que:

- atendam os requisitos do sistema;
- facilitem reusabilidade;
- restrinjam propagação de alterações por decorrência de manutenção.

A Figura 2 ilustra a definição de premissas para metodolologias de análise e projeto orientadas a objetos.

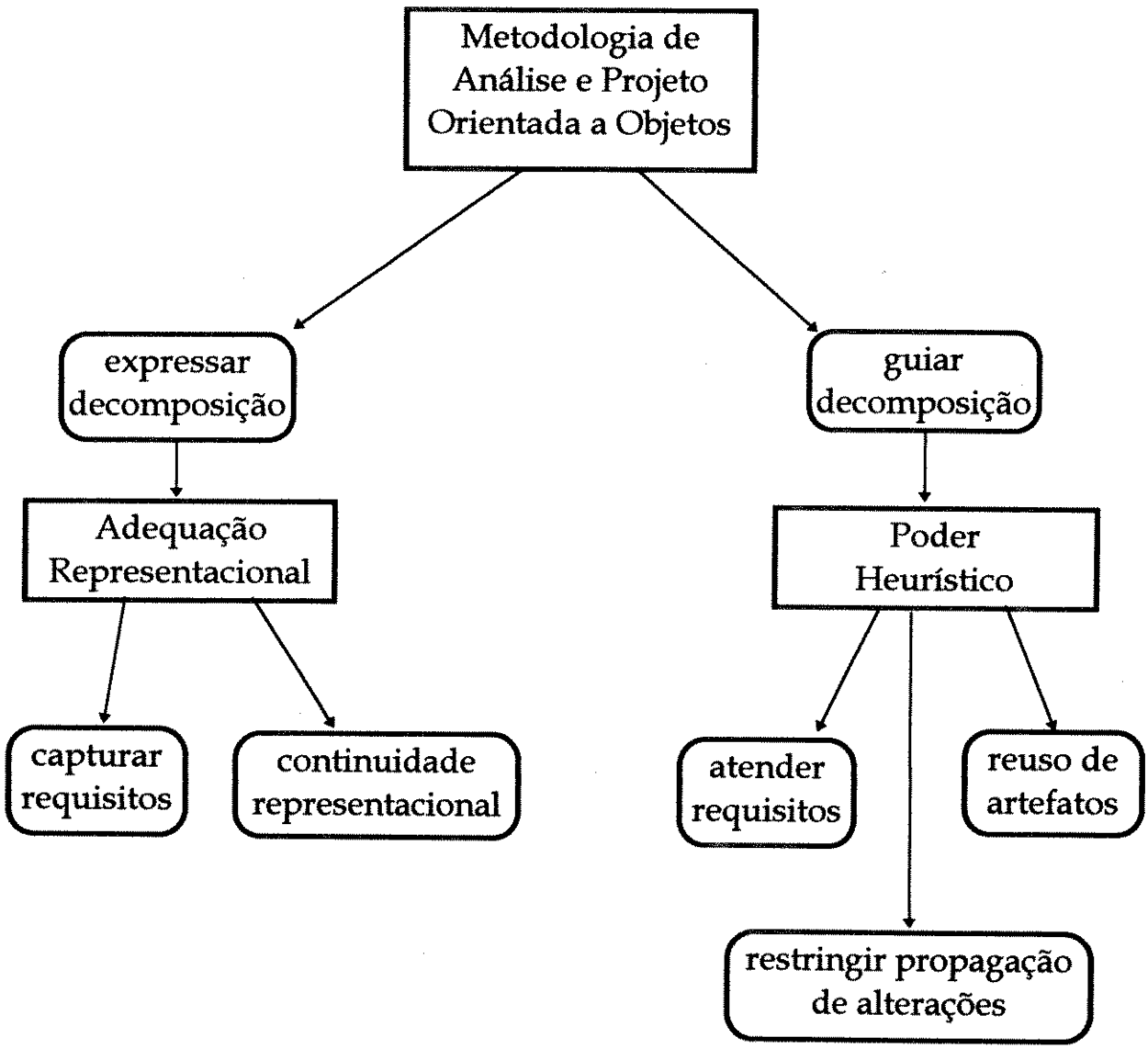

Figura 2: Premissas para Metodologias de Análise e Projeto Orientadas a Objetos

A capacidade de guiar a decomposição do sistema deve se materializar através de heurísticas que suportem o detalhamento do sistema em vários níveis de abstração: na especificação dos requisitos, na análise e no projeto. O atendimento aos requisitos do sistema é conseguido se cada passo de detalhamento do sistema for guiado pelos requisitos (estabelecidos inicialmente e refinados durante a decomposição do sistema). A geração de artefatos reusáveis, com propagação restrita de alterações durante as ações de manutenção corretiva ou evolutiva, decorre do fato das heurísticas utilizadas incorporarem a busca de serviços primitivos (serviços de alta coesão funcional, a partir dos quais serviços mais abrangentes são compostos), de classes de objetos parametrizáveis (das quais derivam instâncias de classes, geradas a partir de instanciação de parâmetros) e encapsuladas segundo os princípios da informação escondida, de forma a facilitar composição e instanciação.

Estes dois fatores, assim redefinidos, sintetizam as premissas que devem ser suportadas por metodologias de análise e projeto orientadas a objetos, e são utilizadas como padrão de comparação entre as metodologias apresentadas neste capítulo.

3.3 Descrição de Metodologias Orientadas a Objetos

Em função da existência de divergência de conceitos entre os metodologistas no paradigma orientado a objetos, como por exemplo classes de objetos que são referenciadas por alguns metodologistas simplesmente como objetos, a descrição de cada uma das metodologias se mantém fiel à conceituação definida pelos seus proponentes.

A maioria das metodologias apresentadas tem ferramentas de suporte em escala comercial. Como o objetivo desta análise comparativa é discutir as fraquezas e as vantagens de cada uma das metodologias apresentadas do ponto de vista de adequação representacional e poder heurístico, informações sobre as ferramentas existentes para suportar as metodologias descritas não foram consideradas relevantes para este trabalho.

3.3.1 Object-Oriented Analysis (OOA)

A metodologia OOA [Coad91], sugere a utilização das seguintes estratégias:

- identificação de objetos;
- identificação de estruturas;
- identificação de assuntos;
- definição de atributos;
- definição de serviços.

A estratégia de identificação de objetos estabelece diretrizes vagas para a identificação das classes de objetos da análise, baseada nos seguintes aspectos:

- como nomear: usar vocabulário padronizado para o domínio de problemas;
- onde procurar: utilizar observações, análises orientadas a objetos anteriores, leituras e protótipos;
- o que procurar: estruturas, sistemas, eventos, papéis executados, procedimentos peracionais, locais e unidades organizacionais;
- o que considerar: atributos de classes de objetos relevantes para o sistema, comportamento necessário ao sistema e algumas restrições (evitar classes com um objeto, considerar requisitos baseados no domínio do sistema independente da tecnologia utilizada).

A estratégia de identificação de estruturas simplemente propõe o uso de hierarquia e agregação, a partir de exemplos apresentados como ilustração.

A estratégia de identificação de assuntos não aborda a divisão de um sistema em subsistemas; assuntos são propostos apenas como orientação ao usuário para a visualização de grandes sistemas. A identificação de assuntos baseia-se em diretrizes de como selecionar e de como aperfeiçoar assuntos. A estratégia de seleção sugere que cada classe de objetos raiz de uma estrutura ou cada classe isolada torna-se um assunto. A estratégia de aperfeiçoamento sugere a caracterização de subdomínios do problema (com enfoque na agregação de entidades e não na decomposição funcional), a minimização de interdependências das hierarquias e referências entre classes de objetos e a minimização das interações correspondentes às mensagens trocadas entre as classes.

A estratégia de definição de atributos propõe a utilização de heurísticas capturadas da área de modelamento de dados, para caracterizar atributos, tais como: capturar todas as informações e estados associados a um objeto, associar o atributo à classe que ele descreve melhor, caracterizar cardinalidade em função dos relacionamentos. Adicionalmente, prevê a revisão da concepção das classes a partir da verificação de casos especiais:

- atributos que não se aplicam a todas as instâncias de classes;
- classe com um único atributo se torna atributo de outra classe;
- representação de cardinalidade de muitos para muitos através de uma nova classe.

A estratégia de definição de serviços prevê: a identificação dos estados da classe de objetos (utilizando diagrama de estados como representação), identificação dos serviços prestados/usados (descritos através de fluxogramas). A identificação de uso/prestação adota a utilização de CRCs (método que propõe a utilização de cartões nos quais são assinaladas responsabilidades e colaborações entre as classes de objetos em concepção) [Beck89]. Serviços são agrupados em duas categorias: simples (calcular, monitorar) e complexos (criar, conectar, acessar e liberar).

A passagem de análise para o projeto orientado a objetos é caracterizada a partir da adição de objetos de projeto (diálogo, administração de tarefas e de dados).

3.3.2 Object-Oriented Systems Analysis (OOSA)

A metodologia OOSA [Shlaer92] propõe as seguintes estratégias:

- definição do modelo de informação;
- definição dos ciclos de vida dos objetos;
- definição da dinâmica de relacionamentos;
- definição da dinâmica do sistema;
- definição dos modelos de processos;
- definição dos domínios e subsistemas.

A definição do Modelo de Informação utiliza um MER (Modelo de Entidade e Relacionamento) estendido com os relacionamentos especiais de herança e agregação para representar o Modelo de Informações. A obtenção das classes de objetos descritas no modelo de informação é sugerida a partir da indagação de classes de objetos do sistema pertencentes às seguintes categorias: objetos tangíveis (válvula, pacote), papéis (instrutor, válvula de isolação, eleitor), eventos (acidente, eleição, entrega), interações (conexão, contrato, interseção) e especificações (receita, composto químico). A descrição dos atributos destas classes diferencia três tipos de atributos: descritivos (Extrato.balanço, Gato.peso), nominativos (Extrato.número, Vôo.número) e referenciais (Gata.nome_proprietário, Extrato.identificador_cliente). Os atributos descritivos têm seus domínios descritos por enumeração de valores discretos, por regra de formação de valores ou por intervalo contínuo de valores. Os relacionamentos que interligam as classes de objetos englobam tipos especiais (generalização/especialização, agregação, associação representada como entidade referencial) e definem cardinalidade e opcionalidade.

A definição dos ciclos de vida dos objetos no Modelo de Objetos se restringe ao modelamento do subconjunto das classes de objetos descritas no Modelo de Informações que são identificadas como apresentando um comportamento dinâmico interessante. O Modelo de Objetos é composto de estados, eventos, transições para novo estado a partir da chegada de um evento e ações realizadas na chegada de um estado.

A definição da dinâmica de relacionamentos prevê o modelamento dos relacionamentos que evoluem no tempo (representando como objetos associativos, no Modelo de Estados) e dos relacionamentos envolvendo competição (exemplificado por caixas bancários e clientes).

A definição da dinâmica do sistema é representada através do Modelo de Comunicação de Objetos que provê uma representação da comunicação assíncrona (eventos) entre objetos de cada subsistema, complementando a descrição do Modelo de Estados que provê uma descrição dinâmica detalhada de cada objeto (com comportamento dinâmico identificado como interessante).

A definição do Modelo de Processos prevê a construção de um DFDA (DFD de Ação: similar ao DFD, com a diferença que mostra processos baseados em ações elementares, em vez da decomposição funcional do sistema) para cada ação associada a um dado estado de uma classe de objetos no Modelo de Objetos. Os processos representados no DFDA são classificados em quatro

tipos: de acesso (criação, destruição, leitura e escrita), gerador de evento (produz um evento como saída), transformação (computação ou transformação de dados) e teste (testa uma condição a partir da qual escolhe uma alternativa de fluxo de controle). A comunicação síncrona entre classes de objetos (realizada por processos de acesso) é descrita no Modelo de Acesso de Objetos.

A estratégia proposta para a construção de grandes sistemas de software prevê a organização dos mesmos em quatro domínios: de serviço, arquitetônico, de implementação e de aplicação. O domínio de serviço está associado a funções de suporte: processos de entrada e saída, alarmes, interface com o usuário, relatórios estatísticos e históricos. O domínio arquitetônico incorpora mecanismos de gerência de dados e controle. O domínio de implementação incorpora a definição de linguagens de programação, sistemas operacionais e bibliotecas de classes. O domínio de aplicação corresponde ao objetivo do sistema.

A passagem da análise para o projeto enfatiza o domínio arquitetônico em função da padronização de classes que suportam o comportamento dinâmico do sistema a partir de estados e transições.

3.3.3 Object-Oriented Design (OOD)

As estratégias da metodologia OOD [Booch91] são ordenadas em quatro etapas:

- identificação das classes de objetos;
- identificação da semântica das classes e objetos;
- identificação dos relacionamentos entre as classes e objetos;
- implementação das classes de objetos.

A etapa de identificação das classes e objetos corresponde à descoberta das abstrações chaves do espaço do problema e dos mecanismos (comportamento dos objetos) importantes. A obtenção das classes é realizada através de análise orientada a objetos, a partir dos tipos dos objetos envolvidos na terminologia do problema, para a qual são propostas técnicas de classificação (categorização clássica, agrupamento conceitual, teoria de protótipos) como base teórica. Estratégias vagas (determinação da fronteira do domínio, descoberta das abstrações do vocabulário do domínio, invenção de novas abstrações) são apresentadas para achar as abstrações chaves do domínio do problema. Heurísticas genéricas (acoplamento, coesão, suficiência, abrangência e primitividade) são citadas como métricas para qualidade das abstrações utilizadas.

A principal estratégia dos trabalhos anteriores em *OOD* (identificação de classes de objetos e mecanismos, a partir dos substantivos e verbos das descrições textuais), apresentada inicialmente em 1983 [Abbott83], defendida por Booch desde então (como heurística essencial da sua metodologia) com influência marcante na maioria das metodologias de análise e projeto orientadas a objetos que surgiram no início da década de 90, é desqualificada com sendo inadequada para a análise de problemas não triviais [Booch91].

A representação das classes utiliza Diagramas de Classes complementados por *templates*. Estes diagramas definem as classes existentes no projeto e os relacionamentos entre elas. Na descrição de grandes sistemas vários Diagramas de Classes são utilizados contendo classes ou categorias de classes (classes descritas em um nível mais alto de abstração). O prenchimento dos *templates* (utilizando representação textual para descrever classes, categorias de classes e mecanismos) é iniciado durante a identificação das classes e objetos e refinado nas etapas posteriores. A obtenção dos mecanismos mais importantes, para a qual é sugerida a técnica do quadro negro (trabalho ccoperativo no qual a solução para um dado problema é construída incrementalmente e oportunisticamente, ou seja, de forma não sistemática), é iniciada ainda neste primeira etapa e descrita através de Diagramas de Objetos (descrição da troca de mensagens entre objetos) complementados por *templates* (de objetos e mensagens).

A etapa de identificação da semântica das classes e objetos corresponde à definição do ciclo de vida dos objetos incluindo seus comportamentos a partir da construção de um *script* para cada objeto. Diagramas de Estados e seus *templates* complementares são criados para complementar os Diagramas de Classes. Estratégias vagas (mudança de nível na hierarquia; sugestões para distinguir a semântica de objetos, classes, operações modificadoras e seletoras através de seus nomes) são apresentadas para refinar as abstrações chaves do domínio do problema.

A etapa de identificação do relacionamento entre as classes e objetos corresponde à reorganização e à simplificação da estrutura de classes, à generalização dos mecanismos e à adoção de decisões de visibilidade entre as classes. A técnica proposta para a definição de visibilidade é a utilização de CRCs (cartões de responsabilidades e colaborações). Heurísticas adicionais como a Lei de Demeter, que restringe a um método de uma determinada classe o acesso aos métodos dos objetos internos que lhe são visíveis, com o objetivo de evitar a propagação de alterações do corpo de uma classe para classes que utilizam a interface, e a modelagem de agregação a partir de relações de uso são propostas para auxiliar a definição de relacionamentos e de visibilidade. A visibilidade entre dois objetos é definida a partir de três tipos: objetos do mesmo escopo léxico, objeto passado como parâmetro de outro objeto e objeto referenciado como campo de outro objeto.

A etapa de implementação de classes de objetos corresponde às decisões sobre a representação das classes, alocação de classes para programas e módulos para processadores e leva a um refinamento de notação já utilizada nas estratégias anteriores.

3.3.4 Object-Oriented Software Enginering (OOSE)

A diferença fundamental da proposta de Jacobson em relação à descrição dos eventos, como proposta pela análise essencial, é a ordenação de eventos sequencialmente correlatos sob a perspectiva dos usuários, denominados casos de uso.

A metodologia OOSE [Jacobson92] propõe encarar o desenvolvimento de software como um processo industrial, considerado como uma seqüência de processos (análise, construção e teste) e um processo auxiliar ao processo de construção (desenvolvimento de componentes).

O processo de Análise corresponde à criação do Modelo de Requisitos e do Modelo de Análise. O processo de Construção corresponde à criação do Modelo de Projeto e do Projeto de Blocos.

O Modelo de Requisitos é composto da Descrição dos Objetos do Domínio do Problema, da Descrição dos Casos de Uso e da Descrição da Interface.

O Modelo de Objetos do Domínio do Problema contém a identificação das entidades do domínio do problema e deve servir apenas como vocabulário base para a descrição dos casos de uso e da interfaces, evitando o transporte mecânico destes objetos para o projeto e a definição de suas operações precocemente. Para a obtenção dos objetos é sugerida a adoção de propostas de outros autores baseadas na análise da terminologia do problema.

Casos de usos constituem os elementos fundamentais da metodologia *OOSE*, a ponto de serem sugeridos como métrica para estimar as fases seguintes do desenvolvimento. O Modelo de Casos de Uso descreve, em linguagem textual, todas as formas de utilização do sistema do ponto de vista dos atores (tipos de usuários que desempenham papéis diferentes na utilização do sistema). A Figura 3 ilustra a especificação de um sistema a partir de seus casos de uso agrupados sob a perspectiva de dois atores distintos.

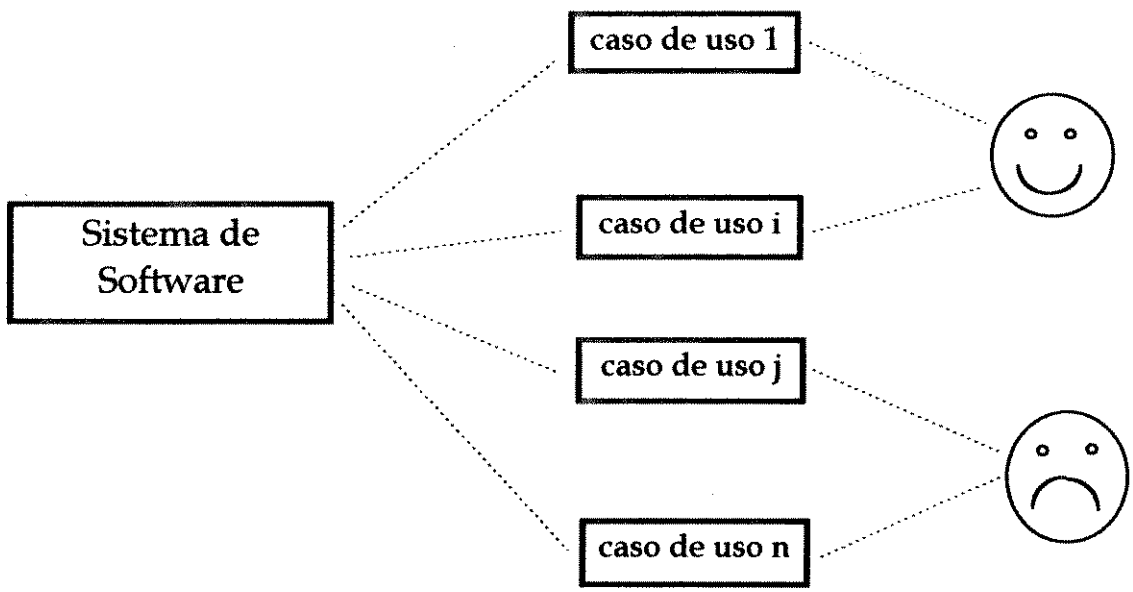

Figura 3: Especificação de Sistema de Software sob a Perspectiva de Atores e Casos de Uso

Um caso de uso corresponde ao curso completo dos eventos, iniciado por um ator, que especifica a interação entre o ator e o sistema. A Figura 4 mostra a representação de um caso de uso com composto por uma ordenação de eventos sequencialmente correlatos. A estratégia de obten-

ção dos casos de uso estabelece questões focando o ponto de vista dos atores (por exemplo, quais as principais tarefas de cada ator?).

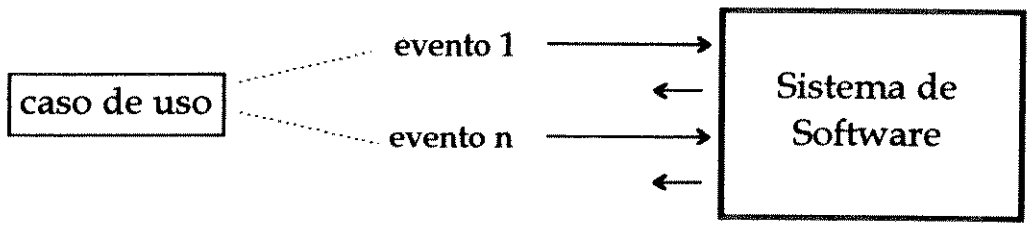

Figura 4: Composição de um Caso de Uso a partir de Eventos Sequencialmente Correlatos

Pequenas diferenças entre casos de usos não justificam a identificação de dois casos de uso distintos (por exemplo, em uma chamada local em uma central telefônica: assinante chamado responde e assinante chamado não responde). Essas variantes de casos de usos são denominadas extensões e são descritas a partir da definição de um fluxo variante aplicado ao caso de uso base. A Figura 5 ilustra a definição de extensões a partir de um caso de uso base.

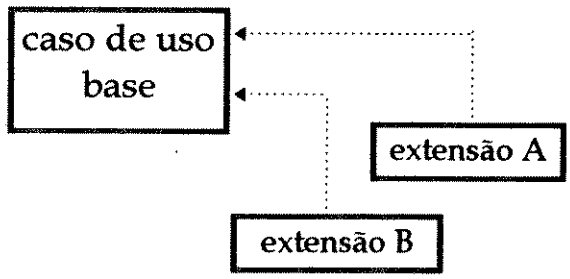

Figura 5: Definição de Extensões a partir de um Caso de Uso Base

Descrições da Interface em linguagem textual são propostas como complementares às descrições dos casos de uso, abrangendo a descrição da interface com os usuários e da interface com sistemas externos.

O Modelo de Requisitos é refinado a partir da identificação de casos de uso abstratos, que correspondem a partes similares utilizados na composição dos casos de uso. A Figura 6 ilustra a definição de casos de uso como refinamentos de casos de uso abstratos. Casos de uso abstratos podem ser diferenciados das extensões em função do seu acoplamento funcional. Extensões apresentam cursos de eventos independentes correspondentes à introdução de novos cursos ou à geração de cursos específicos; enquanto casos de uso abstratos apresentam cursos fortemente acoplados correspondentes à extração de seqüências comuns de diferentes casos de uso.

O Modelo de Análise descreve as classes de objetos decorrentes da análise, categorizadas em três tipos distintos de objetos (de interface, entidade e de controle) e os subsistemas que agrupam tais classes. A Figura 7 mostra a representação pictórica de cada um dos três tipos de objetos na metodologia OOSE.

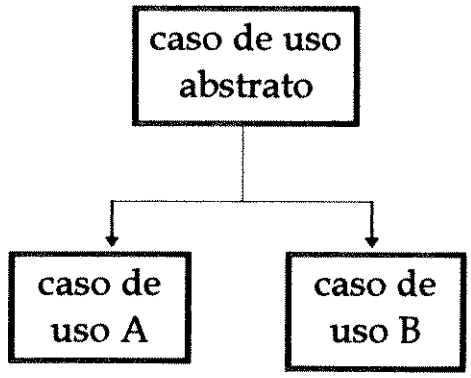

Figura 6: Casos de Uso refinados a partir de Casos de Uso Abstratos

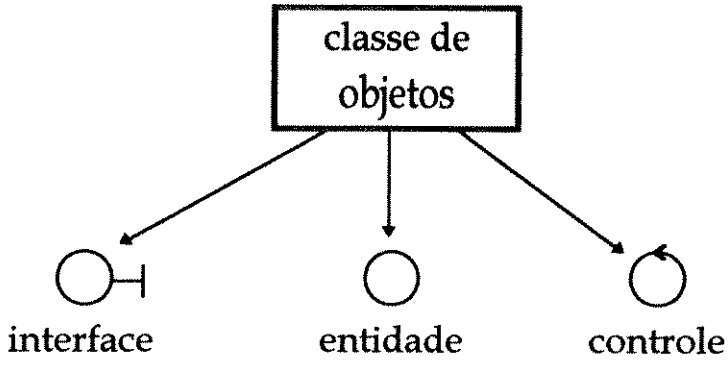

Figura 7: Representação Pictórica de cada um dos três Tipos de Objetos definidos em OOSE

Objetos de interface incorporam a funcionalidade, especificada nas descrições de casos de uso, dependente do meio ambiente do sistema. A identificação dos objetos de interface pode ser feita a partir da extração da funcionalidade específica de interface das descrições dos casos de uso, da demanda dos atores ou das descrições da interface. A estratégia de alocação de controle preconizada para os objetos de interface, denominada controle balanceado, prevê o controle separado do diálogo e da computação caracterizando um cenário no qual objetos de controle coordenam o sequenciamento entre as invocações de funções de diálogo e de funções computacionais.

Objetos entidade manipulam informações de longa duração com tempo de vida que ultrapassa, em geral, o término da execução dos casos de uso. Objetos entidade são obtidos a partir dos casos de uso (excluindo os objetos de interface), objetos do domínio do problema (utilizando os casos de uso como restrição à identificação dos objetos essenciais) e objetos gerados a partir de generalizações. A estratégia de obtenção dos objetos entidade sugere um modelo inicial utilizando apenas objetos de interface e entidade, levando em conta as operações típicas de um objeto entidade (criação e remoção do objeto entidade, informação de armazenamento e subseqüências que devem ser alteradas se o objeto entidade mudar).

Objetos de controle são derivados dos casos de uso a partir do comportamento não alocado a objetos de interface e entidade. São os mais efêmeros, existindo, em geral, somente durante um

caso de uso. A funcionalidade típica dos objetos de controle pode estar relacionada com comportamento associado a uma transação, com seqüências de controle específicas para um ou poucos casos de uso e com funcionalidade para isolar objetos entidade de objetos de interface.

O agrupamento das classes de objetos do Modelo de Análise em subsistemas é baseado em acoplamento funcional, considerando as seguintes heurísticas para determinar se duas classes de objetos pertencem ao mesmo subsistema: mudanças em uma classe de objetos acarretam mudanças em outra classe, classes se comunicam com o mesmo ator, duas classes são ambas dependentes de uma terceira classe e uma classe realiza diversas operações em outra classe. Os passos para a divisão em subsistemas são os seguintes: identificar subsistemas secundários, colocar uma classe de controle em cada subsistema (como primeira tentativa) completando com as classes de interface e entidade fortemente acopladas e separar em outro subsistema uma classe utilizada por mais de um subsistema. Para projetos de grande porte podem ser considerados critérios de divisão adicionais: divisão dos subsistemas entre os grupos de desenvolvimento, na qual cada subsistema corresponde a um nó lógico em um sistema distribuído, e o mapeamento de um determinadado projeto existente, em uso na composição do projeto de grande porte, em um único subsistema.

O Modelo de Projeto é composto das seguintes atividades: identificação do ambiente de implementação, identificação dos objetos de projeto e da construção dos Diagramas de Interação de Objetos. A identificação do ambiente de implementação pode levar à definição de novas classes de objetos com o objetivo de encapsular a dependência do meio ambiente.

Os objetos obtidos no Modelo de Análise, a partir dos casos de uso, tornam-se a base do Modelo de Projeto. A identificação dos objetos de projeto adota a estratégia inicial de transformar cada classe de objetos de análise em um objeto de projeto, denominado bloco. Alterações em relação aos blocos originários das classes da análise devem ser justificadas e documentadas.

Na construção dos Diagramas de Interação são utilizadas heurísticas para a definição das mensagens trocadas entre as classes de objetos tais como limitação do número de parâmetros e compatibilização de nomes de mensagens entre os vários casos de uso. A estrutura de controle das operações realizadas pelas classes de objetos pode ser centralizada (operações que podem mudar de ordem, quando novas operações podem ser inseridas) ou descentralizada (mais usual em orientação a objetos).

No projeto de bloco é definido um Diagrama de Transição de Estados por bloco. Em geral classes de objetos de controle têm tendência a serem controladas por estados (ou seja, com forte ligação temporal entre o estímulo recebido e o estímulo para o qual está preparada para receber), diferentemente das classes de objetos entidade que geralmente são controladas por estímulo (ou seja, realizam a mesma tarefa sempre que recebem o mesmo estímulo).

A passagem da análise para o projeto baseia-se em um desenvolvimento incremental com várias interações e com enfâse na identificação do ambiente de implementação.

O processo de Desenvolvimento de Componentes tem as seguintes fontes de candidatos a componentes reusáveis: objetos entidade gerais, objetos de interface, objetos de controle com funções gerais, associação entre objetos, tipos gerais e encapsulamento do ambiente de implementação. Critérios para construção de bons componentes reusáveis são propostos: redução do número de parâmetros, não utilização de parâmetros para caracterizar opções, consistência de nomes e generalidade de abstração. Métricas (tamanho, complexidade e frequência de reuso) são propostas para avaliar a adequação do reuso de componentes.

3.3.5 Object Modeling Technique (OMT)

Na metodologia OMT [Rumbaugh91] modelos de análise e projeto são perspectivas diferentes mas complementares de documentação do sistema. Os modelos da análise são a visão externa para o cliente (base para elicitar os verdadeiros requisitos do sistema) e devem, portanto, incorporar informação significativa para o mundo real. Os modelos de projeto devem focar a implementação no computador, endereçando detalhes de baixo nível. Na prática pode haver considerável sobreposição entre análise e projeto (grande parte dos modelos da análise implementada sem alterações).

A análise, cujo ponto de partida é uma descrição do problema em linguagem natural, endereça três aspectos dos objetos: estrutura estática (modelo de objetos), seqüência de interações (modelo dinâmico) e transformações de dados (modelo funcional). A classe de problemas a ser modelada define a utilidade de cada modelo: modelo de objetos (estático), modelo dinâmico e modelo funcional. No modelo de objetos quase todos os problemas derivam do mundo real. No modelo dinâmico os problemas estão associados a interações e temporizações (interface com usuário e problemas de controle); quando a lógica correta não depende somente da seqüência de interações mas também, dos tempos exatos das interações o problema caracteriza-se como de tempo real. No modelo funcional a computação é mais significativa (compiladores e cálculos de engenharia). A Figura 8 ilustra os Modelos de Análise da metodologia OMT.

O modelo de objetos, composto por diagrama de modelo de objetos e dicionário de dados, precede os modelos dinâmico e funcional, porque a estrutura estática é melhor definida, menos dependente de detalhes de implementação, mais estável à medida que a solução evolui e mais fácil de entender. O modelo de objetos é obtido a partir dos seguintes passos: identificação de classes de objetos; construção de um dicionário; adição de associações (dependências entre classes); adição dos atributos das classes; organização e simplicação das classes utilizando herança; teste de caminhos de acesso usando cenários (iterando os passos anteriores, se necessário); e agrupamento de classes em módulos (baseado em alto acoplamento e em funções relacionadas).

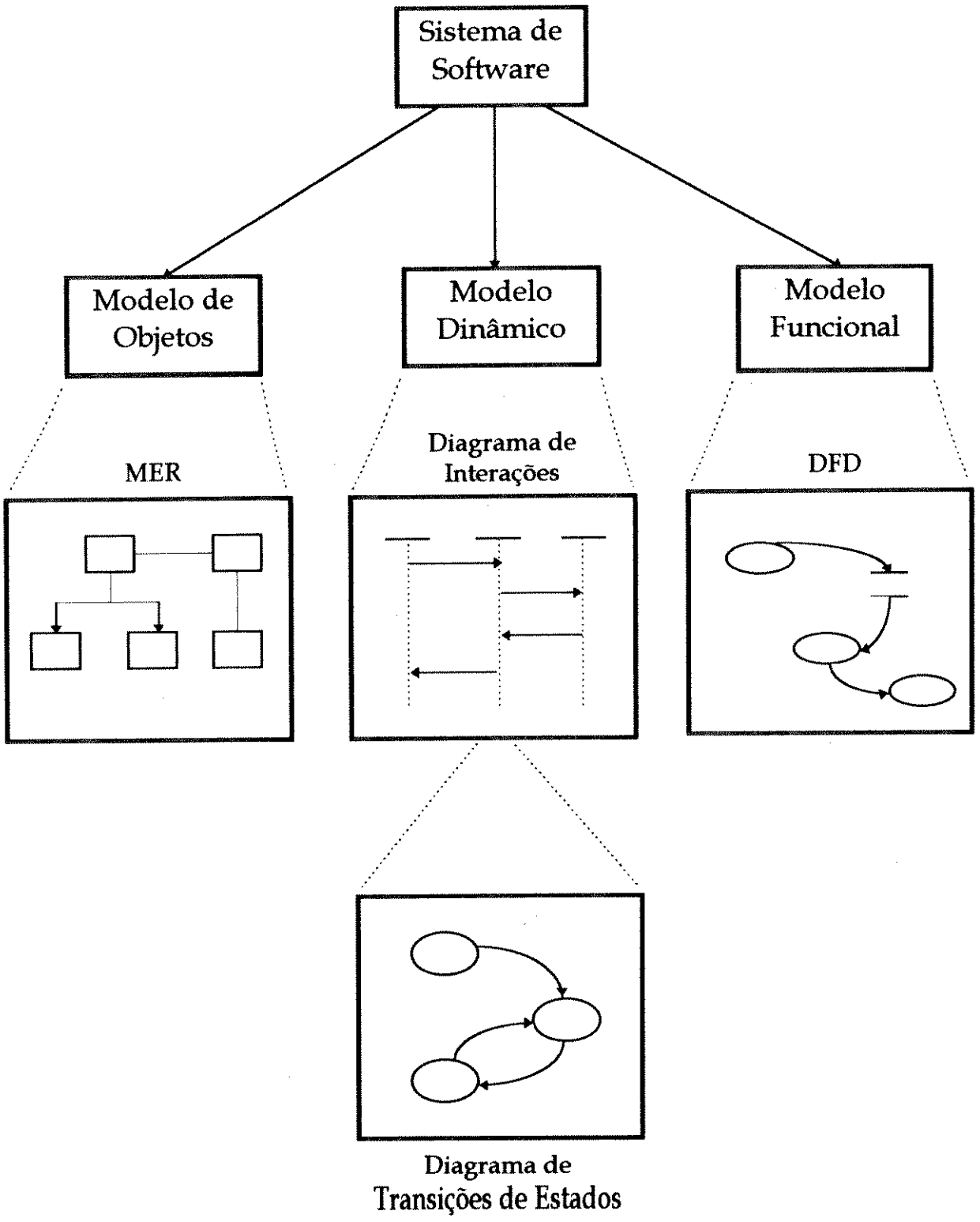

Figura 8: Modelos de Análise da Metodologia OMT

A identificação das classes de objetos é composta das atividades de extração das classes candidatas dos substantivos da descrição do problema e da eliminação das classes espúrias. Nem todas as classes candidatas são explícitas (algumas são implícitas do domínio da aplicação ou de conhecimento geral) e a herança deve ser postergada (para não distorcer a estrutura de classes com noções pré-concebidas). São consideradas classes espúrias as classes redundantes, irrelevantes (ao domínio do problema) ou vagas; as construções de implementação e os atributos, operações ou papéis, cuja caracterização é dependente do contexto.

Na construção do dicionário, um parágrafo preciso deve ser utilizado para descrever cada classe contendo o escopo da classe do problema, asserções e restrições sobre seus membros ou em utilização.

A adição de associações é feita a partir da extração de candidatas das frases verbais da descrição do problema, eliminando-se as associações espúrias.As associações com classes eliminadas, irrelevantes ou de implementação são consideradas espúrias. Associações derivadas são redundantes e associação ternárias devem ser decompostas em binárias (ocasionalmente tal decomposição leva à perda de informação).

Os atributos adicionados às classes de objetos provêem de substantivos seguidos de frases possessivas mas são menos frequentes na descrição do problema. Atributos espúrios (tais como valores internos, valores dependentes de contexto particular e atributos de associação) devem ser eliminados. Atributos discordantes podem indicar necessidade de dividir uma classe em duas.

O modelo dinâmico é montado a partir da identificação de cenários de seqüências típicas de iterações. Para cada cenário identificado é construído um diagrama de interações. Diagramas de fluxo de eventos (mostram os eventos entre as classes de objetos, sem considerar seqüência, incluindo eventos de todos os cenários, inclusive eventos de erro) entre grupos de classes do sistema (equivalentes a módulos) também devem ser construídos. A consistência e a completitude dos eventos partilhados entre diagramas de estados deve ser verificada.

O modelo funcional, composto por DFDs e restrições, mostra a dependência funcional entre valores e funções que os associam, excluindo seqüências de operações, decisões de operação e estrutura de objetos. A descrição de cada função de um DFD pode ser declarativa (relacionando entradas e saídas) ou procedimental (descrita por um algoritmo que pode ser substituído por outro equivalente na implementação). A forma de representação de cada função pode ser linguagem natural, equações matemáticas ou pseudo código. Restrições caracterizadas por dependências funcionais entre objetos não associados a dependências entre entradas e saídas, ou por pré-condições e pós-condições (com procedimentos ativados por decorrência de violações especificados nos modelos dinâmico e funcional). Critérios de otimização podem ser expressos a através da definição de valores para maximizar, minimizar ou otimizar ou de decisões de compromisso em relação a conflitos.

A análise é concluída com o refinamento de seus três modelos, iterando os seguintes passos:

- acrescente ao modelo de objetos, as operações mais importantes decorrentes das funções do modelo funcional;
- compare os três modelos com a descrição inicial do problema, conhecimento do domínio;
- teste os modelos usando cenários mais detalhados (incluindo condições de erro e variações sobre os cenários básicos).

O projeto de sistemas é o responsável pela estrutura da arquitetura básica do sistema, bem como pelas decisões estratégicas de alto nível. Não acrescenta heurísticas específicas ao paradigma orientado a objetos.

O projeto dos objetos guia o detalhamento da análise, ou seja, da passagem da orientação do mundo real do modelo de análise para o mundo computacional da implementação, através dos sequintes passos:

- acrescente operações no modelo de objetos: para cada um dos processos do modelo funcional e para cada um dos eventos do modelo dinâmico;
- projete algorítmos para implementar as operações (que minizem o custo da implementação, definindo novas classes internas e operações quando necessário);
- otimize caminhos de acesso a dados (acrescente associações redundantes para minimizar custo de acesso, rearranjando computação para maior eficiência e armazenando valores derivados para evitar recomputar expressões complicadas);
- ajuste a estrutura de classes para aumentar a utilização de herança;
- projete a implementação de cada uma das associações como objeto distinto ou como um atributo adicional (em uma das classes de objetos da associação ou em ambas);
- empacote classes e associações em módulos.

3.3.6 Object-Oriented Software Design (OOSD)

A metodologia OOSD [Wirfs-Brock90] baseia-se nas seguintes etapas: achar as classes de objetos, definir responsabilidades, definir colaborações, definir hierarquias e definir subsistemas.

As classes de objetos são obtidas a partir de uma descrição textual do problema. Os substantivos sugerem as classes candidatas (sentenças na voz passiva implicam substantivos) e os adjetivos podem sugerir diferentes objetos ou diferentes usos do mesmo objeto. A escolha das classes candidatas pode ser auxiliada pelo modelamento de objetos físicos (por exemplo disco, impressora), das entidades conceituais que formam uma abstração coesa (por exemplo janela e arquivo) e de interfaces (com usuários, com outros programas e com o sistema operacional). Palavras utilizadas para o mesmo conceito devem ser unificadas.

Responsabilidades correspondem aos serviços públicos providos por uma classe de objetos. Responsabilidades são identificadas a partir de cenários da especificação de requisitos e de nomes de classes que sugerem responsabilidade. Atribuir responsabilidades requer a observação de

determinados critérios: a inteligência dos sistema deve ser distribuída, as responsabilidades devem ser estabelecidas da forma mais geral possível, o comportamento de uma classe de objetos deve estar relacionado com a informação que ela trata, a manutenção de informação não deve ser partilhada (o que levaria à duplicação), responsabilidades devem ser partilhadas entre classes de objetos relacionadas. Responsabilidades adicionais podem ser identificadas através do exame de relacionamentos entre classes (especialização, generalização, agregação). Para cada classe de objetos é alocado um cartão no qual são registrados o nome da classe e uma frase para cada responsabilidade.

Colaborações correspondem aos requisitos que uma determinada classe de objetos cliente estabelece para as suas classes servidoras. Colaborações podem ser obtidas a partir dos relacionamentos entre as classes. O relacionamento é_parte_de indica que a classe composta tem a responsabilidade de gerenciar as partes ou manter relacionamentos específicos sobre as partes. Os relacionamentos tem_conhecimento_de e depende_de implicam que a classe pode ser alterada em função de alterações sobre as quais tem conhecimento ou das quais depende. Ao cartão correspondente a uma classe cliente devem ser associadas responsabilidades, e a cada responsabilidade os nomes das classes que colaboram como servidoras.

A definição de hierarquias entre as classes deve observar os seguintes critérios: agrupar responsabilidades utilizadas pelos mesmos clientes, maximizar a coesão das classes e minimizar o número de contratos (um contrato é definido como o conjunto de requisitos que um cliente pode impor a um determinado servidor).

Subsistemas são grupos de classes (ou de outros subsistemas) que colaboram entre si para suportar um conjunto de contratos (suportados por classes que provêem serviços externos ao subsistema). Colaborações entre subsistemas devem ser minimizadas. Para cada subsistema é alocado um cartão no qual são enumerados os serviços fornecidos, associados à classe que os provê.

3.3.7 Object-oriented Systems Analysis (OSA)

Segundo os proponentes da metodologia OSA [Embley92], analistas não são matemáticos e não se pode esperar que criem documentos utilizando cálculo de predicados de primeira ordem ou qualquer outra notação formal; analistas devem ser livres para fazer anotações, esquematizar idéias e evoluir modelos sem a tarefa opressiva de transcrever idéias em matemática formal. Mas embora a notação de OSA não seja formal, seus conceitos estão baseados em definições formais de dados e modelamento de comportamento, o que permite testar a integridade do modelo utilizado e assegurar-lhe uma interpretação consistente.

A metodologia OSA define as seguintes atividades:

- construção do Modelo de Relacionamento de Objetos;
- construção do Modelo de Comportamento de Objetos;

- refinamento dos modelos para grandes sistemas através da definição de visões e de componentes de alto nível;
- construção do Modelo de Interação de Objetos;
- construção do Modelo de Integração.

O Modelo de Relacionamento de Objetos representa o relacionamento entre as classes de objetos do sistema acrescidos de restrições e anotações. Restrições de relacionamentos incluem: número de vezes que um objeto pode participar de um relacionamento (mínimo e máximo) e número de relacionamentos no qual um objeto pode aparecer. Relacionamentos especiais incluem generalização ("é um"), agregação ("é parte de") e associação ("é membro de"); classes especiais incluem a classe *singleton* (para modelar classes com um único objeto) e a classe relacional (para modelar um conjunto de relacionamentos como uma classe de objetos). Restrições mais gerais são expressas através de regras.

O Modelo de Comportamento de Objetos é composto dos seguintes elementos: estados, *triggers* (eventos e condições que ativam transições), transições (processos de mudança de um estado para o outro) e ações (realizadas durante as transições). Eventos podem ser modelados como classes de objetos (Chegada à Estação) e *triggers* podem ser compostos por conjunções e disjunções de condições. Exceções (eventos ou condições que não fazem parte do comportamento normal do sistema) são representadas no diagrama entre um estado e um *trigger*; restrições de tempo podem ser modeladas por exceções de tempo anormal.

Visões são implosões de parte dos componentes de um modelo em componentes de alto nível que podem ser utilizadas para simplificar os modelos.

Visões definidas a partir de classes de objetos são provenientes de uma classe dominante (classe mais representativa dentre as classes implodidas) ou correspondem a uma classe independente (nova classe representa a implosão). A inclusão de um relacionamento (na representação da visão) impõe também a inclusão de suas restrições.

Visões definidas a partir de relacionamentos correspondem a relacionamentos de alto nível que agrupam classes de objetos, conjuntos de relacionamentos e restrições; em geral incluem classes, mas podem implodir somente relacionamentos paralelos. A derivação automática das restrições do relacionamento de alto nível (a partir das restrições dos relacionamentos implodidos) algumas vezes é trivial, mas frequentemente é difícil e, ocasionalmente, é impossível. Não há vantagens na introdução do conceito de relacionamento dominante, uma vez que o nome do relacionamento de alto nível é independente dos nomes dos relacionamentos implodidos.

Visões definidas a partir de uma rede de estados correspondem a um estado de alto nível que agrupa estados, transições e restrições (em geral inclui transições, mas pode implodir somente estados entre duas transições). Pode ocorrer redução do número de fluxos incidentes no estado de alto nível e, neste caso, os fluxos de uma transição externa ao estado de alto nível para vários de seus subestados são convertidos em apenas um fluxo com a implosão. A inclusão de uma transição na representação das visões impõe também a inclusão de seus estados subsequentes e

condições de ativação. A submissão a um estado dominante é o mesmo que a submissão a um estado independente, exceto pelo reuso do nome (como em relacionamentos).

Visões definidas a partir de transições correspondem a uma transição de alto nível que agrupa transições, estados e restrições; em geral inclui estados, mas pode implodir somente transições paralelas entre estados. As transições de um estado implodido devem fazer parte da implosão para que não ocorram transições adjacentes ilegais. Transições de alto nível podem agupar exceções em uma exceção de alto nível.

O conceito de visões, como proposto pela metodologia OSA, incorpora conceitos de outras propostas (classes de objetos dominantes e estados de alto nível), com a diferença de ser estendido para todos os modelos.

Componentes de alto nível podem ser utilizados como blocos construtivos dos modelos em uma abordagem *top-down* (Modelagem de Alto Nível), em vez de implodir componentes de nível mais baixo em uma abordagem *bottom-up* (visões). Uma classe de objetos de alto nível designa um submodelo complexo e, embora geral, não deve ser utilizada para representar uma classe de objetos relacional ou uma agregação (a representação específica tem uma semântica mais precisa).

O Modelo de Interação de Objetos representa a troca de mensagens entre objetos acrescida de várias construções adicionais. Diagramas podem utilizar repositórios intermediários para representar interações assíncronas do tipo mensagem de armazenamento entre objeto e depósito e mensagem de recuperação entre depósito e objeto. Restrições de origem e destino podem complementar a especificação de uma mensagem. Interações podem ser especificadas com unidirecionais (solicitação e resposta em mensagens separadas) ou como bidirecionais. Interações podem utilizar atividades especiais como acessar, modificar, remover, destruir, acrescentar e criar. Restrições de tempo (< 10 ms) ou gerais (selecionar a ordem mais antiga) podem ser associadas às interações. Detalhamentos de interações internas a um objeto podem ser mostradas e objetos especializados herdam interações dos objetos gerais.

Os modelos podem ser integrados segundo os esquemas de comparação de diagramas (através da identificação de semelhanças e conflitos), de conformação de diagramas e de *merge* de diagramas (*merge* de dois componentes conceitualmente idênticos). Na conformação de diagramas conflito de nomes são resolvidos através da alteração de nomes (junção ou separação) e conflitos estruturais são resolvidos através da inserção, remoção, troca ou combinação de elementos.

3.4 Análise das Metodologias Descritas segundo as Premissas Estabelecidas

Do ponto de vista de adequação representacional, metodologias devem suportar:

- representação consistente e não ambígua dos requisitos do sistema;
- continuidade representacional entre os artefatos gerados nos vários níveis de abstração.

Do ponto de vista de poder heurístico, metodologias devem prover a capacidade de guiar a decomposição do sistema em artefatos que:

- atendam aos requisitos do sistema;
- facilitem reusabilidade;
- restrinjam propagação de alterações por decorrência de manutenção.

O suporte das metodologias às premissas acima pode ser classificado como: efetivo, parcial e insuficiente. A Figura 9 mostra uma tabela discriminando o grau de suporte às premissas estabelecidas para cada uma das metodologias descritas (o grau insuficiente está abreviado).

	OOA	OOSA	OOD	OOSE	OMT	OOSD	OSA
representação de requisitos	insuf.	insuf.	insuf.	efetivo	parcial	insuf.	insuf.
continuidade representacional	insuf.	parcial	insuf.	parcial	parcial	insuf.	efetivo
atendimento aos requisitos	insuf.	insuf.	insuf.	parcial	parcial	parcial	insuf.
reuso de artefatos	insuf.	parcial	insuf.	efetivo	insuf.	insuf.	insuf.
restrição à propagação de alterações	insuf.	insuf.	insuf.	efetivo	insuf.	insuf.	insuf.

Figura 9: Grau de suporte das Metodologias Descritas às Premissas Estabelecidas

Do ponto de vista da representação dos requisitos, o suporte fornecido pelas metodologias pode ser classificado como:

- efetivo: na metodologia OOSE;
- parcial: na metodologia OMT;
- insuficiente: nas demais metodologias.

Na metodologia OOSE o suporte é efetivo em função da definição de casos de uso (que expressam todas as formas de utilização requeridas ao sistema) e do vocabulário do domínio (que expressa as entidades do sistema).

Na metodologia OMT o suporte é parcial pelo fato dos requisitos serem representados tardiamente durante a análise e, portanto, de forma mais detalhada do que deveria ocorrer durante a especificação de requisitos, misturando solução com entendimento do problema. Além disso, os modelos utilizados em OMT capturam comportamento dinâmico, estático e funcional a partir da elaboração de uma descrição textual que não se justifica, devido à inadequação da linguagem natural para representações concisas e não ambíguas.

Do ponto de vista de continuidade representacional, o suporte fornecido pelas metodologias pode ser classificado como:

- efetivo: na metodologia OSA;
- parcial: nas metodologias OOSE, OOSA e OMT;

- insuficiente: nas demais metodologias.

Na metodologia OSA o suporte é efetivo devido à interligação de diferentes níveis de abstração através da definição dos conceitos de visões e componentes de alto nível.

Na metodologia OOSE o suporte é parcial porque apesar da utilização dos casos de uso representar um elo de ligação entre especificação de requisitos, análise e projeto, a utilização de linguagem natural na especificação de requisitos e análise não é adequada sob o ponto de vista de representação.

Na metodologia OOSA o suporte é parcial pois abrange somente os objetos com comportamento dinâmico relevante; embora seja bem definido pois, para cada classe de objetos do Modelo de Objetos, é criado um Modelo de Estados e, para cada ação do Modelo de Estados, é criado um DFD.

Na metodologia OMT o suporte é parcial, uma vez que os três modelos são definidos de forma que apenas correlações parciais entre os elementos de cada um dos modelos são identificadas a posteriori.

Do ponto de vista da capacidade de guiar a decomposição do sistema em artefatos que atendam os requisitos, o suporte fornecido pelas metodologias pode ser classificado como:

- parcial: nas metodologias OOSE, OMT e OOSD;
- insuficiente: nas demais metodologias.

Na metodologia OOSE o suporte é parcial porque a utilização de casos de uso se presta basicamente para sistemas reativos. As heurísticas para distinguir os três tipos de classes de objetos, embora gerais, são aplicadas aos casos de uso e necessitam de um refinamento maior.

Na metodologia OOSD o suporte é parcial porque as heurísticas propostas para alocação de responsabilidades e colaborações, embora gerais, não apresentam um compromisso bem estabelecido com os requisitos do problema.

Na metodologia OMT o suporte é parcial porque, embora cenários sejam considerados para caracterização do modelo dinâmico e estratégias detalhadas sejam apresentadas para filtragem das classes de objetos (obtidas a partir da descrição textual), os demais passos apresentados na análise são vagos como um conjunto de heurísticas para decomposição do sistema.

Na metodologia OOSA o suporte é insuficiente porque o refinamento das classes de objetos (cuja escolha inicial, baseada na terminologia do problema e em estratégias de modelagem de dados, é crítica, uma vez que os passos seguintes se baseiam exclusivamente nestas classes) é realizado separadamente para cada classe de objeto (restringindo-se apenas àquelas com comportamento dinâmico relevante). A interação entre os objetos não é utilizada para refiná-los mas, somente apresentada como produto final.

Na metodologia OOD o suporte é insuficiente porque se constitui de um conjunto de técnicas conhecidas (cartões de responsabilidades e colaborações, *blackboard, script*) ou descritas superficialmente (análise de domínio, formas de classificação), sem organização adequada em uma seqüência de estratégias nem uma associação explícita com a extensa notação proposta para repre-

sentação do projeto; distingue-se apenas um ponto interessante: a cooperação entre as classes de objetos é exercitada desde o início do projeto, auxiliando o refinamento das mesmas.

Na metodologia OOA o suporte é insuficiente porque as estratégias apresentadas são vagas ou capturadas de outras fontes (por exemplo, modelagem de dados e cartões de colaborações e responsabilidades) e os serviços prestados/usados pelas classes de objetos não são utilizados para identificá-los (uma vez que a identificação dos serviços é a última atividade).

Do ponto de vista da capacidade de conceber artefatos reusáveis nos vários níveis de abstração, o suporte fornecido pelas metodologias pode ser classificado como:

- efetivo: na metodologia OOSE;
- parcial: na metodologia OOSA;
- insuficiente: nas demais metodologias.

Na metodologia OOSE o suporte é efetivo pois é coberto em todas as fases do desenvolvimento: identificação de extensões e de casos de uso abstratos durante a especificação de requisitos, estratégias para definição dos estímulos dos diagramas de interação (limitação de parâmetros, identificação de estímulos similares e homogeinização) durante a fase de projeto, e de heurísticas para identificar candidatos e construir componentes reusáveis com objetivo de reutilização de código.

Na metodologia OOSA o suporte é parcial porque se restringe ao modelamento de sistemas de grande porte, a partir de quatro domínios.

Do ponto de vista da capacidade de restringir a propagação de alterações, o suporte fornecido pelas metodologias pode ser classificado como:

- efetivo: na metodologia OOSE;
- insuficiente: nas demais metodologias.

Na metodologia OOSE o suporte é efetivo devido à criação do conceito do objeto de controle, com objetivo de obter localidade nas mudanças, sem onerar desta forma a concepção dos objetos entidade com controle que não lhes é natural.

3.5 Proposta da Metodologia Especificação Unificada

No paradigma orientado a objetos, a distância representacional entre análise e projeto é bem menor. Existe uma considerável sobreposição entre os modelos de análise e projeto, levando à implementação de muitas partes do modelo de análise sem mudanças [Rumbaugh91].

Em geral, encontram-se na literatura as seguintes distinções entre análise e projeto:

- análise corresponde à descrição do problema e dos requisitos do usuário, enquanto projeto corresponde à construção da solução (sistema de software que satisfaz os requisitos);
- na análise descreve-se o *what*, e no projeto descreve-se o *how*.

A primeira distinção é subjetiva. Na realidade, as classes de objetos que aparecem na análise correspondem à visão do analista sobre o domínio do problema. Soluções de projeto têm motivações distintas, associadas à necessidade de representação e otimização de recursos decorrentes do mundo computacional, tais como:

- encapsulamento do meio ambiente (sistema operacional e banco de dados);
- duplicação de informação para otimizar acesso;
- utilização de tipos abstratos de dados para representar relacionamentos com cardinalidade múltipla.

A distinção entre *what* e *how* pode ser achada em qualquer nível hierárquico de um modelo de sistema. Todo *how* é a solução para um *what* de mais alto nível, e todo *what* é novamente parte de um *how* de nível mais alto. Uma classe de objetos *stack*, por exemplo, aparece na fase de projeto, independentemente do seu *what* e *how* [Hoydalsvik93].

No ciclo de desenvolvimento de um sistema de software, distintos modelos são utilizados para especificar diferentes níveis de abstração; o detalhamento da especificação de requisitos leva às especificações de análise, projeto e implementação. Sob este ponto de vista, um compilador de uma linguagem de programação orientada a objetos traduz uma especificação da implementação (contendo representações como herança, *templates*, passagem de parâmetros e recursão) para uma especificação em um nível de abstração bem mais baixo (linguagem de máquina). A consistência e a manutenção do vínculo entre as especificações utilizadas nos diversos níveis depende do mapeamento que ocorre entre as várias especificações que representam o sistema de software ao longo do seu ciclo de vida.

Os conceitos suportados pelas linguagens de programação orientadas a objetos, associados às heurísticas de reuso (fatoração, parametrização e componentes primitivos), suportam a criação de camadas de serviços em um dado domínio, a partir das quais o nível de programação passa a ser bem mais alto, facilitando o mapeamento entre o domínio do sistema e a sua implementação. No entanto, como diferentes formas de representação são utilizadas para descrever requisitos, análise, projeto, e implementação, a manutenção desta documentação tem levado muitos desenvolvimentos de sistemas de grande porte que incorporam software (como sistemas de área de telecomunicações) ao seguinte cenário (do qual participam muitos programadores produzindo muitas linhas de código):

- a prática corrente de produzir-se a documentação (ou grande parte da documentação) ao final do projeto, para a liberação do software;
- necessidade de alto grau de disciplina para a produção e a manutenção de documentação que reflita a real implementação, em contraste com o comportamento padrão dos programadores;
- a ausência de uma documentação coerente que acompanhe as diversas fases do projeto tem levado a contínuas e indesejáveis realimentações entre as fases, dificultando a ação da gerência e ocasionando implementações que demonstram na fase de integração um grau muitas vezes inadmissível de inconsistência e duplicação de código.

Da captura adequada dos requisitos iniciais do sistema (que acabam sendo detalhados durante a decomposição) aos processos de mapeamento entre os diversos níveis de abstração, vários pontos merecem ser questionados:

- que elementos compõem uma especificação de requisitos e como expressá-la concisamente?
- como mapear os requisitos nas classes de objetos (com seus atritutos e métodos) de análise, de projeto e de implementação, procurando explicitar apenas o mapeamento necessário?
- como manter o vínculo com os requisitos em cada um destes mapeamentos?
- existe equivalência entre construções utilizadas na especificação de requisitos e nas especificações seguintes (de análise, projeto e implementação)?
- é possível recuperar visões distintas das classes de objetos a partir de uma única representação, evitando a manutenção de vários níveis de modelos e eliminando a quebra de continuidade representacional (impostas por diferentes modelos para cada um dos níveis de abstração)?
- é possível, enfim, especificar um sistema de software como um conjunto de objetos cooperando, em todos os níveis de abstração (desde os requisitos até a sua implementação)?

Como alternativa de resposta aos esses pontos de vista, este trabalho apresenta uma metodologia de desenvolvimento de sistemas de software no paradigma orientado a objetos, denominada Especificação Unificada, na qual se propõe:

- uma única linguagem de representação para as especificações de requisitos, análise e projeto, com suporte à recuperação de visões dos modelos que em geral são utilizados para representá-las (diagramas de interações, modelos de entidades e relacionamentos, e diagramas de estados);
- um corpo de heurísticas para guiar a decomposição do sistema em todos os seus níveis de abstração e contemplando classes de problemas que incluem sistemas reativos e não reativos, com o objetivo de gerar artefatos de software que facilitem ações de manutenção corretiva e evolutiva.

Os temas abordados nos próximos capítulos são ilustrados com exemplos obtidos a partir da aplicação da metodologia Especificação Unificada em um depurador simbólico para a linguagem *CHILL* (quando a referência à procedência do exemplo for omitida, entenda-se que o exemplo foi extraído do sistema de software *CHILL Symbolic Debbuger* (*CSD*) [Chaves94]). Este sistema foi escolhido como exemplo porque tem uma interface fortemente interativa e sua decomposição a partir dos eventos da interface não resulta apenas em serviços primitivos.

CHILL é uma linguagem de alto nível concorrente normatizada por um comitê internacional para a implementação de sistemas de telecomunições. O *CSD* é um componente de software do Ambiente de Programação *CHILL* do CPqD-TELEBRÁS (APCC). A principal ferramenta de software do APCC é um compilador *CHILL*, cujo *front-end* foi adquirido e adaptado e cujo *back-end* e *run-time library* foram desenvolvidos no CPqD-TELEBRÁS. O APCC foi utilizado no desen-

volvimento da Central Telefônica Digital de Grande Porte TRÓPICO-RA (com cerca de dois milhões de linhas de código e mais de 150 centrais instaladas no sistema TELEBRÁS, em todo o país) e atualmente é utilizado na manutenção e evolução do TRÓPICO-RA. *CSD* é um depurador simbolico remoto para programas implementados na linguagem *CHILL*, ou seja executa em *workstation* sob o sistema operacional UNIX e permite depurar programas executando remotamente em sistema operacional dedicado.

CSD possui uma interface gráfica amigável [Chaves94] que permite, por exemplo, o controle da execução, a colocação de *breakpoints*, a verificação e alteração de variáveis correspondentes às *tasks* em execução no programa *CHILL*. O foco de atenção do usuário pode ser alterado para qualquer uma das *tasks* em execução ou para qualquer procedimento (ou processo) da pilha de execução, com implicação na atuação dos comandos de controle de execução e na informação mostrada na interface.

Durante uma parada de programa (controlada pela inserção de *breakpoints* ou pela utilização de comandos de execução do tipo *Next, Step, ExecuteToHere* e *Stop*), a execução de todas as *tasks* é congelada. Dessa forma, o *CSD* consegue tratar a concorrência dos programas implementados em *CHILL* através de uma implementação não concorrente. Este artifício simplifica a implementação do *CSD* e permite ao usuário o acesso a informação de todas as *tasks*. O congelamento apenas da *task* na qual ocorreu a parada de programa, acarretaria uma série de efeitos colaterais:

- não seria possível examinar valores de variáveis da demais *tasks* em execução;
- não seria possível examinar expressões dependentes de variáveis compartilhadas;
- poderiam ocorrer problemas associados a *time-outs* entre as *tasks* em execução e a *task* com execução congelada.

CSD foi concebido no paradigma orientado a objetos e implementado na linguagem C++. A forma de representação e o conjunto de heurísticas da metodologia de Especificação Unificada foram refinados através de sua aplicação a partes do sistema de software *CSD*.

4. Representação da Especificação Unificada

Na metodologia Especificação Unificada a representação dos requisitos é mapeada na especificação de classes e serviços que compõem o sistema. Este mapeamento mantém a informação necessária para a recuperação da especificação de requisitos a partir do nível de abstração representado na Especificação Unificada. As formas de representação utilizadas são apresentadas segundo sua seqüência no ciclo de desenvolvimento de sistemas de software:

- representação transitória da especificação de requisitos (a parte correspondente à especificação da interface é definitiva);
- representação do mapeamento de requisitos em classes e serviços;
- representação das classes e serviços que compõem a Especificação Unificada.

A sintaxe da linguagem de representação da metodologia Especificação Unificada está descrita no Apêndice A.

4.1 Representação dos Requisitos

A utilização da linguagem natural para especificar sistemas de software tem sido criticada devido a inconsistências, ambiguidades e falta de concisão. Na década de 80 uma técnica extremamente atraente do ponto de vista de automação previa a identificação de classes de objetos, operações e atributos a partir dos substantivos, verbos e adjetivos, respectivamente, da descrição em linguagem natural do sistema em desenvolvimento [Abbott83] [Booch86]. A Figura 10 ilustra a extração de objetos proposta a partir de representação textual.

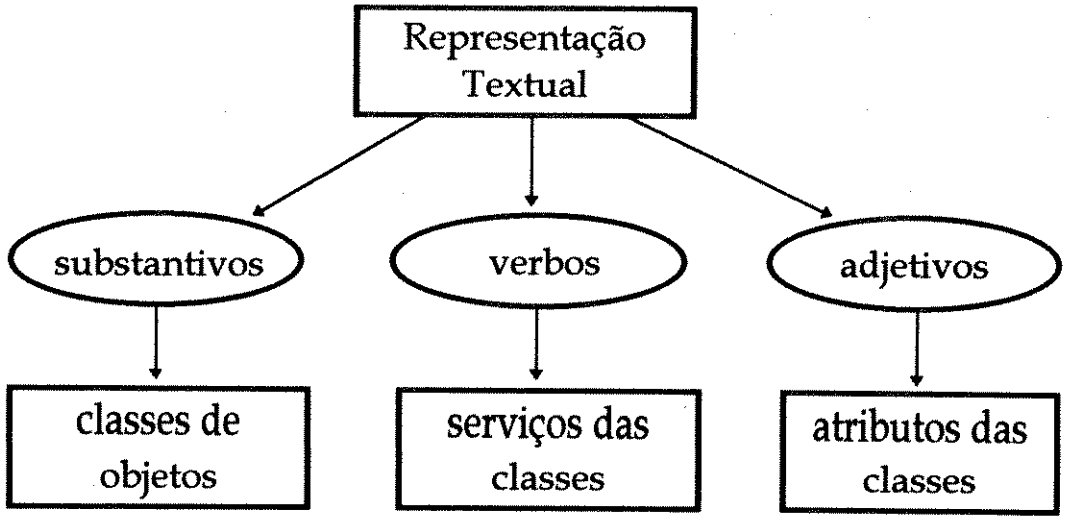

Figura 10: Extração de Objetos a partir de uma Representação Textual

Posteriormente [Booch91], esta técnica foi considerada inadequada para o desenvolvimento de sistemas não triviais. O mecanismo de extração mecânica de objetos, atributos e métodos a partir de descrições textuais de especificações é de fato questionável pois, em geral, as descrições textuais não obedecem a critérios (de concisão, completeza e não ambiguidade) que as qualifiquem como especificações adequadas de sistemas de software. A despeito disso, a maioria das metodologias de análise e projeto orientadas a objetos utilizam este mecanismo (refinado com heurísticas para retirar os candidatos inválidos).

Em geral, descrições textuais da especificação de um sistema deveriam ser refraseadas (focando apenas sujeito, verbo e predicado) e condensadas (eliminando muitas palavras e até frases irrelevantes ou redundantes). O vocabulário do domínio (conceitos do domínio) é obtido de forma mais sistemática e precisa a partir desse refraseamento. A Figura 11 ilustra a obtenção de conceitos a partir do refraseamento de uma representação textual.

Figura 11: Obtenção de Conceitos a partir do Refraseamento de uma Representação Textual

Para ilustrar essas transformações o texto original de uma especificação é refraseado em uma especificação mais concisa e menos ambígua. A aplicação de regras de refraseamento a este exemplo não faz parte da metodologia proposta; sua apresentação tem como objetivo apenas mostrar que o produto do refraseamento é uma forma mais adequada para se descrever uma especificação de um sistema de software do que uma descrição em linguagem natural.

O parágrafo abaixo corresponde à tradução da descrição hipotética de um sistema de software para interligar uma rede de bancos [Rumbaugh91].

"Projetar o software para suportar uma rede de bancos computadorizada incluindo terminais para caixas e caixas automáticos a serem partilhados por um consórcio de bancos. Cada banco provê seu próprio computador para manter suas contas e processar transações sobre elas. Cada banco possui seus próprios terminais para caixas que comunicam-se diretamente com o computador do banco. Caixas entram com dados da conta e da transação. Caixas automáticos comunicam-se com um computador central que encaminha as transações para os bancos apropriados. Um caixa automático aceita um cartão bancário, interage com o usuário, comunica-se com o sistema central para realizar a transação, fornece dinheiro e imprime recibos. O sistema requer registros apropriados e provisões de segurança. O sistema deve manipular acesso concorrente à mesma conta corretamente. Os bancos proverão o software para os seus próprios computadores; você deverá projetar o software para os caixas automáticos e para a rede. O custo do sistema

partilhado será rateado pelos bancos de acordo com o respectivo número de clientes com cartões bancários."

O primeiro passo do refraseamento corresponde à fragmentação do texto em cada uma de suas frases, eliminando-se apenas algumas palavras desnecessárias mas procurando-se manter a essência do original:

- projetar software para_suportar rede_bancos_computadorizada
- rede_bancos_computadorizada: inclui terminais_caixas, caixas_automáticos
- terminais_caixas, caixas_automáticos: são_partilhados_por consórcio_bancos
- banco: provê computador
- computador_banco: mantém contas; processa transações sobre conta
- banco: possui terminais_caixas
- terminais_caixas: comunicam_diretamente_com_o computador_banco
- caixas: entram_com dados_conta, dados_transação
- caixas_automáticos: comunicam_com computador_central
- computador_central encaminha transações para bancos
- caixa_automático: aceita cartão_bancário; interage_com usuário; comunica_com sistema_central para_realizar transação; fornece dinheiro; imprime recibos
- sistema: requer registros, provisões_segurança
- sistema: deve_manipular acesso_concorrente para mesma_conta
- bancos: proverão software para_seus computadores
- projetar software para caixas_automáticos, rede
- custo_sistema_partilhado: será_rateado_pelos bancos de_acordo_com número_clientes_com cartões_bancários

A primeira reestruturação destas frases leva ao agrupamento em torno das entidades associados ao problema (elementos que definem o vocabulário do problema em um dado domínio), utilizando as seguintes heurísticas [Rumbaugh91] :

- adição de frases implícitas:
  - consórcio_bancos consiste_de bancos;
  - consórcio_bancos possui computador_central (caixas_automáticos: são_partilhadas_por consórcio_bancos; comunicam_com computador_central);
  - clientes têm cartões_bancários (de_acordo_com número_clientes_com cartões_bancários);
- do domínio do problema é possível inferir:
  - cartão_bancário acessa contas;
  - banco emprega caixas;
- fatoração de frases com mesmo sujeito:
  - caixas automáticos: são_partilhados_por consórcio_bancos, comunicam_com computador_central.

A segunda reestruturação leva à eliminação ou transformação de entidades e associações utilizando as seguintes heurísticas [Rumbaugh91] :

- remoção de entidades irrelevantes ao problema : custo_sistema_partilhado;
- eliminação de entidades redundantes: rede_bancos (já representada pelas entidades: caixas_automáticos, terminais_caixas, computador_central, computador_banco);
- eliminar associações com entidades eliminadas;
- decompor associações ternárias (computador_banco: mantém contas; processa transações sobre conta) em binárias (computador_banco: mantém contas; transações: são_processadas_por computador_banco) quando possível (nem sempre isto é possível sem que ocorra perda de informação);
- distinguir subtipos de entidades em função de sua utilização (transação é especializada em transação_remota e transação_interna).

Outras transformações foram aplicadas no exemplo original [Rumbaugh91], irrelevantes para o propósito deste texto. Com a descrição deste exemplo procura-se mostrar uma especificação em linguagem natural representada (de forma mais concisa e não ambígua) através de conceitos (que correspondem às entidades do domínio do problema), serviços e requisitos adicionais.

Os conceitos resultantes desta transformação são os seguintes:

- consórcio_bancos:
  - consiste_de mais_de_um banco;
  - possui computador_central;
- banco:
  - possui computador_banco;
  - possui mais_de_um terminal_caixa;
  - mantém mais_de_uma conta;
  - emprega mais_de_um caixa;
- computador_central:
  - comunica_com mais_de_uma caixa_automático;
  - comunica_com mais_de_um computador_banco;
- computador_banco:
  - comunica_com mais_de_um terminal_caixa;
- cliente:
  - tem mais_de_um cartão_bancário;
  - tem mais_de_uma conta;
- cartão_bancário:
  - acessa mais_de_uma conta;
- conta:
  - é_acessada_por mais_de_uma transação_remota;

- transação_remota:
  - é_realizada_em caixa_automático;
  - é_autorizada_por computador_central;
- transação_interna:
  - é_realizada_em terminal_caixa;
  - é_solicitada_por caixa.

O serviço retirada de dinheiro, obtido a partir do texto (apenas com o objetivo de mostrar que serviços são referenciados em descrições em linguagem natural; caso contrário, os serviços teriam que ser mais detalhados), corresponde à seguinte seqüência de subserviços:

- caixa_automático aceita cartão_bancário;
- caixa_automático interage_com usuário;
- caixa_automático solicita_realização_de transação_remota;
- caixa_automático libera dinheiro;
- caixa_automático imprime recibo.

Os requisitos adicionais obtidos a partir do texto são os seguintes:

- sistema_automação_bancária requer registros;
- sistema_automação_bancária requer provisões_segurança.

A partir deste exemplo procura-se mostrar que uma descrição textual não é adequada como representação de uma especificação de um sistema de software e que a utilização direta (sem partir da descrição em linguagem natural) de conceitos, serviços e requisitos adicionais é uma alternativa para obter uma especificação concisa sem, no entanto, perder a legibilidade de uma descrição em linguagem natural.

A forma de representação obtida, baseada em restrições sobre a linguagem natural, poderia ter sido utilizada para especificar diretamente os requisitos do problema. Requisitos podem ser expressos de forma concisa, utilizando nomes para captar apenas a informação essencial, em substituição às habituais descrições em linguagem textual. Mas o objetivo não é utilizar uma forma diagramática (modelos de entidade e relacionamento, diagramas de interações, e de estado) mas, sim, uma forma reduzida de uma descrição textual. Neste sentido, toda informação que poderia estar em uma hipotética descrição em linguagem natural e que necessita permanecer para descrever os requisitos, é capturada de forma condensada a partir dos nomes.

Em função desta opção por uma descrição de requisitos baseada em um subconjunto da linguagem natural, a forma de representação proposta para descrever os requisitos do sistema é apresentada nas subseções posteriores.

O ponto de partida da metodologia Especificação Unificada é a representação dos requisitos essenciais do sistema a ser construído: um sistema de software é definido a partir dos serviços que oferece. A definição dos serviços utiliza conceitos que compõem o vocabulário associado ao domínio do problema. Para completar a definição dos serviços é necessário caracterizar requisitos

adicionais que estabelecem restrições ou propriedades que complementam a descrição dos serviços. Com estes três elementos básicos (serviços, conceitos e requisitos adicionais) propõe-se a representação de requisitos para sistemas de software.

Do ponto de vista de manutenção corretiva ou evolutiva, mudanças na interface do sistema de software devem ser absorvidas pelas classes de objetos responsáveis pela troca de informação com o meio externo sem se propagar para o resto do sistema. Existem sistemas de software cuja interface com usuários é fortemente interativa (para os quais a decomposição por eventos tem forte poder heurístico) enquantos outros sistemas têm interface não interativa (para os quais nem há eventos a caracterizar). Na metodologia Especificação Unificada a concepção do *layout* da interface, que atenda aos usuários e suporte a recepção de eventos e a troca de informação associada aos serviços oferecidos pelo sistema, faz parte da especificação dos requisitos (dispensável ou mais simples em função do tipo de interface requerido pelo sistema).

A Figura 12 ilustra os elementos propostos pela Especificação Unificada para especificar requisitos de sistemas de software.

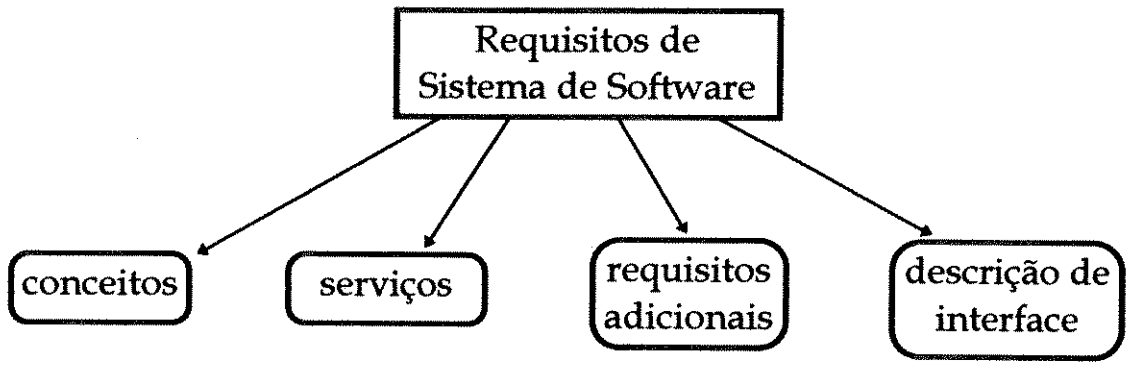

Figura 12: Elementos Propostos para a Especificação de Requisitos

4.1.1 Representação dos Requisitos de Serviços

Observa-se no exemplo de refraseamento da especificação que a descrição em linguagem natural do sistema de software é substituída por uma especificação que se utiliza de três construções básicas:

- conceitos (ou entidades do domínio) : descritos em função de suas propriedades e relacionamentos com outros conceitos (como vocabulário que nos permite expressar outros conceitos, serviços e requisitos adicionais);
- serviços: descritos em função de seqüências, seleções e iterações de procedimentos de utilização do sistema (no exemplo somente seqüências de procedimentos são utilizadas);
- requisitos adicionais: definem restrições ou propriedades que complementam a descrição dos serviços.

Do ponto de vista do usuário, um sistema de software deve ser especificado a partir do conjunto de todas as formas de utilizá-lo. Esta abordagem, proposta originalmente pela metodologia *OOSE* [Jacobson92], tem tido grande aceitação pelos proponentes de metodologias de análise e projeto orientadas a objetos [Rumbaugh94c]. Sob este ponto de vista, os requisitos de um sistema de software devem especificar, essencialmente, os serviços que o sistema deve prestar.

A descrição de um serviço utiliza conceitos que fazem parte dos domínios do problema associado ao sistema de software (a base do entendimento entre as pessoas é a existência de um vocabulário comum que conceitua os termos utilizados). Complementando os requisitos definidos pelos serviços, requisitos adicionais são utilizados para definir restrições ou propriedades que não seriam adequadamente representadas na descrição dos serviços. Portanto, a especificação de requisitos de um sistema de software pode ser representada, genericamente, através de conceitos, serviços e requisitos adicionais.

Um conceito é descrito como um conjunto de relacionamentos com outros conceitos. O ponto de parada desta definição recorrente é a descrição de um conceito baseada somente em relacionamentos com conceitos definidos como primitivos em um dado domínio. Especialização é definida como um relacionamento especial devido à sua relevância no paradigma orientado a objetos.

Um dicionário de uma determinada língua define conceitos, expressando sua relação com outros conceitos a partir de construções de linguagem natural e com várias alternativas de significados (o contexto no qual o conceito é utilizado determina o seu significado). No contexto de especificação de um sistema de software um conceito deve ser precisa e concisamente definido.

Na metodologia prosposta, a referência a conceitos, relacionamentos, serviços e requisitos adicionais é feita através de nomes. O nome é definido como uma composição de palavras (formadas por caracteres alfabéticos) ligadas por separadores (*underscores*). A descrição da especificação de requisitos utiliza apenas nomes em cuja composição só participam palavras essenciais à descrição, tornando a notação mais concisa sem, no entanto, implicar em perda da semântica do elemento descrito.

A definição de conceitos pode ser sistematizada a partir dos seguintes elementos:

- nome do conceito (em geral, corresponde ao sujeito das frases nas descrições textuais);
- conjunto de relacionamentos.

O conceito banco, do exemplo do sistema para interligar uma rede de bancos apresentado na seção anterior, é definido da seguinte forma:

- nome do conceito: banco;
- relacionamentos do conceito:
  - possui computador_banco;
  - possui mais_de_um terminal_caixa;
  - mantém mais_de_uma conta;
  - emprega mais_de_um caixa.

Um relacionamento é definido a partir de:

- nome do relacionamento: em geral, corresponde à frase verbal ligando sujeito ao predicado nas descrições textuais;
- cardinalidade (somente unidirecional) do relacionamento (descrita pelo nome "mais_de_um" para cardinalidade múltipla e omitida para não múltipla);
- nome do conceito associado (em geral, corresponde ao predicado nas descrições textuais).

Os relacionamentos associados ao conceito banco ilustram a definição de relacionamento, como a seguir:

- nome do relacionamento: emprega;
- cardinalidade do relacionamento: mais_de_um;
- nome do conceito associado ao relacionamento: caixa.

O segundo elemento encontrado nas especificações textuais é o serviço a ser prestado pelo sistema de software. A representação de serviços do sistema corresponde a uma especificação de requisitos (comumente encontrada nas especificações em linguagem natural), ao caracterizar o aspecto dinâmico e funcional do conjunto de serviços que o sistema deve prover. A descrição de um serviço pode ser sistematizada a partir da seguinte forma:

- nome do serviço;
- seqüências, seleções e iterações de nomes de: eventos, respostas e serviços (referenciando os conceitos definidos anteriormente).

O serviço retirada de dinheiro, do exemplo do sistema para interligar uma rede de bancos, é definido através da seguinte seqüência de subserviços:

- caixa_automático aceita cartão_bancário;
- caixa_automático interage_com usuário;
- caixa_automático solicita_realização_de transação_remota;
- caixa_automático libera dinheiro;
- caixa_automático imprime recibo.

A utilização de seleções e de iterações na representação de um serviço a partir de subserviços é ilustrada oportunamente na Seção 5.3.

O terceiro elemento encontrado nas especificações textuais é o requisito adicional, descrito por um nome que define restrições e propriedades associadas aos conceitos e serviços. O requisito adicional pode ser utilizado para complementar a descrição de relacionamentos dos conceitos (como a descrição de um relacionamento ternário que não seja conversível em relacionamentos binários sem perda de informação).

Requisitos adicionais podem ser ilustrados a partir daqueles obtidos no exemplo do sistema para interligar uma rede de bancos:

- sistema_automação_bancária requer registros;
- sistema_automação_bancária requer provisões_segurança.

É fundamental observar que qualquer especificação representada em texto livre pode ser refraseada (reordenada e simplificada sem perda da sua informação de especificação) e decomposta em um conjunto de frases concisas, cujo poder de representação é equivalente ao do texto livre, ou seja, consegue expressar qualquer especificação. Dessa forma, a sintaxe associada a requisitos adicionais permite expressar qualquer especificação através de, simplesmente, um conjunto de requisitos.

Mesmo especificações em domínios específicos, como o matemático por exemplo, podem ser expressas através de um conjunto de requisitos, no qual os símbolos matemáticos utilizados são substituídos pelo conceito equivalente em linguagem textual (por exemplo integral). No entanto, essa representação, apesar de ser abrangente, torna-se inadequada para expressar concisamente os conceitos de um dado domínio.

A metodologia Especificação Unificada propõe a representação de especificações de sistemas de software através de conceitos, serviços, requisitos adicionais e uma notação específica para a descrição de interfaces gráficas (descrita na Seção 5.1.2). Esta forma de representação tem equivalência com a forma proposta pela metodologia *OOSE* [Jacobson92]:

- vocabulário do domínio (em *OOSE*): é representado por conceitos (na Especificação Unificada);
- casos de uso: são representados por serviços de sistema;
- descrição de interface: é representada pela descrição de *layouts* de janelas e eventos (ver seção 5.1.2) para o caso específico de interfaces gráficas (o caso geral pode ser representado por conceitos, serviços e requisitos adicionais).

Adicionalmente, requisitos adicionais permitem expressar propriedades e restrições sobre serviços, de forma a simplificar a descrição dos serviços. Portanto, a forma de representação proposta pela metodologia Especificação Unificada é abrangente, do ponto de vista de representação de especificações de sistemas de software, e pode ser considerada como uma formalização para a representação proposta pela metodologia *OOSE*. Esta formalização será extendida para a representação de classes e serviços nas seções seguintes deste capítulo.

Na metodologia Especificação Unificada os conceitos, serviços do sistema e requisitos adicionais são mapeados em classes de objetos, serviços associados às classes e justificativas de decomposições de serviços, resultando em uma única forma de representação. O mecanismo de visões (ilustrado no Capítulo 7) permite a recuperação dos conceitos, serviços e requisitos adicionais que deram origem às classes de objetos e serviços que constituem a especificação unificada.

O modelo de relacionamento entre as entidades que compõem a metodologia Especificação Unificada, descrito a partir de um subconjunto de linguagem natural, é mostrado abaixo:

- conceito:
  - definido_por um_ou_mais conceito;
- serviço:
  - utiliza um_ou_mais conceito;

- utiliza um_ou_mais evento;
- requisito_adicional:
  - utiliza um_ou_mais conceito;
  - justifica um_ou_mais serviço;
- sistema:
  - composto_de um_ou_mais serviço_sistema;
- serviço_sistema:
  - decomposto_em mais_de_um serviço;
- classe_objetos:
  - contém um_ou_mais serviço;
  - mapeada_por conceito;
  - mapeada_por região_interface;
- evento:
  - associado_a um_ou_mais região_interface;
- interface:
  - composta_de um_ou_mais região_interface;
  - composta_por um_ou_mais evento.

É importante observar que o conceito serviço de sistema é utilizado neste trabalho com a mesma semântica de casos de uso, propostos pela metodologia *OOSE* [Jacobson92]. Esta conceituação se deve à comparação de um sistema de software a uma classe abstrata denominada sistema, cujos serviços (ou seja, serviços de sistema) correspondem ao conjunto de serviços que o sistema deve prestar do ponto de vista dos usuários. A Figura 13 ilustra o conjunto de serviços prestados pela classe abstrata sistema.

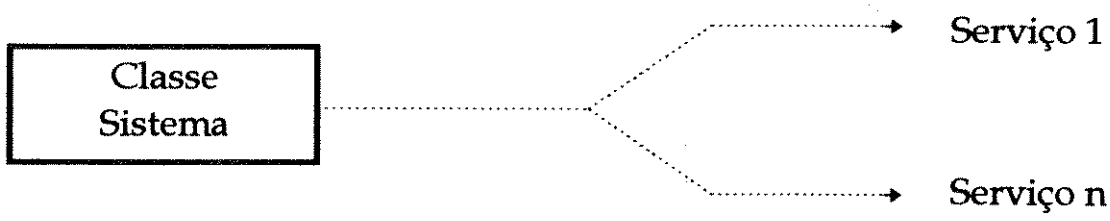

Figura 13: O Conjunto de Serviços prestados pela Classe Abstrata Sistema

O papel das metodologias de análise e projeto orientadas a objetos é o suporte à decomposição da classe sistema nas classes de objetos (que provêem os serviços que expressam esta decomposição) que implementam o sistema como uma grande cooperação entre objetos. A efetividade deste suporte se caracteriza pela capacidade representacional da linguagem de representação associada à metodologia (gráfica, textual formal ou informal, ou mista) de expressar essa de-

composição, e pela capacidade do corpo heurístico provido pela metodologia em guiar esta decomposição. A Figura 14 ilustra a decomposição de um serviço em sub-serviços.

Figura 14: A Decomposição de um Serviço em Sub-Serviços

A Figura 15 apresenta um MER que ilustra o modelo de relacionamento entre as entidades que compõem a metodologia Especificação Unificada.

4.1.2 Representação dos Requisitos de Interface

Um sistema de software pode ser descrito a partir dos serviços demandados por usuários humanos ou sistemas externos. Na interface com sistemas externos, as classes de objetos de interface devem encapsular o protocolo de comunicação com tais sistemas e prover ao resto do sistema uma interface de alto nível. A especificação dos serviços solicitados a sistemas externos envolve, para cada serviço:

- nome do serviço;
- lista de informações necessárias (argumentos);
- lista de informações obtidas (resultado).

A representação da interface com usuários em sistemas muito interativos pode ser complexa merecendo atenção especial na metodologia Especificação Unificada. Especificações de interfaces mais simples, que não utilizam janelas, não requerem as descrições adicionais introduzidas nesta seção; no entanto, a utilização de interfaces gráficas que utilizam janelas é dominante no cenário atual de desenvolvimento de software. Esta tendência se deve ao fato de que a interface deve tornar amigável a comunicação de seres humanos com sistemas de software, premissa que pode ser atendida através de recursos como *layout* de interfaces compostas de elementos gráficos (janelas e ícones) ou através de requisitos tais como *display* de informações (para caracterizar um dado contexto ou mensagens de erro) e necessidade de consistência da informação fornecida (*string* digitado) e, algumas vezes, até da informação selecionada (opção escolhida de um menu).

De maneira geral, a execução de um serviço demandado por um usuário pode envolver: recepção de eventos (solicitando serviço ou fornecendo informação necessária à sua execução), envio de respostas (incluindo mensagens de erro) para troca de informação com o usuário, transformação de informação, armazenamento de informação (para utilização durante o serviço entre eventos ou entre serviços) e delegação de serviços a sistemas externos.

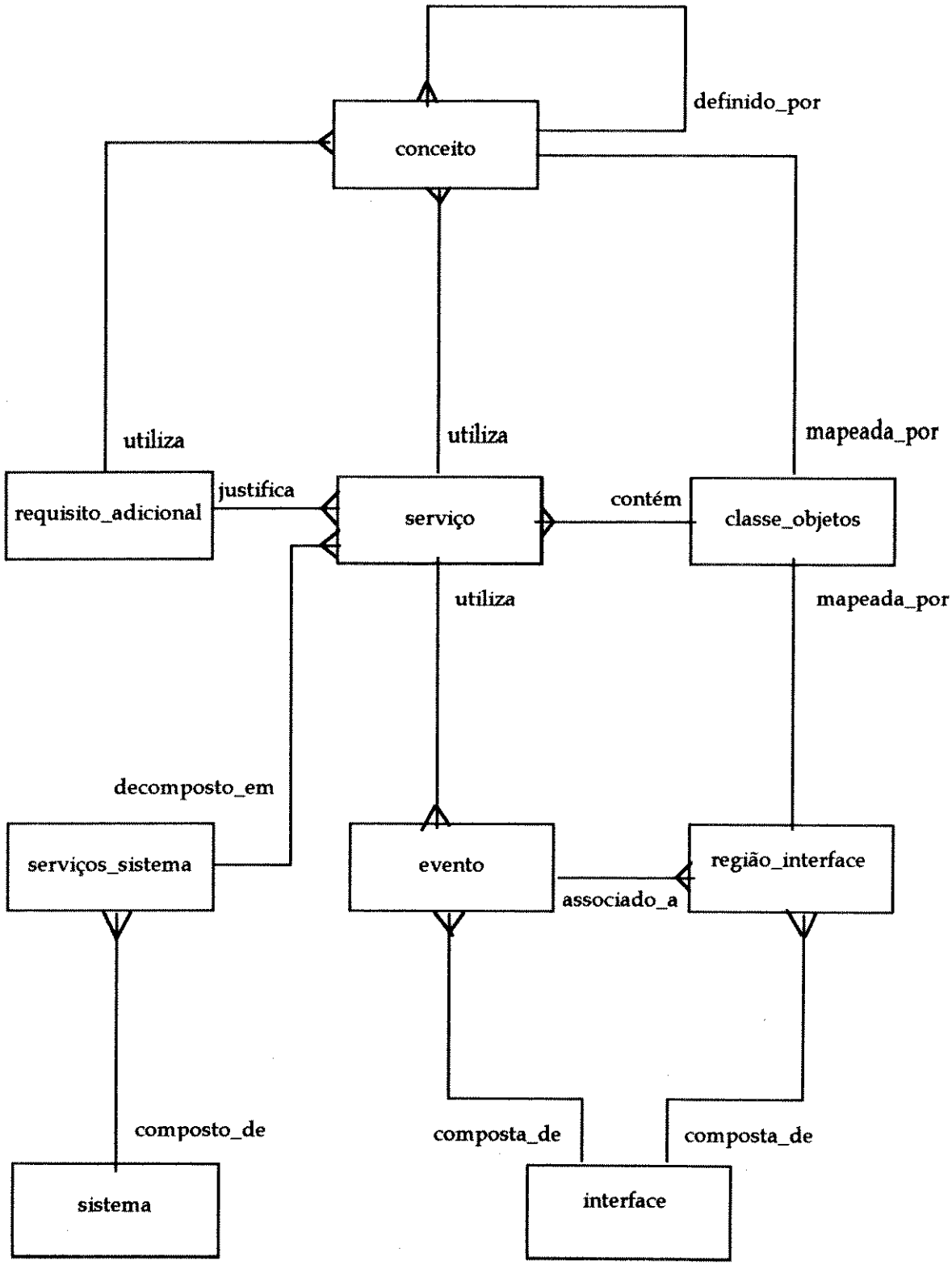

Figura 15: MER da Especificação Unificada

A especificação da interface com usuário deve servir de base para simulações de interface com usuários, a partir de prototipagem rápida utilizando *GUIMS* (*Graphical User Interface Manager System*) e para a obtenção dos objetos que acoplam a interface dos usuários com o resto do sistema.

A especificação dos requisitos de interface é representada através de:

- descrição do *layout* de cada uma das janelas independentes que compõem a interface, discriminando cada um dos elementos que poderão ser acessados na troca de informações com o usuário;
- descrição dos eventos, no nível adequado de abstração dos objetos de interface, como compostos de eventos mais primitivos (como evento primitivo não se entenda a recepção de cada caracter de uma informação fornecida mas, por exemplo, a seleção de uma opção de menu ou o fornecimento de uma informação).

A forma de representação da interface faz parte da Especificação Unificada (diferentemente de conceitos, serviços e requisitos adicionais, cuja forma de representação inicial na especificação de requisitos deve ser recuperada através do mecanismo de visões) . A sintaxe utilizada para representá-la é apresentada nas subseções que se seguem.

4.1.2.1 Descrição das Janelas da Interface

Na metodologia Especificação Unificada a concepção das classes de objetos de interface (responsáveis pela troca de informações do sistema de software com o seu meio ambiente) é proposta a partir do mapeamento das regiões que compõem as janelas (ou das próprias janelas) que constituem a interface (ver Seção 6.2). Eventos podem ser associados a uma dada região da interface a partir de diferentes visões de acesso; por exemplo, um programa fonte em depuração (em *display* em uma determinada janela ou região de janela) pode ser visto como um texto único, como um conjunto de linhas ou como um conjunto de símbolos de programa. Em função dessas necessidades, a descrição da interface deve permitir a caracterização de cada uma das regiões e de cada uma das visões de acesso dessas regiões.

Para exemplificar a representação de janelas de interface são mostrados a seguir *layouts* correspondentes às duas janelas que compõem a interface (que corresponde à parte reativa do sistema de software, para a qual são definidas uma forma de representação e heurísticas de decomposição específicas): de diretório e de programa. A janela de diretório (*DirectoryWindow*) é utilizada para navegar nos diretórios da máquina alvo (na qual o programa a ser depurado é executado) e para escolher um arquivo executável para depuração. Sua composição inclui:

- *directory_path*: linha com *path* do diretório do arquivo escolhido;
- *MessageLine*: linha de mensagens (onde são mostradas mensagens de erro);
- *Quit*: botão de *quit*;

- *DirectoryRegion*: região de diretório (cujas entradas correspondem a arquivos de um dado diretório na máquina alvo, permitindo selecionar um outro diretório ou um arquivo executável para depuração).

A janela de programa (*ProgramWindow*) é utilizada para instalar *breakpoints*, colocar variáveis em *display* e executar o programa, examinando seu estado (pilha e variáveis) passo a passo ou em pontos de parada. Sua composição inclui:

- *program_name*: linha com nome do programa em depuração;
- *MessageLine*: linha de mensagens;
- *Quit*: botão de *quit*;
- *TypingLine*: linha de edição para solicitação de comandos tais como *display* de fonte de procedimento, processo na região de fonte ou *display* de variáveis na região de estado (através de *path* para identificar a localização do símbolo desejado);
- *StateRegion*: região de estado, utilizada para apresentação do estado do programa (e ações adicionais tais como: selecão de *breakpoints* para remoção, seleção de variáveis em *display* para remoção ou formas alternativas de *display*, seleção de procedimento da pilha para foco na região fonte, etc); é composta por quatro áreas, cada uma delas contendo um *header* com seu nome e entradas: *breakpoints*, tarefas ativadas (uma vez que linguagem é concorrente), procedimentos na pilha do programa (aos quais pode se associar variáveis em *display*) e fontes que compõem o programa executável (idem para suas variáveis globais);
- *ProgramMenu*: menu de comandos de execução do programa;
- *SourceRegion*: região fonte composta de linha indicando fonte em *display* e linha sob foco do *mouse*, menu de comandos para posicionamento do fonte, e região de *display* do texto de um dos fontes do programa (a partir do qual é possível inserir ou remover *breakpoints*, executar até uma linha selecionada e selecionar váriaveis para *display* na região de estado).

```
LAYOUT DirectoryWindow
LAYER directory_path
LAYER
    PARTITION MessageLine
    PARTITION Quit
LAYER DirectoryRegion : SET (FileEntry)

LAYOUT ProgramWindow
LAYER program_name
LAYER
    PARTITION MessageLine
    PARTITION Quit
LAYER TypingLine
LAYER StateRegion
    LAYER BreakArea : BreakHeader, SET (BreakEntry)
    LAYER TaskArea : TaskHeader, SET (TaskEntry)
    LAYER StackArea : StackHeader, SET (FrameEntry)
    LAYER SourceArea : SourceHeader, SET (SourceEntry)
```

LAYER ProgramMenu
PARTITION Run
PARTITION Cont
PARTITION Next
PARTITION Step
PARTITION Stop
PARTITION Abort
PARTITION Detach
LAYER SourceRegion
LAYER
PARTITION source_name, line_n
PARTITION
PARTITION Back
PARTITION Up
PARTITION Down
PARTITION Search
LAYER TextSource : SET (LineSource) : SET (SymbolSource)

O *layout* de uma janela pode ser representado a partir de camadas (subdivisões horizontais, como *LAYER StateRegion*) e partições (subdivisões verticais como *PARTITION MessageLine*). Tanto uma camada como uma partição podem conter outras camadas e partições (a camada StateRegion, por exemplo, contém as camadas *BreakArea, TaskArea, StackArea e SourceArea*). O *layout* não especifica as dimensões da janela e de seus componentes (uma vez que esta informação não é necessária nem para a simulação nem para a concepção dos objetos de interface). Elementos acessíveis (*headers*, entradas e linhas) das camadas ou partições primitivas deverão ser associados a um nome que qualifique cada uma das visões de acesso: o texto em *display* na região fonte (*LAYER TextSource*), por exemplo, pode ser identificado como um todo, como um conjunto de linhas (*SET (LineSource)*), ou como um conjunto de símbolos ou palavras (*SET (SymbolSource)*).

4.1.2.2 Descrição dos Eventos da Interface

A descrição dos eventos recebidos pela interface é fundamental para a descrição dos serviços providos pelo sistema a partir da perspectiva do usuário. Basicamente os serviços de sistema são descritos a partir da seqüência de eventos que caracteriza uma determinada forma de utilização do sistema (caso de uso). A noção de evento composto é muito útil para encapsular uma seqüência de eventos, quando o núcleo do sistema (sistema excluindo sua interface) não precisa tomar conhecimento dos eventos internos à seqüência. Serviços do sistema podem estar associados a um evento, a uma alternativa de eventos ou a estados (aos quais podem ser associadas alternativas de eventos). Esta seção apresenta exemplos para ilustrar:

- a descrição de eventos associados a janelas e regiões da interface, associando a geração de eventos à utilização de *mouse* e ao aparecimento de menus auxiliares;
- a definição de eventos compostos;

- a associação e eventos e estados à descrição dos serviços de sistema.

Os eventos gerados através da utilização de *mouse* são listados para cada uma das configurações. Se a seleção de um elemento causa o aparecimento de um menu ou *template* auxiliar, este novo elemento deverá ter sua composição descrita. No caso de *template*, seu *layout* deverá ser descrito.

A seguir são mostrados os eventos associados aos *mouses* das duas janelas. Na janela de diretórios, o botão esquerdo do *mouse* é utilizado para:

- selecionar um arquivo (diretório para *display* ou executável para depuração): *LEFT MouseSelect FileEntry.*

Na janela de programa os três botões do *mouse* podem ser utilizados:

- o botão esquerdo permite selecionar:
  - tarefas: *LEFT MouseSelect TaskEntry;*
  - procedimentos da pilha: *LEFT MouseSelect FrameEntry;*
  - símbolos do texto em *display* na região fonte: *LEFT MouseSelect SymbolSource;*
  - comandos de execução do programa: *LEFT MouseSelect Run, Cont, Next, Step, Stop, Abort, Detach;*
  - comandos de posicionamento do fonte: *LEFT MouseSelect Back, Up, Down;*
- o botão central indica início de edição:
  - na linha de edição (*path* de variável para *display*): *MIDLLE MouseEdit TypingLine;* ou
  - na área de texto da região fonte (para inclusão de código interpretado, como *patch* ao programa em depuração: *MIDLLE MouseEdit TextSource;*
- o botão direito permite selecionar opções de menus associados às regiões da janela de programa:
  - *quit: RIGHT MouseMenu menu_quit (Quit) = [Cancel, Confirm];*
  - *headers* das áreas de *display*:
    - dos *breakpoints*: *RIGHT MouseMenu menu_break_header(BreakHeader) = [Expand, Collapse, RemoveAll] ;*
    - das tarefas ativas: *RIGHT MouseMenu menu_task_header (TaskHeader) = [Expand, Collapse, SelectCurrent];*
    - dos procedimentos da pilha: *RIGHT MouseMenu menu_stack (StackHeader) = [Expand, Collapse];*
    - dos fontes do programa: *RIGHT MouseMenu menu_source_header (SourceHeader) = [Expand, Collapse] ;*
  - entradas das áreas de *display*:
    - dos *breakpoints*: *RIGHT MouseMenu menu_break(BreakEntry) = [Remove, Source] ;*
    - das tarefas ativas: *RIGHT MouseMenu menu_task (TaskEntry) = [Source];*

- dos procedimentos da pilha: *RIGHT MouseMenu menu_frame (FrameEntry) = [Source, Collapse, Evaluate, Arguments, Locals];*
- dos fontes do programa: *RIGHT MouseMenu menu_frame (FrameEntry) = [Source, Collapse, Evaluate, Arguments, Locals];*

- opções do comando de pesquisa de *strings*: *menu_search (Search) = [Backwards, Forwards];*
- ações associadas às linha do fonte em *display* (bem como das linhas resultantes da inserção de *breakpoints*): *menu_source_line (LineSource) = [SetBreak, ExecuteToHere, Evaluate], menu_extra_line (LineSource) = [RemoveBreak].*

MOUSE DirectoryWindow
LEFT MouseSelect
- *FileEntry*

MOUSE ProgramWindow
LEFT MouseSelect
- *TaskEntry*
- *FrameEntry*
- *SymbolSource*
- *Run*
- *Cont*
- *Next*
- *Step*
- *Stop*
- *Abort*
- *Detach*
- *Back*
- *Up*
- *Down*

MIDLLE MouseEdit
- *TypingLine*
- *TextSource*

RIGHT MouseMenu
- *menu_quit (Quit) = [Cancel, Confirm]*
- *menu_break_header(BreakHeader) = [Expand, Collapse, RemoveAll]*
- *menu_break(BreakEntry) = [Remove, Source]*
- *menu_task_header (TaskHeader) = [Expand, Collapse, SelectCurrent]*
- *menu_task (TaskEntry) = [Source]*
- *menu_stack (StackHeader) = [Expand, Collapse]*
- *menu_frame (FrameEntry) = [Source, Collapse, Evaluate, Arguments, Locals]*
- *menu_source_header (SourceHeader) = [Expand, Collapse]*
- *menu_source (SourceEntry) = [Source, Collapse, Evaluate, Expand, ExpandGrants]*
- *menu_search (Search) = [Backwards, Forwards]*
- *menu_source_line (LineSource) = [SetBreak, ExecuteToHere, Evaluate]*
- *menu_extra_line (LineSource) = [RemoveBreak]*

Os eventos compostos (eventos de mais alto nível que ativam objetos de interface) são descritos a partir dos eventos mais primitivos. Para exemplificar são mostrados dois eventos compostos associados à classe de objetos *BreakArea*:

- solicitação de expansão dos *breakpoints* na região de estado: usuário seleciona menu associado ao *header* da área de *breakpoints* (*menu_break_header*) e, em seguida, seleciona a opção de expansão (*Expand*) do referido menu;
- remoção de *breakpoint*: usuário seleciona menu associado ao *breakpoint* escolhido (*menu_break*) para remoção (entrada na área de *breakpoints*) e, em seguida, seleciona a opção remoção (*Remove*) do referido menu.

CLASS BreakArea

EVENT Expand
EVENT = menu_break_header User.MouseMenu (BreakHeader)
EVENT = User.Expand (menu_break_header)

EVENT Remove
EVENT = menu_break User.MouseMenu (BreakEntry)
EVENT = entry_n User.Remove (menu_break)

A um determinado serviço podem ser associados uma alternativa de eventos (cujo caso particular e mais comum é um único evento) ou uma seqüência de alternativas de eventos. Alguns serviços do sistema, descritos em função de seus eventos, são relacionados abaixo. Pode-se observar que as descrições esquemáticas dos serviços *NextLineOverCall* e *NextLineIntoCall*, por exemplo, requerem a utilização de estados e de alternativas de eventos (para um dado estado).

Alguns serviços são caracterizados por um único evento (composto de seqüências de eventos mais primitivos), como os serviços:

- mudança de diretório na *DirectoryWindow* (*ChangeDirectory*) através da seleção de uma entrada de diretório;
- inserção de *breakpoint* na *SourceRegion* (*SetBreak*) através da seleção da opção inserir *breakpoint* associada a uma linha do fonte.

SERVICE ChangeDirectory
EVENT DirectoryRegion.ChangeDirectory

SERVICE SetBreak
EVENT SourceRegion.SetBreak

Existem serviços que são caracterizados por uma alternativa de eventos, que indica que qualquer um dos eventos relacionados pode ativá-lo, como os serviços:

- remoção de *breakpoint* (*RemoveBreak*): na *SourceRegion* (através da seleção de uma linha com *breakpoint*) ou na *BreakArea* (através da seleção de uma entrada de *breakpoint*);
- *display* de fonte de *task* (*DisplaySourceTask*): na *SourceRegion* (através da seleção de um símbolo que corresponde a uma definição de processo), na *TaskArea* (através da seleção da opção de

display de fonte associada a uma entrada correspondente a uma *task*), ou na *TypingLine* (através da edição do caminho correspondente à definição de um processo, denominado *symbol path*).

SERVICE RemoveBreak
EVENT SourceRegion.RemoveBreak
EVENT BreakArea.Remove

SERVICE DisplaySourceTask
EVENT SourceRegion.SelectedSymbol
EVENT TaskArea.Source
EVENT TemplateMenu.SymbolPath

Existem serviços cuja seqüência de eventos exige a representação de estados, como o serviço *ExecuteToHere*: execute até uma uma determinada linha (selecionada no fonte com a opção de executar até aquele ponto), que ignora os *breakpoints* de usuários que forem encontrados (permanecendo no estado *dragging*) até alcançar a linha desejada (na qual é colocado um *breakpoint* artificial temporário).

SERVICE ExecuteToHere
STATE stopped
EVENT SourceRegion.ExecuteToHere
STATE dragging (running)
EVENT ProgramWindow.BreakStop

Existem serviços que contêm estados com alternativas de eventos associados a estados, como o serviço *NextLineIntoCall*: execute até a próxima linha sem excluir uma chamada interna como ponto de parada. Este comando é usualmente denominado como *Step* nos depuradores. O serviço associado ao comando *Step* (*NextLineIntoCall*) é utilizado para ilustrar a recuperação de visões na Especificação Unificada; na seção 7.3.1 é descrito o seu diagrama de transição de estados. A descrição seguinte foi simplificada com o objetivo apenas de ilustrar a ocorrência de mais de um evento em um dado estado.

Ao receber a solicitação da execução do comando (estado *stopped*), através da região de interface *ProgramMenu*, o depurador comanda a execução na máquina alvo do trecho de código correspondente à linha fonte corrente (indo para o estado *stepping*). Dois eventos distintos podem chegar no estado *stepping*: *breakpoint* de usuário (*BreakStop*) e execução fora do trecho de código especificado (*OutStepRange*). Se a execução corresponde a um procedimento interno, o depurador comanda a inserção de um *breakpoint* artificial na máquina alvo com o objetivo de pular o trecho de instruções associado ao *header* do procedimento e parar na primeira instrução executável (indo para o estado *stepping_inner_header*). Caso contrário, se ocorrer um *breakpoint* do usuário ou a exe-

cução está em um procedimento mais externo ou no mesmo procedimento, o serviço é considerado concluído.

SERVICE NextLineIntoCall
STATE stopped
EVENT ProgramMenu.Step
STATE stepping (running)
EVENT ProgramWindow.BreakStop
EVENT ProgramWindow.OutStepRange
STATE stepping_inner (running)
EVENT ProgramWindow.BreakStop
STATE stepping_inner_header (running)
EVENT ProgramWindow.BreakStop

A seqüência de eventos leva à noção de estado. Assim sendo, na sua forma mais geral um serviço pode ser composto por um conjunto de estados. A cada estado é associado um conjunto alternativo de eventos. A cada par estado/evento será associada uma ordenação de serviços, finalizada pela definição do próximo estado (que pode ser o estado atual). A noção de estado é utilizada para caracterizar a recepção de eventos na interface (informação assíncrona sobre cuja chegada o sistema não tem controle).

4.2 Representação de Classes

A Especificação Unificada mantém apenas um nível de abstração a partir do qual são recuperados os níveis mais altos de abstração. A descrição dos conceitos definidos nas especificações de requisitos é recuperada a partir da definição das classes de objetos e da seguinte informação adicional:

- *layouts* de interface;
- lista dos conceitos;
- mapeamentos.

Os *layouts* de interface definem as regiões de interfaces que são mapeadas diretamente em classes de interface. A lista de conceitos contém os nomes de todos os conceitos determinados na especificação de requisitos. Para ilustrar são listados alguns dos conceitos.

CONCEPT Program, Address, Source, Line, Block, Symbol, Break, Task, Stack, Frame

Mapeamentos são utilizados para derivar classes a partir de conceitos ou de classes de interface (associadas a regiões da interface). Quando um conceito (ou região de interface) é mapeado em uma única classe não há necessidade de explicitar o mapeamento, que é identificado pelo fato da classe utilizar o mesmo nome do conceito. Caso contrário, o mapeamento associa ao con-

ceito (ou região de interface) as classes de objeto geradas. Para ilustrar são mostrados dois mapeamentos:

MAPPING Program : Program, BreakSet, TaskSet, SourceSet, AddressSet
MAPPING SourceRegion : SourceRegion, ExtraSection, TextRegion

A classe de objetos *Program* é mapeada em *Program* (que fica com os serviços relativos a todo o programa), *BreakSet* (controla os *breakpoints* do programa), *TaskSet* (controla as *tasks* do programa), *SourceSet* (controla os fontes do programa) e *AddressSet* (controla os endereços do programa). O mapeamento da classe de objetos de interface *SourceRegion* nas classes de objeto: *SourceRegion, ExtraSection* (que armazena informação sobre as seções extras, correspondentes aos *breakpoints*, acrescentada em um dado fonte em *display*) e *TextRegion* (que supre os serviços que permitem controlar o *display* do texto na região fonte).

No paradigma orientado a objetos, características como facilidade de manutenção e reuso passam necessariamente por uma política de nomenclatura que capture a semântica dos elementos (classes de objetos, atributos de classe, serviços, argumentos e resultados de serviços) referenciados. Nomes abreviados devem ser abolidos, com exceção dos casos em que a abreviação pertença ao jargão do domínio do problema. O nome deve corresponder à contração (eliminação de palavras supérfluas) da frase substantiva que identificaria o elemento referenciado em um texto descritivo hipotético. Na linguagem da Especificação Unificada:

- classes, serviços e eventos são nomeados por junções de palavras em letras capitais;
- outros elementos (tais como atributos e relacionamentos de classes, argumentos e resultados de serviços) são nomeados por junções de palavras minúsculas separadas por *underscores*.

Todo atributo (que associa à classe um dos valores possíveis do conjunto de valores de uma determinada propriedade da classe ou simplesmente o valor de uma informação) e todo relacionamento (que associa a classe a que pertence a uma outra classe de objetos) de uma classe de objetos são definidos em uma linha exclusiva, iniciado pelo caracter ".".

Um relacionamento é representado pela composição de três partes:

- nome do relacionamento do conceito (a partir do qual a classe foi mapeada), separado do nome do relacionamento da classe pelo caracter ":", cujo único objetivo é permitir a recuperação do conceito a partir da classe (ou das classes) na qual foi mapeado;
- seguido do prefixo do nome, separado pelo caracter "/": utilizado quando é necessário distinguir radicais de nomes;
- radical do nome: corresponde ao nome da classe de objetos com a qual existe o relacionamento (se o relacionamento for de cardinalidade múltipla, o nome da classe é precedido pela palavra chave *SET*).

No atributo não existe nome do relacionamento e o radical do nome não corresponde a um nome de uma classe de objetos e sim ao nome do atributo.

O radical do nome é único para todas as referências a um determinado tipo de informação que apareça como atributo, argumento ou resultado de serviço, ou variável transitória. O prefixo do nome, no entanto, é utilizado somente quando é necessário distinguir diferentes instâncias do mesmo tipo de informação. O objetivo básico desta forma de construir nomes é o de aumentar a legibilidade da especificação quanto à manipulação de informação; da mesma forma como serviços de classes distintas mas com mesma funcionalidade devem ter o mesmo nome. Mas a conseqüência desta construção é a de fatorar a utilização de tipos distintos de informação (não confundir com os tipos básicos das linguagens de programação) utilizados na especificação, reduzindo com isto o vocabulário utilizado e diminuindo a geração de duplicidades e inconsistências. Esta forma de representação reduz significativamente a quantidade de tipos distintos de informação que devem ser mapeados para os tipos da linguagem de implementação. A Figura 16 ilustra a composição de nomes associada a relacionamento e à informação.

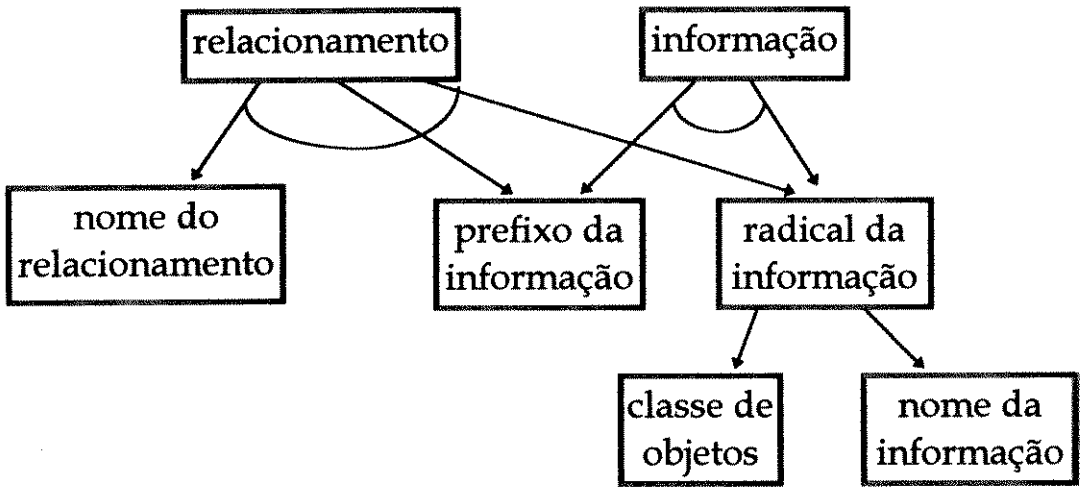

Figura 16: Composição de Nomes associados a Relacionamento e à Informação

A utilização de prefixos para os nomes leva um nome completo à seguinte forma: prefixo"/"radical. Para ilustrar a utilização de prefixos é descrita a classe *SourceRegion* (derivada da região de mesmo nome que compõe a janela de interface *ProgramWindow*) como composta de:

- dois relacionamentos:
  - *"handles : extra_sections/SET (ExtraSection)"* manipula um conjunto de seções extras de texto fonte correspondente às linhas onde são inseridas marcação de *breakpoint* ou código interpretado (*patches*);
  - *"displays_line_on : text_region/TextRegion"* mostra linhas do fonte em uma região de texto da interface;
- dois atributos:
  - *"active/source_name"* denota o nome do fonte correspondente ao arquivo fonte em *display*;

- *"back/SET (source_line)"* armazena conjunto de posições no arquivo texto (*source_line = source_name, line_number*), para retornar a posições anteriores através do comando *back*.

CLASS SourceRegion

.handles : extra_sections/SET (ExtraSection)
.displays_line_on : text_region/TextRegion
.active/source_name
.back/SET (source_line)

Pode-se observar a utilização de relacionamentos oriundos de conceitos (*handles*), prefixos de nomes (*extra_sections*) e radicais de nomes (*ExtraSection*). Os radicais de nomes formados por nomes como argumentos de *SET* indicam relacionamentos de cardinalidade múltipla (*SET (ExtraSection)*) ou atributos compostos de conjuntos de valores (*SET (source_line)*).

Existem situações nas quais uma classe de objetos requer a definição de um tipo enumerado. A definição do tipo enumerado segue a seguinte sintaxe: "**%**", nome do tipo, "=", conjunto de valores possíveis separados por vírgulas e agrupados entre "[" e "]". A classe *Task* ilustra a definição de tipos enumerados, tais como:

- tipo de *breakpoint* (*break_type*): uma parada de programa em função da execução de um *breakpoint* pode encontrar as seguintes situações:
  - *breakpoint* de usuário (*user_break*);
  - *breakpoint* artificial temporário inserido durante a execução de comandos de controle (*ExecuteToHere, Next, Step*) pertencente à *task* na qual o programa está parado (*current_task_artificial_break*);
  - breakpoint artificial temporário de outra *task*, irrelevante para a execução de comandos da *task* atual (*another_task_artificial_break*);
- referência de frame (*frame_reference*): nos comandos *Next* e *Step* a execução de uma linha de programa pode passar por uma *procedure* (denominada *stack_frame* ou simplesmente *frame*).

CLASS Task

%break_type = [user_break, current_task_artificial_break, another_task_artificial_break]
%frame_reference = [same_frame, outer_frame, inner_frame, no_symbolic_inner_frame]
%range_reference = [inner_range, outer_range]
%status = [stopped, conting, dragging, nexting, nexting_inner, stepping, stepping_inner, step_header]

O relacionamento de especialização é definido para cada classe especializada, sucedendo o nome da classe (em definição) pelo separador ":" seguido do nome da classe herdada, como ilustrado pela classes *BreakArea, TaskArea, StackArea e SourceArea* definidas como especializações da classe *StateArea*.

CLASS BreakArea : StateArea
CLASS TaskArea : StateArea
CLASS StackArea : StateArea
CLASS SourceArea : StateArea

A definição de classes genéricas, ou seja, com parametrização de tipos, permite a instanciação de classes com diferentes tipos de informação. Uma classe é definida como genérica quando seu nome é sucedido por uma lista de argumentos separados por vírgulas e agrupados entre "<"e ">". As classes *EntryBreak, EntryTask, EntryFrame* e *EntrySource* são especializações de instâncias da classe genérica *EntryStateArea.*

CLASS EntryStateArea<identifier>

%area = [break, task, stack, source]
.identifier
.area

CLASS EntryBreak : EntryStateArea<source_line>
CLASS EntryTask : EntryStateArea<task_id>
CLASS EntryFrame : EntryStateArea<stack_position>
CLASS EntrySource : EntryStateArea<source_name>

4.3 Representação de Serviços

No paradigma orientado a objetos um sistema de software é decomposto em uma cooperação de objetos em todos os níveis de abstração: da especificação dos requisitos dinâmicos e funcionais à implementação. O diagrama que melhor caracteriza esta cooperação entre objetos é o diagrama de interação de objetos utilizado para representar cenários de utilização de serviços do sistema. A construção de um cenário corresponde ao rastreamento dos serviços desencadeados pela recepção dos eventos associados a um determinado serviço oferecido pelo sistema. O que equivale a dizer que, a partir da seqüência de eventos associada a cada serviço oferecido pelo sistema e da decomposição de cada serviço do sistema (em função de serviços internos), é possível recuperar todos os cenários que definem o sistema.

As construções utilizadas para representar diagramas de interação correspondem às construções básicas para representar serviços:

- solicitação de um dado serviço;
- ordenações de solicitações de serviços.

Solicitações de serviços podem passar informação nos dois sentidos:

- mensagens contêm argumentos de informação;

- mensagens internas ao sistema podem incorporar resultados como resposta;
- mensagens externas geradas pelo sistema podem levar à recepção de eventos de resposta, separados da solicitação externa por estados, devido à assincronia na recepção de eventos.

A solicitação de um serviço pode ser representada genericamente por três componentes:

- nome do serviço solicitado;
- lista de informações que correspondem aos argumentos do serviço;
- lista de informações que correspondem aos resultados retornados pelo serviço.

A sintaxe de solicitação de serviços é ilustrada abaixo (na classe *BreakArea*). Pode-se observar que o número de argumentos retornados também pode ser zero ou maior que um (o que vale também para o número de argumentos de parâmetros) e que na soliciticação de um serviço interno o nome da classe é omitido.

CLASS BreakArea

SERVICE RemoveBreak (entry_n)
+ source_name, line_n .SourceLine (entry_n)
- BreakSet.RemoveUserBreak (source_name, line_n)
- BreakArea.RemoveBreak (source_name, line_n)
- SourceRegion.RemoveBreak (source_name, line_n)

INTERNAL SourceLine (entry_n) : source_name, line_n

O serviço *RemoveBreak* da classe *BreakArea* (mapeada da região de interface *BreakArea*) tem um argumento (*entry _n*), não retorna nenhum resultado e decompõe-se na solicitação de quatro serviços:

- o serviço interno *SourceLine* (a solicitação de um serviço interno omite o nome da classe) tem um argumento (*entry_n*) e retorna uma lista de resultados (*source_name, line_n*);
- o serviço *RemoveUserBreak* da classe *BreakSet* tem uma lista de argumentos (source_name, line_n) e não retorna nenhum resultado;
- idem para os serviços *RemoveBreak* das classes *BreakArea* e *SourceRegion.*

O *header* do serviço interno *SourceLine* é mostrado ao final para ilustrar a sintaxe de declaração de argumentos e resultados de serviços. Os caracteres "+" (que indica solicitação de leitura de informação) e "-" (que indica seqüência, ao invés de paralelismo, na solicitação de serviços), que iniciam as linhas de solicitações de serviços, serão esclarecidos em detalhe oportunamente.

O mapeamento de um serviço em subserviços utiliza operadores de composição: seqüência, seleção e iteração. Estes operadores, que correspondem aos conceitos básicos de qualquer linguagem de programação procedimental, são utilizados para descrever o mapeamento dos serviços em especificações de todos os níveis de abstração (dos requisitos à implementação). Observe-se que não se trata de incorporar construções de implementação em níveis mais altos de abstração, pois estes três operadores (ou construções equivalentes) são necessários tanto para expressar os

aspectos dinâmicos encontrados nas especificações de requisitos em linguagem natural como para representar diagramas de interação (como os utilizados na metodologia *OOSE* [Jacobson92]).

A Figura 17 ilustra as construções provenientes do diagrama de interações de objetos utilizadas para definir serviços.

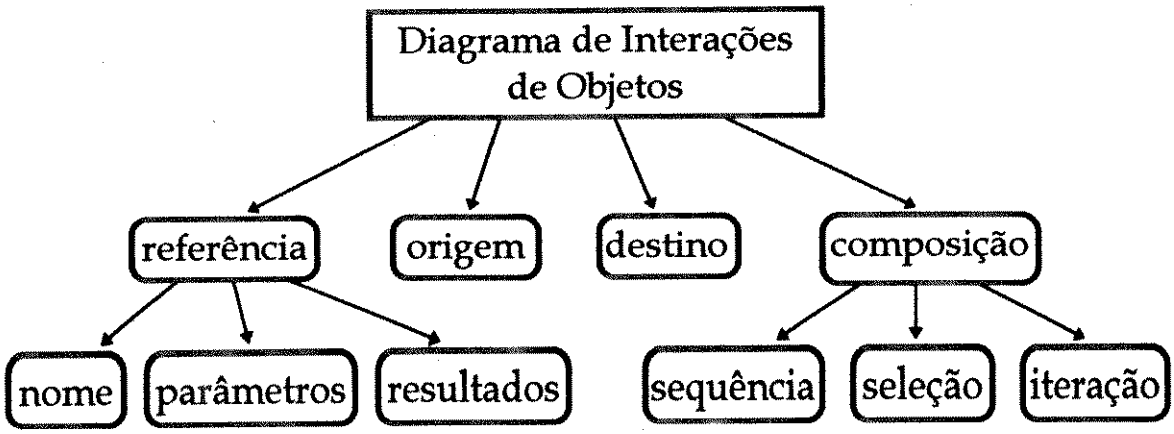

Figura 17: Construções provenientes do Diagrama de Interações de Objetos

O operador de seqüência ocorre de forma implícita nos mapeamentos de serviços, com exceção dos casos em que os operadores de seleção ou iteração forem explicitamente utilizados. O último exemplo apresentado ilustra sua utilização: o serviço *RemoveBreak* da classe *BreakArea* decompõe-se na seqüência de solicitação de quatro outros serviços.

O operador de seleção, identificado pela palavra chave *CONSTRAINT*, é ilustrado abaixo. A seleção mais externa corresponde à construção do tipo *if_then_else*, enquanto a mais interna corresponde à construção do tipo *case*. O argumento do operador de seleção corresponde a uma comparação explícita (como *CONSTRAINT symbol_kind = process, procedure*) ou implícita (comparação com nulo, como *CONSTRAINT Symbol*). É importante notar que os argumentos utilizados nos operadores de seleção e de iteração abstraem a operação realizada, evitando expressões que denotem uma mera implementação. O resultado inválido para todas as comparações anteriores (*else* para construção do tipo *if_then_else* ou *otherwise* para a construção case) é associado à palavra chave *OTHER*.

```
CLASS ProgramWindow
SERVICE DisplaySelectedSymbol (Symbol)
   CONSTRAINT Symbol
      - symbol_kind Symbol.SymbolKind
      CONSTRAINT symbol_kind
      = process, procedure
         - .SourceBlock (Symbol)
      = variable
         - .InsertVariable (Symbol)
      OTHER
```

```
        - EXCEPTION message = "no symbol permitted"
    OTHER
        - EXCEPTION message = "no symbol reached"
```

O operador de iteração, identificado pela palavra chave *ITERATION*, é ilustrado abaixo. Pode-se observar também a utilização do operador complementar representado pela palavra chave *ALL* que indica iteração por todos os elementos do conjunto especificado no argumento da iteração (que representa um conjunto denominado *user_breaks* composto de *source_lines*).

```
CLASS BreakSet

SERVICE InstallUserBreaks
    ITERATION ALL (user_breaks/SET (source_line))
        + Address SourceSet.Address (source_line)
        - .InsertBreak (Address)
```

Pode-se observar que a linguagem de representação da Especificação Unificada utiliza itemização em substituição às usuais palavras chaves para determinar início e fim de um determinado bloco de seleção ou de iteração. O objetivo desta simplificação sintática foi tornar a linguagem mais compacta, menos carregada de palavras chaves e mais próxima de suas construções equivalentes em linguagem natural (no caso explícito dos operadores de seleção e iteração).

Para representar condições na forma negativa, ou comparação por diferença, utiliza-se um operador de negação definido pelo caracter "#". Pode-se observar no exemplo abaixo que o operador de negação pode ser utilizado de duas formas. A condição do primeiro operador de seleção (*CONSTRAINT program/status # loaded*) utiliza o operador de negação como uma contração de uma comparação seguida de negação (= #). No segundo operador de seleção (*CONSTRAINT #Break*) o operador de negação é utilizado para negar a condição.

```
CLASS BreakSet

SERVICE InsertBreak (Address)
    + program/status Program.Status
    CONSTRAINT program/status # loaded
        + Break program_breaks/SET [Address]
        CONSTRAINT #Break
            * program_breaks/SET (Break) = + Break (Address)
        - Break.InsertOcurrence
        - .RemoveBreakData_WhenNoOcurrences (Break)
```

Em geral, desvios ou seleções de seqüência são motivados por requisitos que devem ser explicitados. O operador de seleção (cujo caso particular é um único desvio de seqüência) é sem-

pre considerado um ponto propício para o ancoramento de requisitos adicionais (uma vez que a seqüência de serviços deriva da especificação de requisitos) representando:

- uma restrição aplicada à execução de um serviço;
- ou seleção de alternativa de execução do serviço.

O exemplo abaixo ilustra a vinculação de um requisito (referenciado pela palavra chave *REQUIREMENT*) a uma seleção (representada por *CONSTRAINT Line*) e significa que a consistência de uma linha executável (objeto *Line* não nulo) correspondente a uma linha no programa fonte (*source _line* agrupa nome do fonte e número da linha) decorre do requisito que estabelece que um *breakpoint* só pode ser inserido em uma linha executável do programa fonte. Pode-se observar que a declaração de requisito é indentada de forma a se referir a todas as declarações englobadas pela seleção.

```
SERVICE InsertBreak (source_line)
   + Line SourceSet.Line (source_line)
   CONSTRAINT Line
      REQUIREMENT BreakLine_MustBe_Executable
      - BreakSet.InsertUserBreak (source_line)
      - BreakArea.InsertBreak (source_line)
      - SourceRegion.InsertBreak (source_line)
```

A vinculação de requisitos, no entanto, não se restringe aos operadores de seleção e iteração; em solicitações de serviço motivadas por requisitos, a vinculação também deve ser explicitada. O exemplo abaixo ilustra vinculações de requisitos nas duas situações.

As vinculações de requisitos a solicitações de serviços são as seguintes (pode-se notar que a declaração do requisito aparece indentada em relação à solicitação de serviço a qual se refere):

- solicitação do serviço interno (da classe *TaskArea*) *UpdateFocus* justificada pelo requisito *TaskArea_HighlightsSelectedTask* (tarefa selecionada deve ser distinguida na TaskArea);
- solicitação do serviço *Update* da classe *StackArea*, justificada pelo requisito *StackArea_DisplaysStack_SelectedTask* (*StackArea* deve mostrar pilha de execução correspondente à tarefa selecionada).

As vinculações de requisitos a seleções de serviços são as seguintes (pode-se notar que a declaração de requisito aparece indentada em relação à declaração de seleção, de forma a referir-se a todas as declarações internas à seleção):

- seqüência de serviços, decorrente da existência de linha fonte, justificada pelo requisito *SourceRegion_Focus_CurrentLine* (*SourceRegion* deve distinguir linha corrente do fonte em *display*);
- seqüência de serviços, decorrente da não existência de linha fonte, justificada pelo requisito *SourceRegion_ClearFocus_CurrentLine_NoSymbolic* (*SourceRegion* sem foco quando linha corrente não tem representação simbólica).

CLASS TaskArea

```
SERVICE SelectCurrentTask
   + task_id TaskSet.CurrentTask
   - .ChangeSelectedTask (task_id)

SERVICE SelectedTask (entry_n)
   + task_id .Task (entry_n)
   - .ChangeSelectedTask (task_id)

INTERNAL ChangeSelectedTask (task_id)
   - TaskSet.ChangeSelectedTask (task_id)
   - .UpdateFocus
      REQUIREMENT TaskArea_HighlightsSelectedTask
   - StackArea.Update
      REQUIREMENT StackArea_DisplaysStack_SelectedTask
   + source_line .CurrentLine_SelectedFrame_SelectedTask
   CONSTRAINT source_line
      REQUIREMENT SourceRegion_Focus_CurrentLine
      - SourceRegion.UpdateSourceFocus (source_line)
   OTHER
      REQUIREMENT SourceRegion_ClearFocus_CurrentLine_NoSymbolic
      - SourceRegion.ClearFocus
```

Pode-se observar do exemplo acima que separadores (*underscores*) podem ser utilizados (entre palavras capitais) na representação de nomes de requisitos. Constituídos de nomes longos, derivados de simplicações de frases, os nomes de requisitos tornam-se mais legíveis com a adição dos separadores.

Requisitos também podem ser explicitados como motivações para decisões de projeto (em geral, associados a requisitos de desempenho). O exemplo abaixo ilustra a vinculação de um requisito de projeto, justificado por um requisito de especificação. A solicitação do serviço *Update* à classe *TaskSet* é justificada pelo requisito de projeto (caracterizado pela palavra chave *DESIGN*) *TaskSetUpdating_OnlyWhenRequired* (atualizar o conjunto de *tasks* somente quando tal informação for requisitada pela interface) que corresponde a uma solução para o requisito de especificação *MinimizeHostTarget_InformationFlow* (fluxo de informação entre a máquina hospedeira e a máquina alvo deve ser minizado). Pode-se observar que a declaração do requisito não aparece indentada em relação à declaração de requisito de projeto, uma vez que vários requisitos podem ser utilizados para a justificação, o que torna a indentação indesejável.

```
CLASS TaskArea

SERVICE ExpandEntries
   - TaskSet.Update
      DESIGN TaskSetUpdating_OnlyWhenRequired
      REQUIREMENT MinimizeHostTarget_InformationFlow
   + identifiers/SET (task_id) TaskSet.Identifiers
   ITERATION ALL (identifiers/SET (task_id))
      - .InsertEntry (task_id)
```

- .*UpdateSelectedTask*

Operadores de iteração muitas vezes atuam sobre elementos de um dado conjunto (como ilustrado no exemplo do serviço *InstallUserBreaks* da classe *BreakSet*). Para isto é utilizada uma classe abstrata denominada Conjunto (referenciada pela palavra chave *SET*) sobre a qual é aplicado qualquer método de interação, inserção ou remoção de elementos. O exemplo abaixo ilustra a inserção (= +) e a remoção (= -) do elemento *source_line* no conjunto denominado *user_breaks/SET* (composto de *source_lines*). No nível de abstração de projeto cada utilização desta classe deverá ser especializada em tipos abstratos de dados (em geral, componentes disponíveis em bibliotecas). É importante notar que esta abstração pode ser mapeada com facilidade em linguagens de implementação orientadas a objetos que suportam o mecanismo de *overloading* de operadores (que permite redefinir operadores da linguagem).

CLASS BreakSet

.is_composed_by : user_breaks/SET (source_line)
.is_composed_by : program_breaks/SET (Break : Address)

SERVICE InsertUserBreak (source_line)
+ *Address SourceSet.Address (source_line)*
- *.InsertBreak (Address)*
* *user_breaks/SET(source_line) = + source_line*

SERVICE RemoveUserBreak (source_line)
+ *Address SourceSet.Address (source_line)*
- *.RemoveBreak (Address)*
* *user_breaks/SET(source_line) = - source_line*

O diagrama de fluxo de dados tem sido utilizado para representar o comportamento funcional de um sistema de software. No paradigma orientado a objetos os dados são encapsulados nas classes de objetos de tal forma que a leitura ou alteração dos mesmos é feita mediante o protocolo determinado pelos serviços providos pela classes. Como conseqüência, o fluxo de dados do sistema ocorre somente através da interação entre os objetos do sistema, o que privilegia o diagrama de interação de objetos do ponto de vista de representação, uma vez que consegue expressar tanto o aspecto dinâmico do sistema, como o fluxo de dados correspondente a seu aspecto funcional. A Figura 18 ilustra o papel fundamental do diagrama de interações de objetos.

Mas, embora os diagramas de interação de objetos capturem todo o fluxo de dados do sistema, eles não representam a obtenção, armazenamento e descarte de informação dos depósitos de dados como ocorre nos diagramas de fluxo de dados. É importante caracterizar tais operações uma vez que a informação armazenada (nos depósitos de dados) é o guia fundamental para a decomposição do sistema nas classes de objetos (utilizadas nos diagramas de interação de objetos). A Figura 19 ilustra as construções da Especificação Unificada provenientes do DFD. Partindo des-

sa premissa, a decomposição de um serviço na linguagem de representação da Especificação Unificada distingue a solicitação de serviços nos seguintes tipos:

- obtenção de informação: normalmente associada à preparação de argumentos para solicitação de serviços (identificada por uma linha iniciada com o caracter "+");
- armazenamento ou descarte de informação (idem com o caracter "*");
- solicitação de serviços (idem com o caracter "-").

A Figura 20 ilustra a separação entre preparação de argumentos e efetiva solicitação de um serviço.

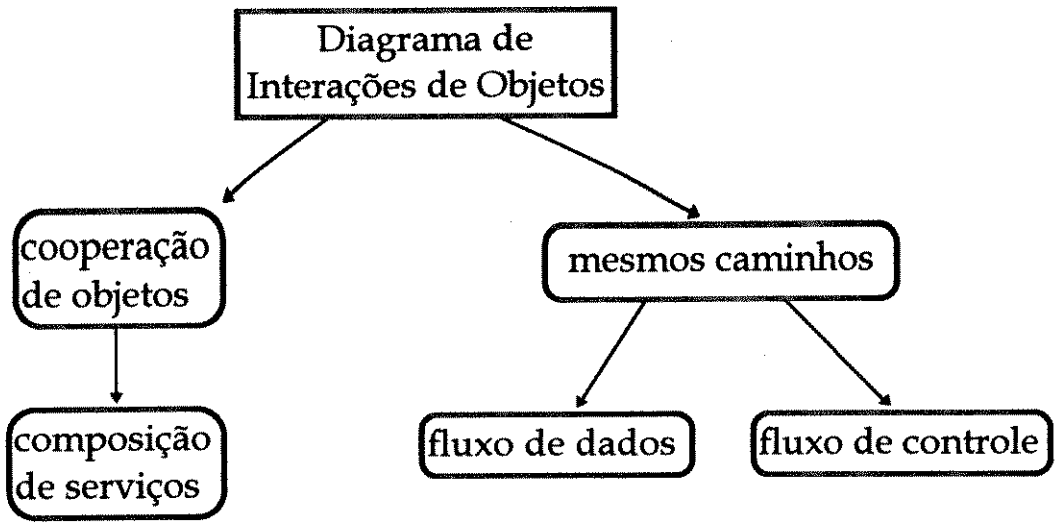

Figura 18: O Papel Fundamental do Diagrama de Interações de Objetos

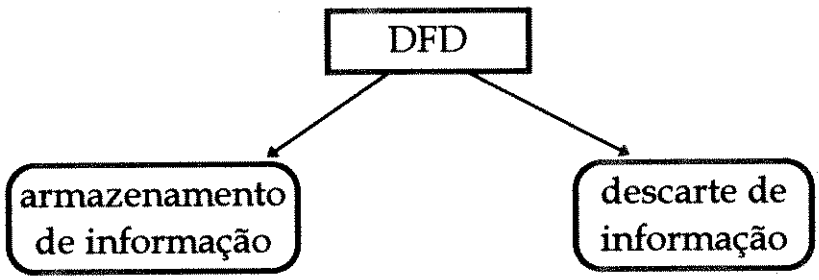

Figura 19: Construções provenientes do DFD

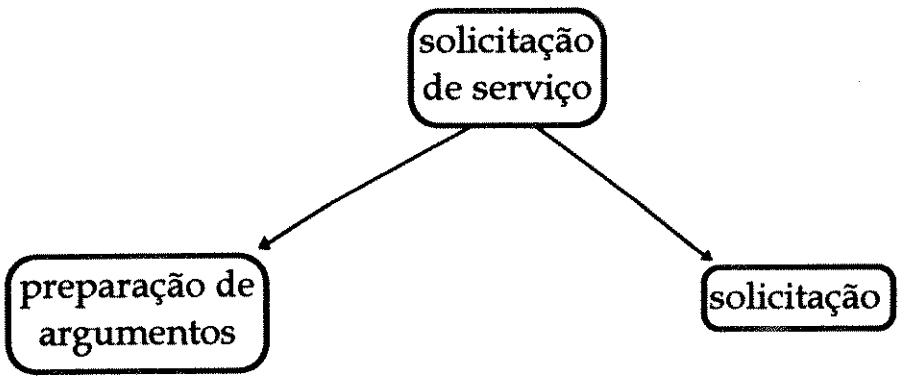

Figura 20: A Fatoração da Preparação de Argumentos na Solicitação de um Serviço

O exemplo anterior ilustra esta separação na solicitação dos serviços:

- obtenção da informação *Address* através do serviço *Address* da classe *SourceSet*;
- solicitação de execução do serviços *InsertBreak* e *RemoveBreak* da própria classe (*BreakSet*);
- armazenamento da informação *source_line* no conjunto *user_breaks*;
- descarte da informação *source_line* no conjunto *user_breaks*.

Essa diferenciação de serviços, além de evidenciar o armazenamento ou descarte de informação (o que é fundamental para a determinação das classes de objetos), separa solicitação de serviços secundários (utilizados para compor os argumentos) dos serviços principais a serem solicitados (essa separação tem um papel preponderante para qualificar visões de diferentes níveis de abstração).

Quando o descarte não corresponde à remoção de um elemento de um conjunto é necessário utilizar o operador de negação. O exemplo abaixo ilustra as duas situações:

- remoção de um elemento de conjunto (* *active_tasks/SET (Task)* = - ~*Task*);
- remoção de informações isoladas (*selected/task_id, current/task_id, debug_phase*).

Pode-se notar que a remoção de um elemento (utilizada no exemplo) de um conjunto engloba também a destruição do objeto correspondente.

```
CLASS TaskSet

SERVICE Reset
   ITERATION ALL (tasks/SET (Task))
      * active_tasks/SET (Task) = - ~Task
   * #selected/task_id
   * #current/task_id
   * #debug_phase
```

O caracter "~", que aparece no exemplo acima precedendo o nome de um construtor é utilizado para denotar um destrutor. Um construtor corresponde, em C++, a um método definido com o mesmo nome da classe, sendo chamado automaticamente durante a construção de um objeto da classe. Um destrutor é definido, em C++, com o nome da classe precedido do caracter "~" e é chamado ao final do tempo de vida de um objeto da classe.

A fatoração (através da criação de serviços que encapsulem procedimentos comuns a outros serviços) e a parametrização dos serviços (através da criação de interfaces genéricas que facilitem a instanciação de serviços diferenciados unicamente pelo valor dos seus parâmetros) é são buscadas a cada decomposição. Serviços são então classificados como:

- primitivos: realizam serviços elementares (armazenam, descartam, atualizam ou convertem uma dada informação);
- compostos: delegam serviços primitivos ou compostos (fatoram uso de serviços ou encapsulam visibilidade de objetos internos a uma camada);

- definições: trechos comuns de serviços que são convenientes do ponto de vista de fatoração da representação (de serviços de uma dada classe) mas cuja coesão funcional não justifica o status de serviço.

Definições também são utilizadas para encapsular serviços de níveis de abstração mais baixos, permitindo ao mecanismo de recuperação de visões diferenciá-los de serviços a serem recuperados. São distinguidas dos serviços (públicos e internos) através da palavra chave DEF, como no exemplo abaixo. Pode-se notar que a utililização do conceito de definições permite capturar uma seqüência de declarações cujo agrupamento não apresenta uma coesão funcional necessária para caracterizar um serviço.

CLASS ProgramMenu

SERVICE Cont
- *Program.ContinueExecution*
- *.EnableStop*

SERVICE Next
- *Program.NextLineOver*
- *.EnableStop*

SERVICE Step
- *Program.NextLineInto*
- *.EnableStop*

DEF EnableStop
- *.DisableAll*
- *.Enable (stop)*

Na linguagem de representação da Especificação Unificada a escolha dos nomes adequados para os nomes deve ser muito criteriosa de forma a expressar precisamente a semântica de cada elemento representado. Neste contexto, definições devem ser utilizadas não apenas com o objetivo de fatoração mas também visando abstrair a referência a subseqüências de serviços.

A utilização de dados internos a um serviço pode ser necessária em níveis mais baixos de abstração. Tanto a iniciação do dado (que corresponde também à sua própria definição) quanto suas atualizações são vistas no contexto de um serviço como definições de um valor transitório (cuja tempo de vida termina após a execução do serviço). Definições de dados internos a serviços são caracterizadas pelo caracter "%" como no exemplo abaixo. Pode-se observar a possibilidade de utilização de uma expressão na atribuição associada a uma definição.

CLASS EntryTask

INTERNAL Description : description
+ *process_name TaskSet.ProcessName (task_id)*
CONSTRAINT process_name
+ *source_name, line_n TaskSet.ProcessSourceLine (task_id)*

```
    % description
        = task_id + process_name + source_name + ":" + line_n
OTHER
    % description = task_id
```

Com o objetivo de isolar classes de diferentes níveis de abstração é acrescentado o conceito de camadas de serviços. A sintaxe para a declaração de camadas de serviços é a seguinte:

SERVICE LAYER nome

4.4 Conclusões

A linguagem de representação da metodologia Especificação Unificada dispõe de construções que suportam a representação dos requisitos (com atenção especial à especificação da interface gráfica com usuários) e a representação das classes e serviços de análise e projeto. Construções utilizadas na Especificação Unificada provêem do diagrama de interações de objetos, DFD, MER e diagrama de transição de estados. A Figura 21 ilustra as construções provenientes do MER e a Figura 22 ilustra as construções provenientes do diagrama de transição de estados.

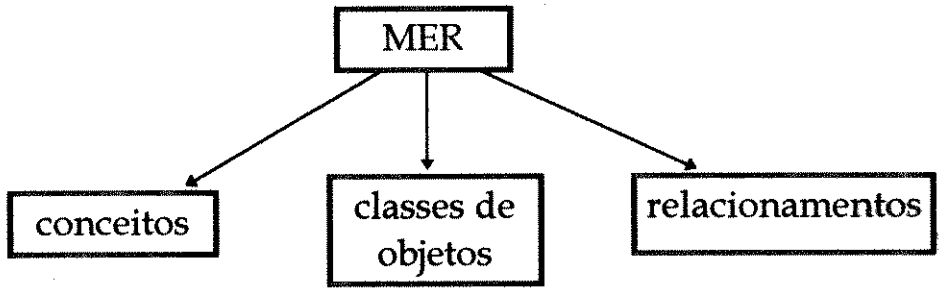

Figura 21: Construções provenientes do MER

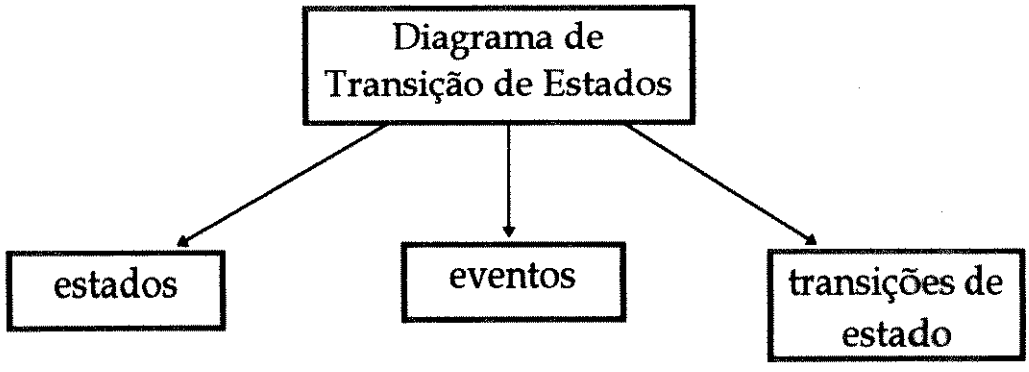

Figura 22: Construções provenientes do Diagrama de Transição de Estados

Classes de projeto são utilizadas, em geral, a partir da abstração da classe especial *SET*. O detalhamento das classes de projeto que encapsulam estruturas de dados é função da necessidade imposta pela indisponibilidade de componentes de software com os serviços solicitados à classe hipotética *SET*. Construções da Especificação Unificada são provenientes de abstrações adicionais: vinculação de requisitos, definições e camadas de serviços. A Figura 23 ilustra as construções da Especificação Unificada provenientes de tais abstrações.

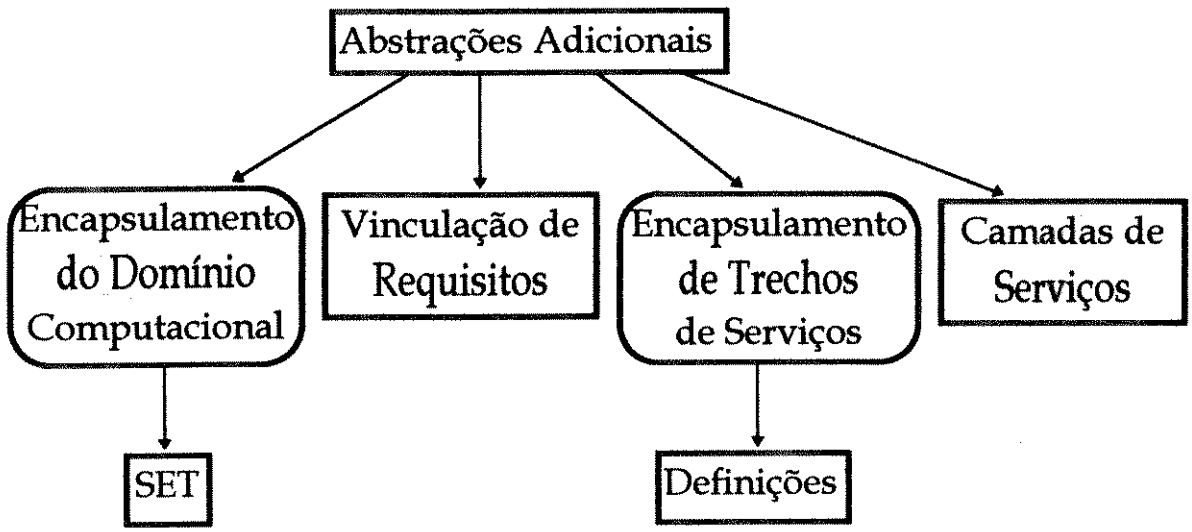

Figura 23: Construções provenientes de Abstrações Adicionais

A informação nos níveis mais altos de abstração correspondentes à especificação de requisitos e análise deve ser extraída da Especificação Unificada a partir do mecanismo de Recuperação de Visões.

5. Heurísticas da Especificação Unificada

As subseções seguintes descrevem as heurísticas propostas na Especificação Unificada para subsidiar a especificação de requisitos e a concepção das classes de objetos e seus serviços.

Os passos propostos para a especificação de requisitos decorrem da sua representação através de conceitos, serviços e requisitos adicionais.

Após a especificação de requisitos, as principais classes de objetos da interface são mapeadas a partir das regiões que compõem o layout das janelas que fazem parte da interface. As classes restantes da interface derivam da necessidade de tradução de informação inerente à interface (acoplamento de formas diferentes de informação entre o meio externo e o resto do sistema). A concepção das classes de objetos e principais serviços da interface determina os serviços requeridos e a informação a ser tratada pelo resto do sistema, cujas classes de objetos são mapeadas a partir dos conceitos (estabelecidos na representação dos requisitos) segundo a necessidade de armazenamento e manipulação de informação.

Uma heurística básica, fundamentada no tratamento de informação, é proposta para determinar a alocação de controle na definição dos serviços. Heurísticas adicionais são propostas para refiná-la.

5.1 Passos da Especificação de Requisitos

As heurísticas associadas à especificação de requisitos são uma decorrência da representação de requisitos através de conceitos, serviços e requisitos adicionais, podendo ser resumidas nos seguintes passos:

- listar os nomes dos conceitos relevantes à descrição do sistema;
- descrever cada conceito a partir dos seus relacionamentos com os demais (explicitando, para cada relacionamento, os nomes do relacionamento e do conceito associado e a indicação da cardinalidade);
- utilizar os relacionamentos de especialização sempre que relevante ao domínio do sistema (este procedimento não é prematuro, porque os conceitos não serão mecanicamente transportados para os níveis de abstração seguintes à especificação de requisitos);
- listar os serviços a serem oferecidos pelo sistema para cada um dos diferentes papéis representados pelos seus usuários (esta heurística foi proposta pela metodologia OOSE [Jacobson92]);
- listar as restrições e propriedades de serviços ou de conceitos como requisitos adicionais;
- descrever os requisitos adicionais através de nomes, referenciando conceitos e serviços (este é um ponto de revisão para os conceitos);

- descrever o *layout* de cada uma das janelas de interface especificando as diferentes visões de acesso requeridas para entrada e saída de informação em determinadas regiões;
- descrever os eventos primitivos associados à interface, bem como menus associados à ativação dos eventos;
- encapsular eventos primitivos através da descrição de eventos compostos, que serão utilizados para descrever a ativação e troca de informação com os serviços do sistema;
- descrever os serviços a partir dos nomes de eventos, respostas e subserviços (cujos nomes ao referenciar conceitos, podem levar à necessidade de conceitos adicionais), dos operadores de composição (seqüência, seleção e iteração) e dos serviços associados à recepção e envio de informação através das regiões das janelas da interface.

5.2 Mapeamento dos Requisitos

Na metodologia Especificação Unificada as classes de objetos que acoplam informação externa correspondem às janelas e algumas de suas regiões (camadas ou partições) definidas nos *layouts*. As classes de objetos principais da interface, ilustradas na Seção 5.1.2.1, são obtidas da seguinte forma:

- do *LAYOUT* da janela *ProgramWindow* são extraídas as classes de objetos: *ProgramWindow, SourceRegion, StateRegion, BreakArea, TaskArea, StackArea, SourceArea, TypingLine, ProgramMenu* e *MessageLine;*
- do *LAYOUT* da janela *DirectoryWindow* são extraídas as classes de objetos: *DirectoryWindow, DirectoryRegion* e *MessageLine.*

Cada um dos serviços descritos na especificação de requisitos é reescrito em função dos serviços prestados pela interface. Os serviços adicionais necessários são associados a serviços internos (ao núcleo do sistema) de mais alto nível.

A decomposição dos serviços referentes ao núcleo do sistema (ou seja, excluindo-se a interface) é fundamentada em dois princípios:

- manipulação de informação pela classe de objetos que a encapsula;
- decomposição de serviços em serviços primitivos.

As classes de objetos do núcleo do sistema são concebidas a partir de um mapeamento dos conceitos e segundo a necessidade de armazenamento de informação (recebida ou obtida em decorrência do tratamento de um dado evento, mas que será necessária durante o tratamento de outros eventos, associados ao mesmo serviço ou a outros serviços). A obtenção de serviços primitivos apenas caracteriza o fim do processo de decomposição de um dado serviço.

A Figura 24 ilustra o mapeamento dos conceitos e regiões de interface que compõem a especificação de requisitos em classes de objetos.

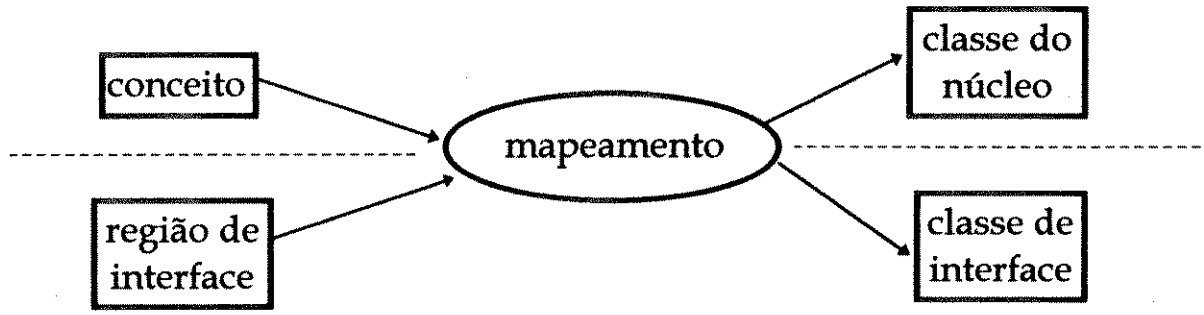

Figura 24: Mapeamento dos Requisitos em Classes de Objetos

O ponto de partida para continuação da decomposição dos serviços do núcleo do sistema engloba os seguintes elementos:

- conceitos;
- informação a ser armazenada durante o tratamento de um evento, para ser consumida durante o tratamento de outro evento do mesmo serviço ou de outro serviço;
- requisitos adicionais;
- serviços de interface;
- demais serviços prestados pelo sistema.

Após o mapeamento das classes de interface ocorre um processo iterativo; ou seja, para cada decomposição de serviço que utiliza informação para a qual ainda não foi identificada uma classe de objetos responsável o processo de escolha da classe adequada deverá ser repetido. Este processo consiste em procurar, para cada item de informação identificado, um conceito que lhe sirva de classe de objetos. A não identificação de um conceito candidato pode levar à necessidade de rever a descrição dos conceitos.

Como primeira aproximação (ilustrada no exemplo abaixo), obtém-se uma classe de objetos especificamente para os serviços prestados pelo sistema, para os quais é descrito:

- conversão de informação recebida da interface (*source_line .SourceLine (entry_n)*);
- delegação de serviço no mais alto nível para a classe (ou mais de uma, se necessário) detentora da informação (*BreakSet.RemoveUserBreak (source_line)*);
- encaminhamento de respostas, mensagens ou novas solicitações a uma ou mais classes de interface (demais serviços).

CLASS BreakArea

SERVICE RemoveBreak (entry_n)
+ *source_line .SourceLine (entry_n)*
- *BreakSet.RemoveUserBreak (source_line)*
- *.RemoveBreak (source_line)*
- *SourceRegion.RemoveBreak (source_line)*

Após a decomposição dos serviços prestados pelo sistema as classes de interface são detalhadas para armazenar informação entre diferentes serviços, resultando na concepção de classes adicionais que não têm interface direta com o recebimento ou envio de informação externa. Basicamente a informação armazenada pelas classes de interface é a necessária para converter informação de entrada (ou saída) do sistema para informação entendida pelo sistema (ou informação entendida pelo usuário). Tal necessidade pode levar a conjuntos de informações criando a necessidade de novas classes de interface, agregadas e controladas pelas classes de interface já existentes.

À medida que os serviços são decompostos através da delegação de outros serviços ou da execução de serviços elementares, é necessário justificar a manipulação de informação em função dos requisitos adicionais. A decomposição de serviços busca delegar subserviços (sempre que possível) ou executá-los (quando associados a informações manipuladas pela própria classe de objetos). Mas, além desta heurística, é preciso explicitar os requisitos que implicitamente direcionam construções (seqüências, seleções, iterações, solicitações de serviços, armazenamento ou descarte de informação) utilizadas na decomposição. Dessa forma, a referência ao requisito é ancorada à construção que ele justifica, o que se torna bem mais eficiente do ponto de vista de rastreamento do que a associação de comentários.

Em níveis mais baixos de abstração podem aparecer classes adicionais, devido a otimizações ou necessidades de encapsulamento de protocolo com sistemas externos ou elementos do ambiente computacional (tais como: sistema operacional e banco de dados). O detalhamento de tais classes deve ser adiado até o último nível de abstração.

Atributos adicionados são em geral justificados por requisitos de projeto de otimização de acesso à informação; mas, qualquer que seja a motivação, um requisito de projeto deve ser ancorado à definição desses atributos.

Tipos abstratos de dados são utilizados de forma abstrata através de uma classe denominada genericamente de classe Conjunto, de forma a isolar os objetos do domínio do problema dos objetos do mundo computacional. Caso os serviços utilizados não permitam a identificação de um tipo abstrato de dado existente em biblioteca, será necessário especificar uma classe de projeto que suporte os serviços utilizados.

5.3 Alocação de Controle

Nos sistemas reativos, a especificação dos serviços (associados aos eventos da interface) atua como heurística de decomposição do sistema (especialmente quando o sistema é constituído por um conjunto de serviços não complexos). No caso de sistemas não reativos existe um único serviço, em geral complexo, cujo detalhamento após a especificação de requisitos é fundamental para sua decomposição.

Do ponto de vista da interação com a interface, um sistema de software pode ser definido como um sistema composto de parte reativa (associada a sua interface com usuários) e parte não reativa (correspondente ao núcleo do sistema). A decomposição de um sistema genérico pode ser dividida em duas etapas:

- decomposição de sua parte reativa: basicamente corresponde ao agrupamento dos eventos externos no conjunto de serviços prestados pelo sistema;
- decomposição de sua parte não reativa: corresponde à decomposição dos serviços decorrentes da decomposição da parte reativa (ou serviços do núcleo do sistema).

A Figura 25 ilustra a visão de um sistema de software como composto das partes reativa e não reativa.

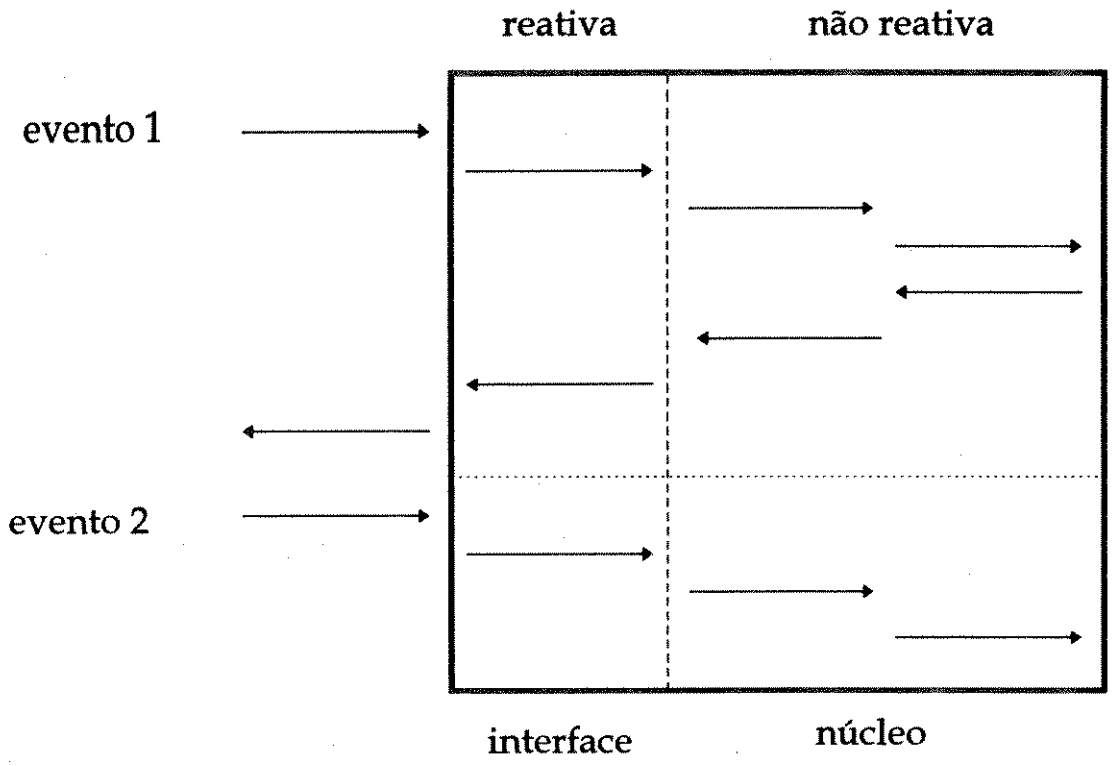

Figura 25: Sistemas de Software do Ponto de Vista Reativo

Cada um dos serviços do núcleo do sistema pode ser especificado com uma ordenação de serviços em alto nível (omitindo-se a priori a solicitação de informação necessária à preparação de argumentos de cada um desses serviços), cuja alocação de controle é definida em função da caracterização de uma das situações abaixo (na ordem em que estão apresentadas):

- assumir execução: que por sua vez está associada a uma das atividades abaixo:
  - executar: quando a classe do serviço é responsável pela informação;
  - controlar: quando a classe do serviço agrega a classe responsável pela informação;

- delegar: quando o serviço (ou subserviço, que corresponde a um dos serviços nos quais o serviço em questão foi decomposto) pode ser delegado à classe responsável pela informação ou controladora da mesma;
- coordenar: quando a execução do serviço implica
  - na coordenação de várias classes, responsáveis pelas informações (ou controladoras das classes responsáveis);
  - ou na coordenação de serviços externos com restrições de informação ou com requisitos de funcionalidade primitiva, obrigando a uma coordenação em um nível mais baixo de abstração.

A Figura 26 ilustra a heurística base de alocação de controle da Especificação Unificada.

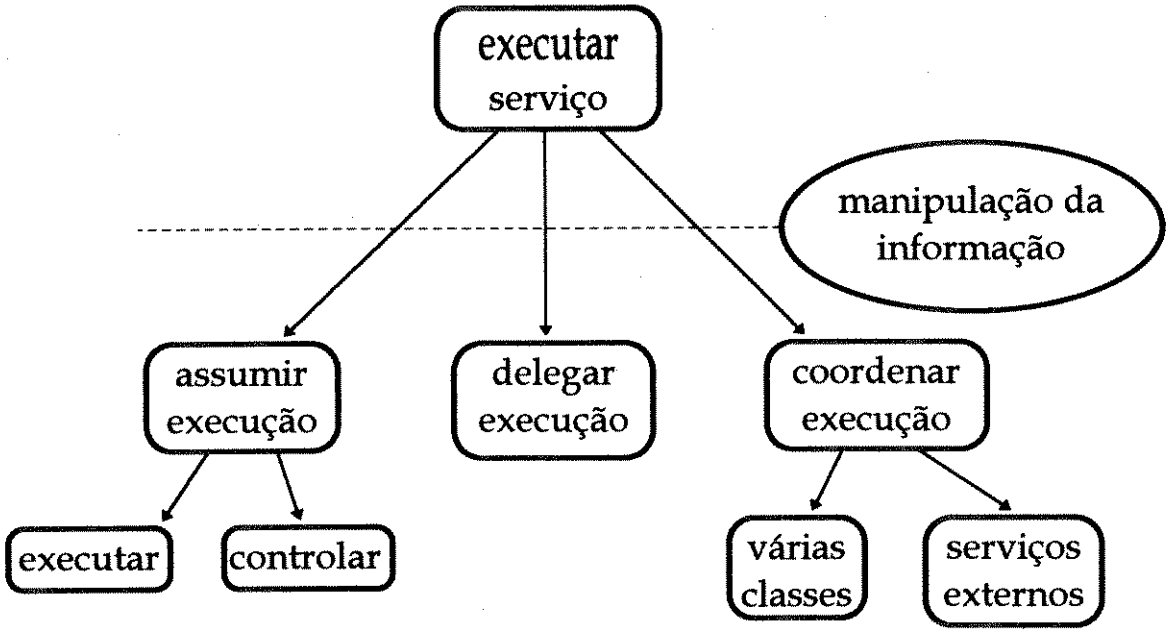

Figura 26: Heurística Base de Alocação de Controle

A situação na qual uma classe de objetos assume a execução do serviço é ilustrada através do serviço *ChangeSelectedTask* da classe *TaskSet*. Este comando é ativado quando o usuário solicita a mudança de foco de atenção da interface em relação às *tasks*; ou seja, seleciona outra *task* para a qual a interface deverá mostrar a pilha de execução e sobre a qual atuarão os comandos de controle de execução (*next e step*). A decomposição do serviço *ChangeSelectedTask* implica, do ponto de vista da classe *TaskSet*, em comandar a atualização da classe agregada *Task* (*Task.Update*) e em executar a atualização de informação sobre a qual é responsável (*selected/task_id = task_id*).

CLASS TaskSet

SERVICE ChangeSelectedTask (task_id)
+ *Task active_tasks/SET [task_id]*
- *Task.Update*

** selected/task_id = task_id*

A delegação de execução pode ser ilustrada na decomposição do serviço *InsertBreak* da classe *SourceRegion,* ativado pela solicitação do usuário de inserção de um *breakpoint* em uma linha do programa fonte. A decomposição deste serviço implica:

- delegação do serviço *InsertUserBreak* à classe *BreakSet,* que comanda a inserção do *breakpoint* no programa sob depuração na máquina alvo;
- delegação do serviço *InsertBreak* à classe *BreakArea,* que comanda a inserção de uma linha correspondente ao breakpoint na *BreakArea;*
- execução do serviço *InsertBreak,* que comanda a inserção de uma linha com indição de *breakpoint* na própria *SourceRegion.*

CLASS SourceRegion

SERVICE InsertBreak (source_line)
+ Line SourceSet.Line (source_line)
CONSTRAINT Line
REQUIREMENT BreakLine_MustBe_Executable
- BreakSet.InsertUserBreak (source_line)
- BreakArea.InsertBreak (source_line)
- .InsertBreak (source_line)

A coordenação de várias classes pode ser ilustrada pelo serviço *BreakStop* (juntamente com as definições que ele utiliza: *IfStopped, Stopped* e *UpdateStateRegion*) da classe *ProgramWindow.* Quando o programa em execução encontra um *breakpoint* é gerado um evento na interface a ser recebido pela classe *ProgramWindow,* a qual repassa o evento à classe *Program* (que deverá comandar o seu tratamento) e coordena (em função do estado do programa) a atualização das classes de interface que compõem a classe *ProgramWindow* (*SourceRegion, ProgramMenu, TaskArea, StackArea, SourceArea*).

CLASS ProgramWindow

SERVICE BreakStop (task_id, cpu)
- Program.BreakStop (task_id, cpu)
- .IfStopped

DEF IfStopped
+ program/status Program.Status
CONSTRAINT STATE (Program) program/status = stopped
.Stopped

DEF Stopped
REQUIREMENT BreakStop_UpdatesUserInterface
- .UpdateStateRegion
+ source_line .CurrentLine_SelectedFrame_SelectedTask
- SourceRegion.UpdateSourceFocus (source_line)

- *ProgramMenu.Enable ([cont, next, step, abort, detach])*

DEF UpdateStateRegion
- *TaskArea.Update*
- *StackArea.Update*
- *SourceArea.Update*

A coordenação de serviços externos pode ser ilustrada pelo serviço *NextLineIntoCall* (ativado pelo usuário através do comando de controle de execução *Step*) da classe *Task*. O depurador simbólico *CHILL* é composto basicamente de um depurador remoto (cuja máquina hospedeira é uma estação de trabalho) que interage com um monitor (que controla a execução do programa em depuração na máquina alvo). A construção do depurador remoto teve como objetivo aliviar a máquina alvo do espaço ocupado pelo antigo depurador (não remoto), razão pela qual o monitor que executa na máquina alvo deve fornecer um conjunto de primitivas que permitam o controle da execução do programa pelo depurador remoto. Em função da restrições de funcionalidade e de informações do monitor (as informações provenientes da tabela de símbolos geradas pelo compilador *CHILL* são acessadas somente pelo depurador remoto), a execução do comando *Step* não pode ser delegada em alto nível para o monitor.

Através do comando *Step* o usuário indica ao depurador que o programa deve executar até a próxima linha (do programa fonte) parando na primeira linha executável de um procedimento interno se for o caso. Uma linha genérica do programa fonte pode conter vários comandos na linguagem CHILL cuja execução pode levar a um procedimento interno ou a um salto na seqüência de instruções no programa executando na máquina alvo. O depurador calcula os endereços das instruções correspondentes à linha corrente e a próxima linha e solicita ao monitor que o avise (através de interrupção) quando a execução sair da seqüência de instruções delimitada pelos dois endereços. Ao receber a interrupção correspondente, o depurador precisa analisar várias alternativas: volta ao procedimento chamador, pára na próxima linha do procedimento corrente, pára na primeira instrução de um procedimento interno (que não corresponde à uma linha executável) ou pára em uma chamada recursiva do próprio procedimento. Os serviços associados à execução do comando *Step* estão detalhados na Seção 7.3.1, mas o que importa saber do ponto de vista da coordenação de serviços externos é que o depurador remoto, impossibilitado de delegar a execução do comando *Step* ao monitor, é obrigado a montar uma máquina de estados para controlar sua execução através de procedimentos primitivos.

Após a identificação da atividade relacionada a cada um dos serviços são determinadas as solicitacões de informação (que implicam a execução de serviços secundários) associadas à preparação de argumentos para cada um desses serviços.

A heurística apresentada acima é base para a decomposição e alocação de controle em sistemas genéricos. Heurísticas adicionais são apresentadas abaixo, com o propósito de refinar a heurística base.

A decomposição de um serviço utiliza os operadores de composição (seqüência, seleção e iteração). A seqüência pode ser determinada pelas seguintes heurísticas adicionais:

- antecipar serviços que consistem informação de entrada (nas classes de objetos que acoplam informação com usuários);
- antecipar serviços que determinam a viabilidade da execução do serviço;
- e levar em conta a necessidade de precedência de informação: informação utilizada na composição de outra, ou com argumento para solicitação de um serviço.

A seqüência de serviços *InsertBreak* (inserção de *breakpoint* no programa a partir de escolha de uma linha no programa fonte) ilustra a utilização das heurísticas de ordenação da seqüência de comandos. A primeira restrição aplicada (*CONSTRAINT section = source*) consiste se a informação de entrada correspondente à linha selecionada pelo usuário (*extended_line_n*) pertencente ao programa fonte original (e não a uma linha de *breakpoint*). A segunda restrição aplicada (*CONSTRAINT Line*) consiste a viabilidade da execução do serviço (se a linha selecionada é executável). A aplicação destas duas restrições precede as solicitações de inserção do *breakpoint* no programa em execução e a atualização das regiões de interface *BreakArea* e *SourceRegion*.

```
CLASS SourceRegion

SERVICE InsertBreak (extended_line_n)
    + section, source_line .SourceLine (extended_line_n)
    CONSTRAINT section = source
        -.InsertBreak(selected/source_line)

SERVICE InsertBreak (source_line)
    + Line SourceSet.Line (source_line)
    CONSTRAINT Line
        REQUIREMENT BreakLine_MustBe_Executable
        - BreakSet.InsertUserBreak (source_line)
        - BreakArea.InsertBreak (source_line)
        - .InsertBreak (source_line)
```

Na ausência de situações onde se aplicam as heurísticas de precedência, os serviços poderiam ser executados em paralelo. O operador de paralelismo é útil principalmente para multiprocessamento, que não faz parte do enfoque deste trabalho.

As classes de objeto de interface são responsáveis pelas seguintes ações de controle:

- transformação de informação de entrada para informação reconhecida pelo núcleo do sistema;
- transformação de informação do núcleo do sistema para informação de saída, reconhecida pelo usuário;
- caracterização do serviço solicitado pelo usuário, quando o evento de solicitação não é suficiente para identificar o serviço desejado, que é dependente do contexto;
- solicitação de execução do serviço a uma classe do núcleo.

Para a execução destes tipos de controle, as classes de objetos de interface necessitam eventualmente solicitar informações das classes do núcleo do sistema, o que não implica em operações de controle, mas simplesmente de leitura.

A solicitação de execução de serviço deve restringir-se a uma única classe do núcleo, encarregada de controlar a execução do serviço ou simplesmente executá-lo. Se a classe de interface precisa controlar várias classes do núcleo para a execução do sistema, deve ser criada uma classe adicional no núcleo que assuma este procedimento de sequenciar a execução dos serviços nas demais classes envolvidas.

No entanto, somente o controle de saída de informação na interface não justifica a criação de uma classe de controle. Se existe uma classe de controle cujo único propósito é receber solicitações de serviço da interface, somente repassando a solicitação dos serviços para as classes do núcleo, as solicitações de saída (eventualmente durante a execução de um serviço, várias regiões da interface podem ser envolvidas para apresentar informações de saída) devem ser efetuadas pela classe de interface que recebeu o evento associado ao serviço.

As regiões da interface são autônomas e devem prover serviços de saída de informação com uma interface desacoplada com a recepção de eventos de tal forma que o protocolo de solicitação seja idêntico, independentemente do fato da solicitação provir de uma classe de controle ou de uma classe de interface.

O serviço *InsertBreak* da classe *SourceRegion*, utilizado para ilustrar as heurísticas de seqüência de serviços, também é adequado para ilustrar dois aspectos relativos ao controle da saída na interface:

- o controle de saída de informação (na região de interface *BreakArea*) é feito através de uma classe de interface (*SourceRegion*), uma vez que a solicitação de inserção do *breakpoint* é somente repassada pela classe de interface (à classe *BreakSet*) o que não justifica a intermediação de uma classe de controle;
- e a autonomia dos serviços que provêem saídas na interface é caracterizada através da separação do serviço *InsertBreak* em dois serviços, desacoplando a consistência da informação de entrada (se linha pertence ao programa fonte) do encaminhamento do tratamento do comando, de forma que possa ser chamado por outras regiões da interface ou por classes de controle.

A Figura 27 ilustra os tipos de serviços associados às classes de interface na Especificação Unificada.

As classes de interface, mapeadas a partir das regiões das janelas da interface com os usuários, devem restringir o acesso às suas classes internas (em geral, criadas pela necessidade de armazenamento de informação de conversão entre meio externo e núcleo do sistema) às solicitações de saída de informação.

Entre classes de objetos, derivadas dos conceitos, pode-se identificar relacionamentos de agregação: uma classe de objetos pode agregar um ou mais conjuntos de classes de objetos.

Quando isto ocorre, a classe agregadora exerce controle sobre as classes agregadas, respondendo pela criação e remoção dos objetos das mesmas. Este tipo de controle deriva naturalmente da relação de agregação.

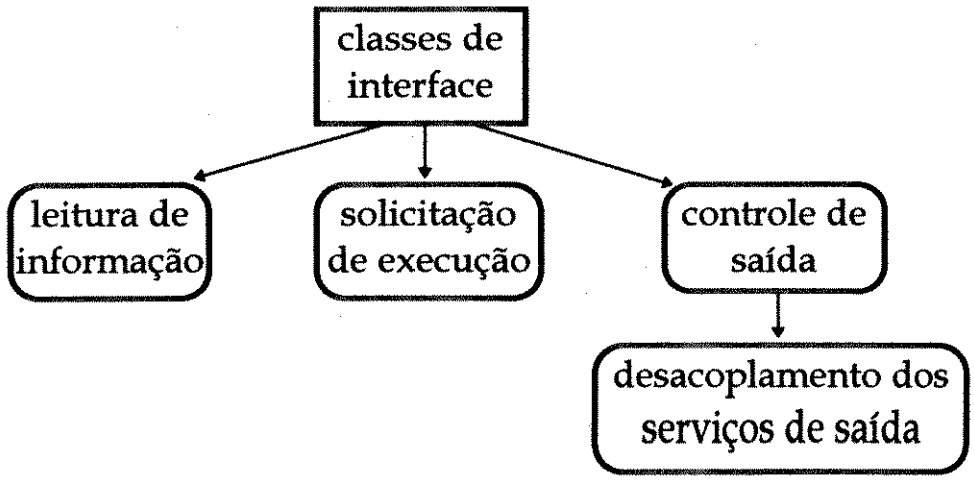

Figura 27: Tipos de Serviços associados às Classes de Interface

O acesso às classes agregadas, que não tem impacto nas mesmas mas na forma de solicitação do serviço e na interface provida pelas classes agregadoras, envolve o seguinte compromisso:

- a classe agregadora fornece o acesso à classe agregada: o que simplifica sua interface, mas aumenta a visibidade entre as classes;
- ou a classe agregadora fornece serviços que simplesmente repassam os serviços da classe agregada: o que onera sua interface (pela replicação das chamadas dos serviços das classes agregadas), mas diminui a visibilidade entre usuários e prestadores de serviços.

O serviço *SelectedTask* (solicitação da *task* selecionada) da classe *TaskSet* ilustra o fornecimento de acesso `a classe agregada (*Task*) pela classe agregadora (*TaskSet*).

SERVICE TaskSet

SERVICE SelectedTask : Task
+ Task active_tasks/SET [selected/task_id]

O serviço *BreakStop* (ativado pelo evento associado a parada do programa em execução em um *breakpoint*) e suas definições (*IfNoTask, IfStopped*) ilustram o repasse de serviço à classe agregada (*Task.BreakStop*), justificado pela atualização de estado da classe agregadora caracterizado pela criação de uma *task* na definição *IfNoTask* (*Task (task_id)*) e pelas atualizações de informação (*current/task_id = task_id, selected/task_id = task_id*) na definição *IfStopped.*

CLASS TaskSet

SERVICE BreakStop (task_id, cpu)
+ Task .IfNoTask

```
   - Task.BreakStop (cpu)
   - .IfStopped (task_id)

DEF IfNoTask : Task
   + Task active_tasks/SET [task_id]
   CONSTRAINT #Task
      REQUIREMENT NewTasks_CanBeActivatedOn_StopEvents
      * Task (task_id)

DEF IfStopped (task_id)
   + task/status .TaskStatus (task_id)
   CONSTRAINT STATE (Task) task/status = stopped
      * current/task_id = task_id
      * selected/task_id = task_id
         REQUIREMENT SelectedTask_CorrespondTo_CurrentTask_OnProgramStop
```

O mero repasse de serviços às classes agregadas pelas classes agregadoras deve ser evitado, uma vez que pode causar uma proliferação de vários serviços de repasse que podem ser substituídos por um único serviço de fornecimento de acesso. Algumas vezes, no entanto, a existência do repasse de controle é justificada pela necessidade complementar de alteração de estado da classe agregadora. Complementarmente, classes de objetos agregadas devem evitar solicitar informações à classes agregadoras. A Figura 28 ilustra o compromisso entre fornecer acesso e repassar serviços às classes agregadas.

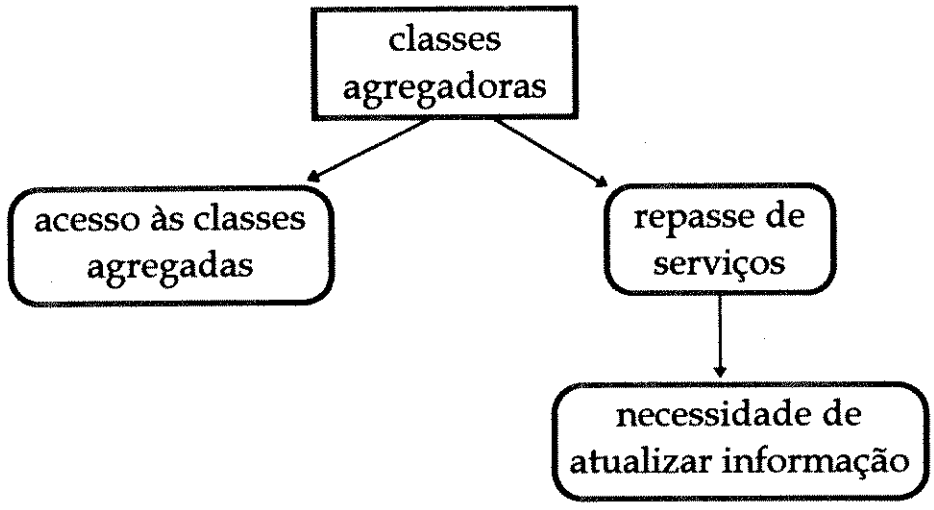

Figura 28: Compromisso entre Fornecer Acesso ou Repassar Serviços às Classes Agregadas

O agrupamento de classes em camadas de serviços é orientado em função de transições de níveis de abstração. Classes de níveis de abstração distintos devem pertencer a camadas de serviços distintas. A generalização dos serviços prestados por uma determinada camada é fundamental na concepção de *frameworks* de reuso.

Os serviços devem ser decompostos buscando maximizar delegação e a geração de serviços reusáveis. A decomposição dos serviços do núcleo do sistema (referenciados pelos serviços decompostos inicialmente em função da caracterização de seqüências de eventos na interface) baseia-se no princípio de delegar a sua execução a serviços mais primitivos.

A reusabilidade de serviços é função das seguintes condições:

- geração de serviços primitivos a partir dos quais são compostos os demais serviços;
- fatoração e parametrização de serviços.

Se durante o processo de decomposição de serviços existe a classe de objeto que mantém a informação a ser manipulada mas o serviço desejado não está disponível, este deverá ser criado. Se não corresponder a um serviço primitivo, será especificado a partir da utilização dos serviços existentes. A inclusão de novas classes de objetos ou serviços pode levar a um rearranjamento na hierarquia de classes ou na fatoração de serviços.

5.4 Conclusões

Muitas metodologias de análise e projeto orientadas a objetos enfatizam os aspectos de representação mas, apresentam um baixo poder heurístico no que se refere à capacidade de guiar a decomposição do sistema de software (vide seção 3.4: Análise das Metodologias Descritas segundo as Premissas Estabelecidas). Metodologias baseadas no comportamento dinâmico dos objetos, dentre as quais a metodologia OOSE [Jacobson92] é a mais bem estabelecida, apresentam um grande poder heurístico na decomposição da parte reativa de sistemas de software.

Na metodologia Especificação Unificada heurísticas de alocação de controle são apresentadas com vistas à cobertura da parte não reativa dos sistemas de software, coberta apenas superficialmente pelas metodologias existentes.

6. Recuperação de Visões da Especificação Unificada

Inicialmente discute-se o papel fundamental do mecanismo de recuperação de visões no suporte às atividades decorrentes da utilização da metodologia Especificação Unificada no desenvolvimento e na manutencão corretiva ou evolutiva de sistemas de software.

Recuperação de visões associadas aos requisitos são ilustradas com a recuperação de conceitos e de requisitos adicionais. Recuperações de visões, correspondentes ao modelamento dinâmico de sistemas de software, são ilustradas em diferentes níveis de abstração.

6.1 O Papel do Mecanismo de Recuperação de Visões

A utilização da metodologia Especificação Unificada é viável somente em função do suporte oferecido pelo mecanismo de recuperação de visões. Este mecanismo deve ser utilizado durante o desenvolvimento do sistema de software e após o seu desenvolvimento, em função da necessidade de avaliação e de operacionalização de ações de manutenção corretiva e evolutiva.

O mecanismo de recuperação de visões pode auxiliar o entendimento da parte especificada do sistema durante o desenvolvimento (quando o sistema está especificado parcialmente) para todas as atividades previstas na metodologia para o desenvolvimento de sistemas software, tais como:

- definição de conceitos, serviços de sistema e requisitos adicionais;
- definição das regiões da interfaces e dos seus eventos associados;
- mapeamento de regiões da interface e dos eventos, guiado pelos requisitos adicionais, nas classes de interface e seus principais serviços;
- mapeamento de conceitos e serviços de sistema, guiado pelos requisitos adicionais, nas classes e serviços do núcleo do sistema;
- adição de classes auxiliares em função das necessidades de armazenamento de informação;
- adição de classes para encapsular conceitos do ambiente computacional;
- adição de atributos ou de relacionamentos das classes para atender compromissos de eficiência.

Durante a avaliação ou operacionalização de ações de manutenção corretiva e evolutiva, o mecanismo de recuperação de visões é fundamental no suporte às seguintes atividades:

- entendimento dos artefatos (classes, serviços, conceitos, regiões de interface, atributos, eventos e estados) que compõem a especificação do sistema software, do relacionamento com os demais artefatos e do seu vínculo com os requisitos do sistema, a partir da recuperação de visões em vários níveis de abstração;
- avaliação das necessidades de alterações decorrentes de manutenção corretiva ou evolutiva;

- avaliação da introdução das alterações no sistema sob o ponto de vista do atendimento aos requisitos existentes e da restrição da propagação de alterações.

A Especificação Unificada incorpora várias construções que viabilizam a recuperação dos níveis mais altos de abstração da especificação a partir do mecanismo de extração de visões: vinculação de requisitos nas condições de seleções e de iterações, separação de tipos de serviços, e operadores de composição (utilizados na especificação dos serviços em todos os níveis de abstração). A Especificação Unificada suporta o rastreamento dos requisitos e a recuperação de níveis distintos de abstração a partir da representação do sistema como um conjunto de classes de objetos em cooperação.

A Especificação Unificada suporta a extração de visões de forma seletiva (automática ou interativa), mostrando ou escondendo a decomposição de um serviço (nos serviços que o compõem) até obter o detalhamento de interesse. A recuperação e, principalmente, o rastreamento dos requisitos adicionais é possível devido ao seu ancoramento explícito na decomposição dos serviços.

A Especificação Unificada é construída a partir de refinamentos sucessivos dos requisitos mas apenas o nível detalhado de especificação é mantido. Para permitir a recuperação dos níveis mais altos de abstração, a seguinte informação complementar é mantida na especificação:

- lista dos nomes dos conceitos;
- *layouts* de interface;
- mapeamento dos conceitos nas classes de objetos adicionais;
- estados e eventos que compõem cada um dos serviços prestados pelo sistema.

Além da recuperação de informação em nível mais alto de abstração, a Especificação Unificada fornece a base para a geração da implementação em linguagem orientada a objetos desde que providos parâmetros e defaults de mapeamento e informações adicionais (os tipos associados à informação utilizada).

A Figura 29 ilustra os mapeamentos possíveis a partir da Especificação Unificada: os diagramas obtidos a partir do mecanismo de recuperação de visões e a implementação em linguagem orientada a objetos.

6.2 Recuperação de Visões de Requisitos

Segundo a metodologia Especificação Unificada a especificação de requisitos abrange as descrições de conceitos, de serviços de sistema, de requisitos adicionais e de interface. A descrição de interface permanece na forma de única representação da Especificação Unificada, não necessitando portanto, de recuperação. A recuperação de visões requisitos é ilustrada com a recuperação de conceitos e de requisitos adicionais.

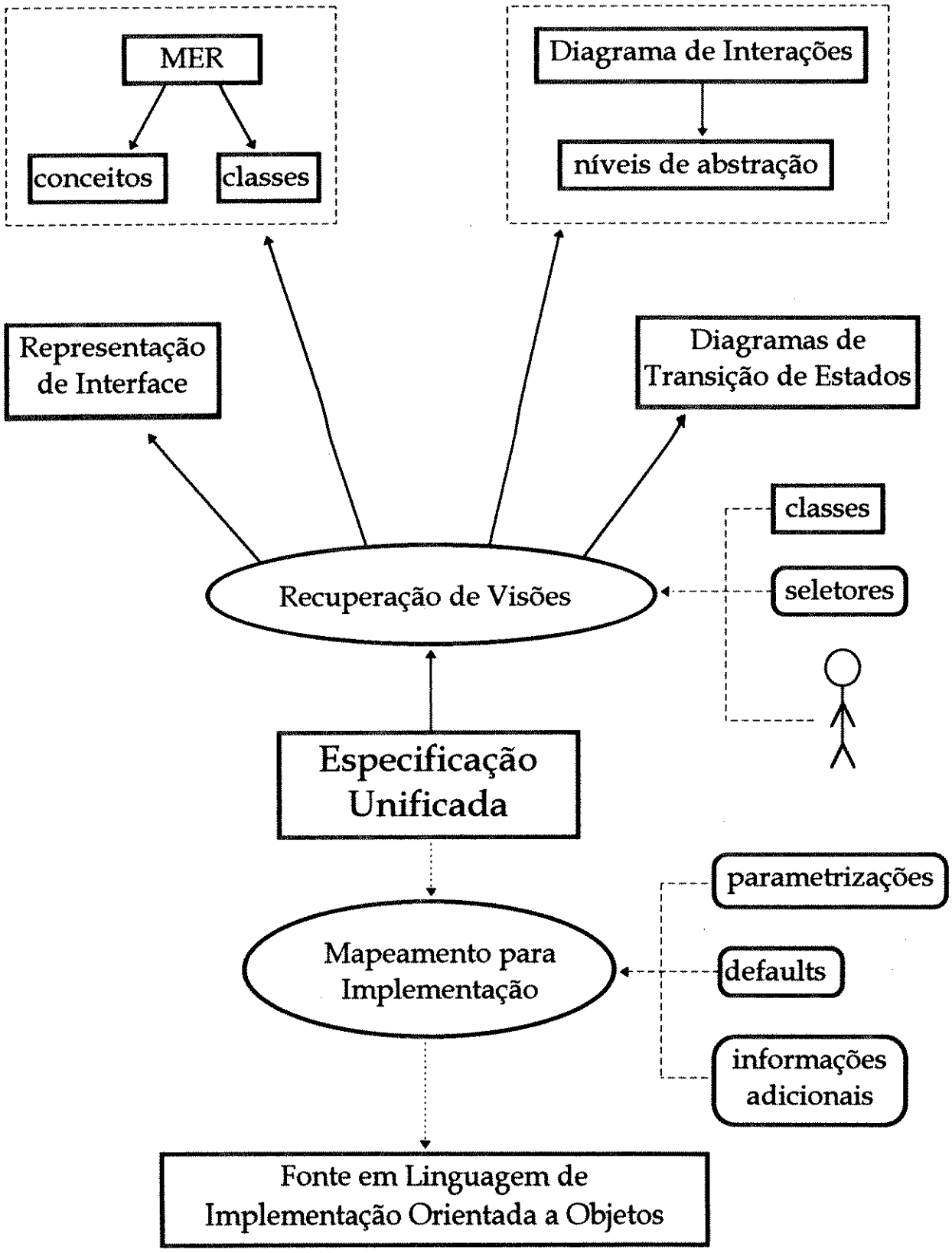

Figura 29: Mapeamentos a partir da Especificação Unificada

A recuperação dos conceitos é feita a partir das classes de objetos e das declarações de mapeamento que justificam a existência de cada classe como originária:

- dos conceitos;
- dos *layouts* de interface;
- da necessidade de armazenamento de informação não abrangida pelos conceitos (que justifica as classes agregadas pelas classes de interface);
- dos requisitos de projeto (incluindo atributos adicionais); e
- da necessidade de classes de controle.

Na extração de conceitos a partir da Especificação Unificada podem ocorrer duas situações:

- recuperação de conceito com correspondência direta com a classe de objetos da especificação unificada;
- recuperação de conceito a partir do qual foram mapeadas classes de objetos.

A primeira situação é ilustrada com a recuperação do conceito *Task*, cujos relacionamentos provêm somente da classe de objetos *Task*.

As declarações relevantes na especificação unificada são apresentadas abaixo.

```
CONCEPT Program, Address, Source, Line, Block, Symbol, Break, Task, Stack, Frame

CLASS Task
%break_type = [user_break, task_break, another_task_break]
%frame_reference = [same_frame, outer_frame, inner_frame, no_symbolic_inner_frame]
%range_reference = [inner_range, outer_range]
%status = [stopped, conting, dragging, nexting, nexting_inner, stepping, stepping_inner,
          step_header]
.is_associated_to : Stack
.is_defined_by : process/Block
.can_be_associated_to : artificial_break/Address
.is_identified_by : task_id
.status : status
.debug_phase
.referred/Frame
.last_stop/CPU
```

A recuperação correspondente é mostrada abaixo. Observe-se que os atributos que compõem o conceito correspondem aos atributos da classe que são precedidos de nome de relacionamento. A utilização de nomes de relacionamentos além de permitir a recuperação do atributo associado, aumenta a legibilidade das declarações de atributos e possibilita a geração de uma documentação sobre as classes, baseada na linguagem natural restrita utilizada para definir os requisitos do sistema.

```
CONCEPT Task
.is_associated_to : Stack
.is_defined_by : process/Block
.can_be_associated_to : artificial_break/Address
.is_identified_by : task_id
```

.status : status

A segunda situação é ilustrada com a recuperação do conceito *Program* das classes mapeadas a partir do conceito. As declarações relevantes na especificação unificada são mostradas abaixo.

CONCEPT Program, Address, Source, Line, Block, Symbol, Break, Task, Stack, Frame

MAPPING Program : Program, BreakSet, TaskSet, SourceSet, AddressSet

CLASS Program
.outermost_process : Block
%status = [loaded, starting, running, stooping, stopped]
.debug_phase
.status : status

CLASS BreakSet
.is_composed_by : user_breaks/SET (source_line)
.is_composed_by : program_breaks/SET (Break : Address)

CLASS TaskSet
.is_composed_by : active_tasks/SET (Task : task_id)
.identifies_the : selected/task_id
.identifies_the : current/task_id
.debug_phase

CLASS SourceSet
.is_composed_by : sources/SET (Source : source_name)

CLASS AddressSet
.is_composed_by : addresses/SET(Address)

A recuperação correspondente é mostrada abaixo. Pode-se notar que a precedência do nome do relacionamento viabiliza a recuperação de um conceito a partir de uma única classe ou de várias classes.

CONCEPT Program
.outermost_process : Block
.is_composed_by : user_breaks/SET (source_line)
.is_composed_by : program_breaks/SET (Break)
.is_composed_by : active_tasks/SET (Task)
.identifies_the : selected/task_id
.identifies_the : current/task_id
.is_composed_by : sources/SET (Source)
.is_composed_by : addresses/SET(Address)

Requisitos adicionais, extraídos da especificação unificada, são mostrados abaixo:

REQUIREMENT BreakLine_MustBe_Executable
REQUIREMENT TaskArea_HighlightsSelectedTask
REQUIREMENT StackArea_DisplaysStack_SelectedTask
REQUIREMENT SourceRegion_Focus_CurrentLine

REQUIREMENT SourceRegion_ClearFocus_CurrentLine_NoSymbolic
REQUIREMENT StackArea_HighlightsSelectedFrame
REQUIREMENT MinimizeHostTarget_InformationFlow
REQUIREMENT VariableFrameDisplay_AreMaintained
REQUIREMENT NextLine_SelectedFrame_OverCalledInnerFrame
REQUIREMENT NextLine_CurrentFrame_IntoCalledInnerFrame
REQUIREMENT ExecuteToHere_PassOver_UserBreaks
REQUIREMENT UserBreakStop_InterruptsAny_NextLineCommand

Com base na informação dos nomes de relacionamentos é possível obter também, diferentes visões correspondentes a modelos de entidade e relacionamentos. A obtenção dessas visões pode ser feita seletivamente, de forma automática ou iterativa, possibilitando a visualização dos artefatos de projeto segundo diferentes padrões de recuperação:

- diferentes níveis de abstração: conceitos e classes de projeto;
- diferentes categorias: classes de interface, classes mapeadas a partir dos conceitos ou classes de controle;
- diferentes camadas de serviços.

6.3 Recuperação de Visões de Modelamento Dinâmico

A recuperação de diagramas de interação (entre objetos na execução de um serviço) e de diagramas de transição de estados (de um dado objeto) é possível em vários níveis de abstração, devido às seguintes construções:

- distinção entre solicitação de serviço, solicitação de leitura de informação (normalmente associada à preparação de argumentos para solicitação de um serviço) e solicitação de armazenamento ou de descarte de informação;
- definições que agrupam subseqüências de serviços, com a finalidade de caracterizar diferentes níveis de abstração;
- informações de estados ou eventos;
- vinculação de requisitos;
- principais classes de interface;
- classes de núcleo relevantes (a relevância é definida através da iteração seletiva).

As seções seguintes ilustram recuperação do diagrama de transição de estados a partir de de um serviço descrito no nível de abstração da Especificação Unificada.

6.3.1 Descrição do Serviço Utilizado como Exemplo

O serviço *NextLineIntoCall* foi escolhido como exemplo para ilustrar o nível de abstração da Especificação Unificada e a extração de visões de níveis crescentes de abstração. Este serviço é

um dos mais complexos do sistema e ilustra grande parte das construções utilizadas na linguagem de representação. A seguir são mostradas as descrições do referido serviço (alguns dos serviços referenciados não são descritos, para não estender desnecessariamente o exemplo apresentado, uma vez que não são relevantes para o seu entendimento) no nível de abstração da especificação unificada.

Os diagramas de transição de estados e de interações entre objetos, excluindo serviços de leitura e descarte ou armazenamento de informação, são mostrados respectivamente nas Figuras 30 e 31. O diagrama de interações de objetos foi acrescido de informações adicionais de estado e de condições de seleção para facilitar a sua associação com o exemplo apresentado. Pelo mesmo motivo, alguns eventos do diagramas de transição de estados foram substituídos por condições de seleção (que correspondem à informação mais detalhada).

Devido ao número de classes de objetos distintas envolvidas no diagrama da Figura 31 (nove ao todo mais a classe hipotética *User*) adotou-se a estratégia de separá-las em duas partes. A primeira página da Figura 31 (que é ilustrada em quatro páginas) representa as seis primeiras classes, a partir da interface. As três páginas seguintes duplicam as duas últimas classes da primeira página (*TaskSet* e *Task*), com o objetivo de estabelecer um elo entre a primeira página e cada uma das três últimas páginas, e acrescentam as quatro classes restantes. A convenção "..." é utilizada para designar uma seqüência de mensagens correspondente a um serviço com mesmo nome descrito anteriormente.

Para esclarecer a informação dos diagramas de transição de estados e de interações entre objetos, a seqüência dos estados, eventos e principais serviços associados à execução do comando Step é descrita a seguir.

O usuário solicita o comando *Step* através da região de interface *ProgramMenu*, cuja solicitação se propaga (com o nome de NextLineIntoCall) através das classes *Program* (cujos estados dependem de estados da classe *Task*) e *TaskSet* até chegar à classe executora *Task* que comanda na máquina alvo (através da classe *TargetProgram*) a execução do trecho de código correspondente à linha fonte corrente e passa do estado *stopped* para *stepping*.

Dois eventos distintos podem ocorrer no estado *stepping*, uma vez que o monitor da máquina alvo (que representa a parte primitiva do depurador residente na máquina alvo) pode informar ao depurador (através de um evento gerado na *ProgramWindow*) a ocorrência de um *breakpoint* (evento *BreakStop*) ou a saída do trecho de código demarcado para execução (evento *OutStepRange*). A propagação dos eventos que chegam à interface ocorre de forma idêntica à descrita para o comando Step e será considerada implícita no restante da descrição.

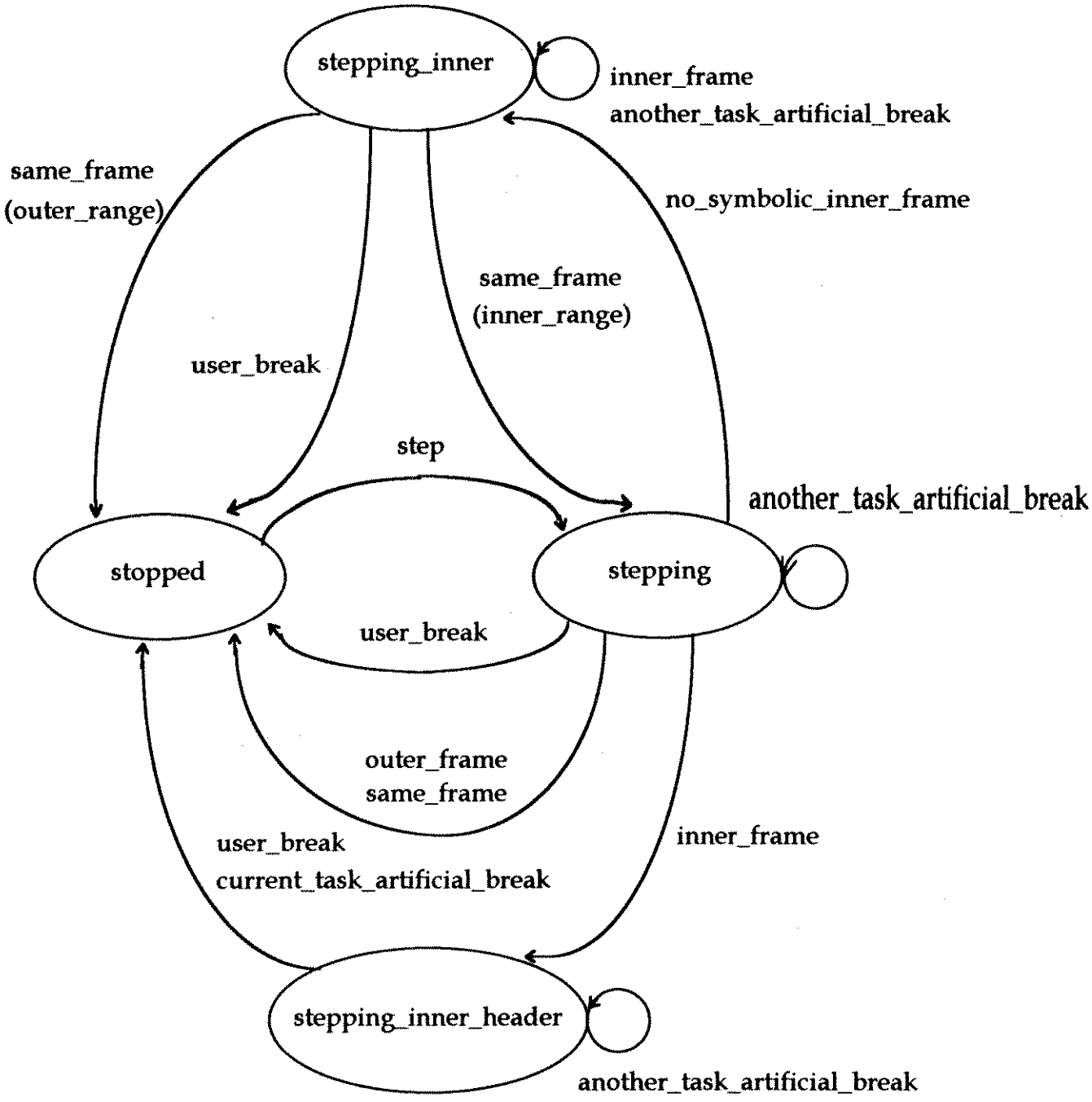

Figura 30: Diagrama de Transição de Estados do Serviço NextLineIntoCall

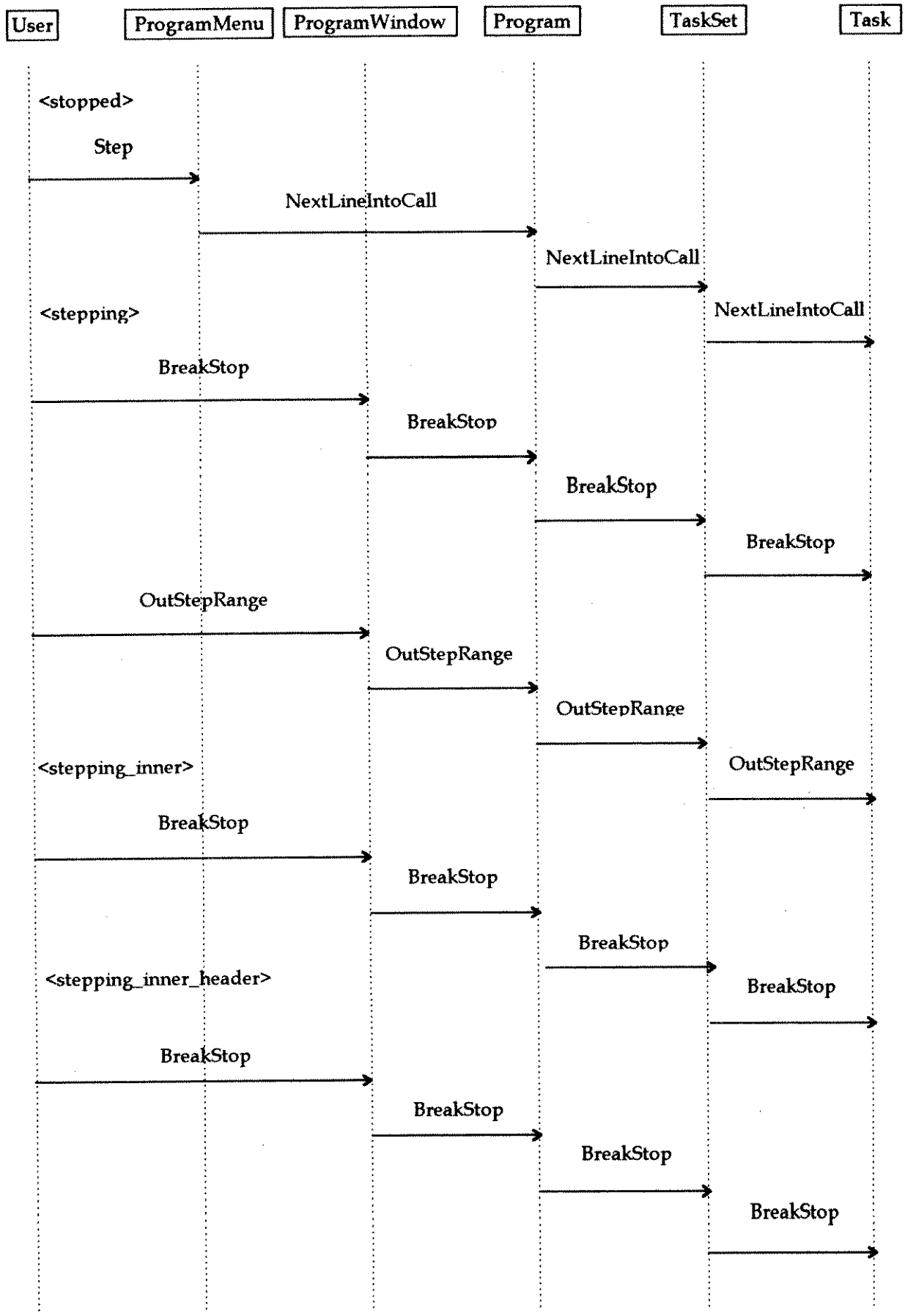

Figura 31: Diagrama de Interações do Serviço NextLineIntoCall

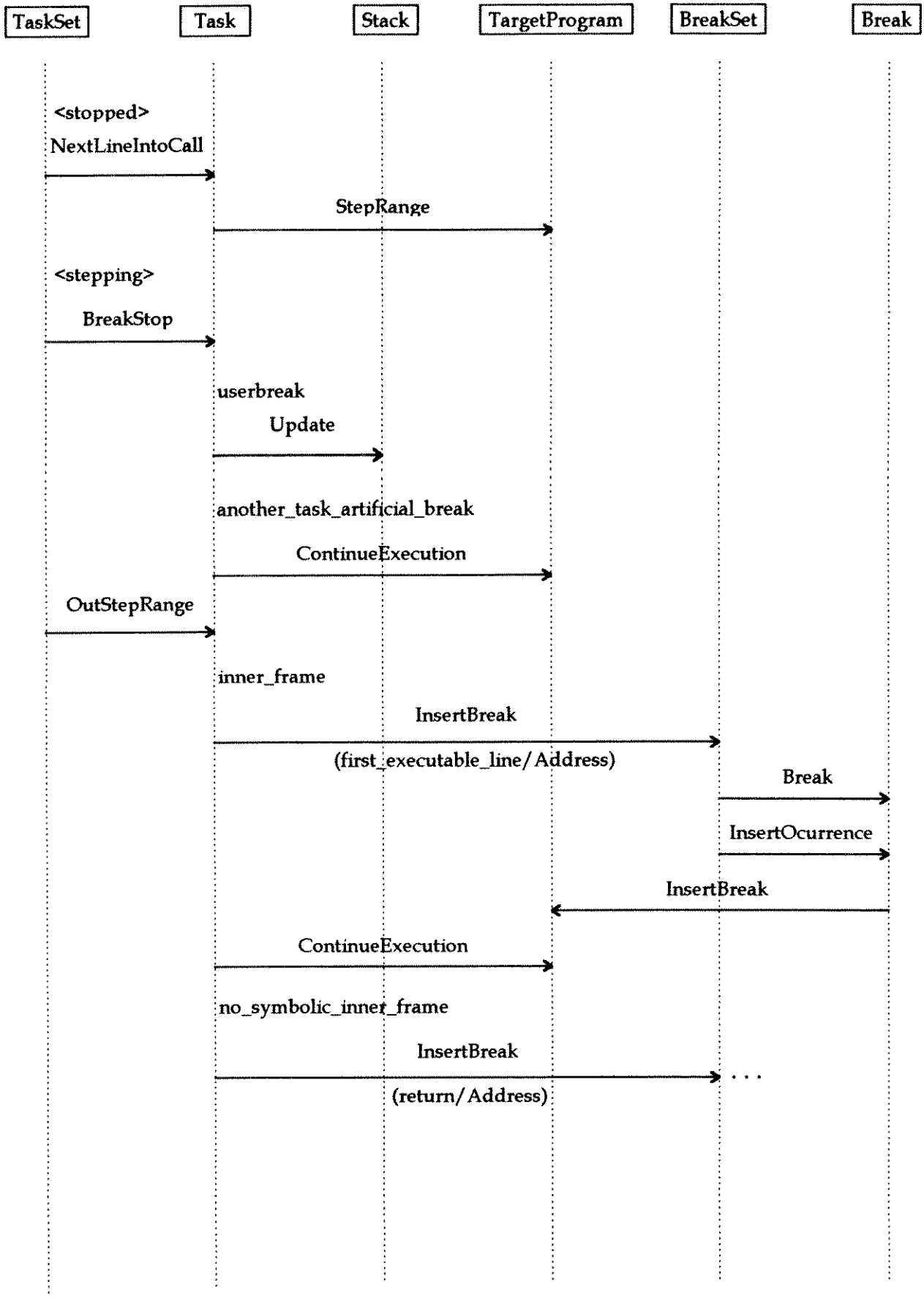

Figura 31: Diagrama de Interações do Serviço NextLineIntoCall (Continuação)

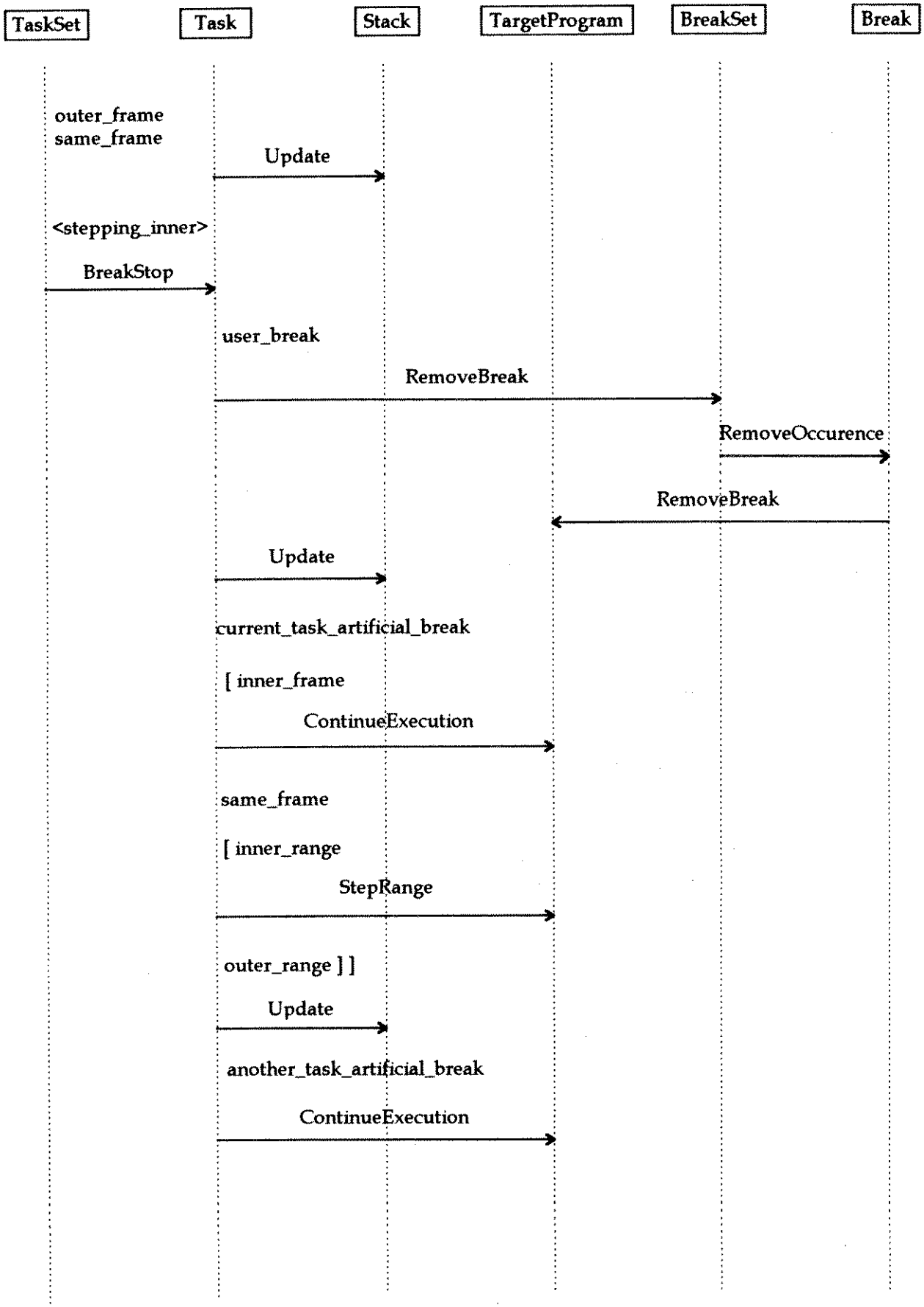

Figura 31: Diagrama de Interações do Serviço NextLineIntoCall (Continuação)

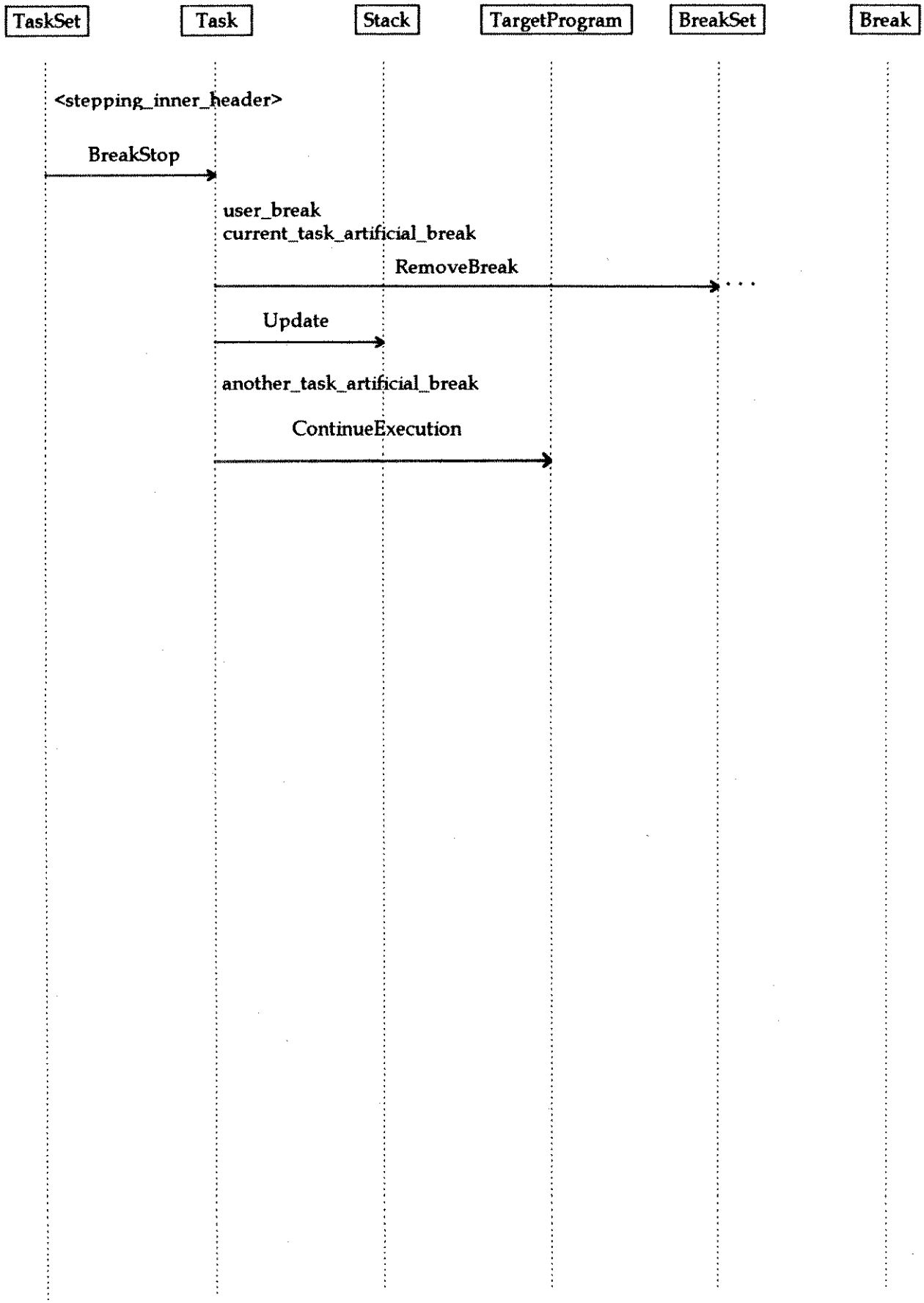

Figura 31: Diagrama de Interações do Serviço NextLineIntoCall (Continuação)

A recepção do evento *BreakStop* no estado *stepping* pode estar associada a duas situações distintas:

- ocorrência de *breakpoint* de usuário (que ocasiona a interrupção da execução do comando *Step*): a classe *Task* comanda a atualização do estado da pilha de execução do programa através da classe *Stack* e volta para o estado *stopped;*
- ocorrência de *breakpoint* artificial proveniente de comando de controle (*step, next* ou *execute_to_here*) associado a outra *task*: a classe *Task* comanda a continuação da execução do programa (através da classe *TargetProgram*).

A recepção do evento *OutStepRange* no estado *stepping* pode estar associada a três situações distintas:

- execução de instrução correspondente a *stack frame* interno (*procedure* ou processo) com representação simbólica: a classe *Task* comanda (através da classe *BreakSet* que comanda a classe *Break* que por sua vez comanda a classe *TargetProgram*) a inserção de um *breakpoint* artificial cujo endereço corresponde à primeira linha executável do *stack frame,* solicita continuação da execução (através da classe *TargetProgram*) e vai para o estado *stepping_inner_header;*
- execução de instrução correspondente a *stack frame* interno sem representação simbólica: a classe *Task* comanda a mesma seqüência de serviços com a diferença de que o *breakpoint* corresponde ao endereço de retorno do *stack frame* (uma vez que não há como finalizar a execução do comando *Step* em uma linha sem representação simbólica) e o estado destino é *stepping_inner;*
- execução de instrução correspondente a *stack frame* externo ou ao mesmo *stack frame*: o comando *Step* é considerado concluído e a classe *Task* comanda a mesma seqüência de serviços equivalente à ocorrência de *breakpoint* do usuário.

A recepção do evento *BreakStop* no estado *stepping_inner* pode estar associada a várias situações distintas:

- ocorrência de *breakpoint* de usuário: a classe *Task* comanda a remoção do *breakpoint* artificial (de forma semelhante a sua inserção) e a seguir comanda a mesma seqüência de serviços equivalente à ocorrência de *breakpoint* do usuário no estado *stepping;*
- ocorrência de *breakpoint* artificial pode estar associada as seguintes situações:
  - em *stack frame* interno (o que indica chamada recursiva a ser continuada): a classe Task comanda a continuação da execução;
  - no mesmo *stack frame* podem estar associadas as seguintes situações:
    - parada dentro do trecho de execução delimitado: a classe *Task* comanda a execução do trecho de código correspondente à linha fonte corrente do *stack frame* interno e passa para o estado *stepping;*

- parada fora do trecho de execução delimitado: o comando *Step* é considerado finalizado (idem *breakpoint* de usuário no estado *stepping*);

- ocorrência de *breakpoint* artificial associado a outra *task*: idem à mesma situação no estado *stepping*.

A recepção do evento *BreakStop* no estado *stepping_inner_header* pode estar associada a duas situações distintas:

- ocorrência de *breakpoint* de usuário ou de *breakpoint* artificial: a classe *Task* comanda a remoção do *breakpoint* artificial e em seguida comanda seqüência correspondente a *breakpoint* de usuário no estado *stepping*;
- ocorrência de *breakpoint* artificial associado a outra *task*: idem à mesma situação no estado *stepping*.

6.3.2 Descrição do Exemplo no Nível de Abstração da Especificação Unificada

SYSTEM CHILL_Symbolic_Debbuger

STATE Task INTERNAL Program

SERVICE NextLineIntoCall
STATE stopped
EVENT ProgramMenu.Step
STATE stepping (running)
EVENT ProgramWindow.BreakStop
EVENT ProgramWindow.OutStepRange
STATE stepping_inner (running)
EVENT ProgramWindow.BreakStop
STATE stepping_header (running)
EVENT ProgramWindow.BreakStop

STATE Task INTERNAL Program significa que os estados da *Task* são subestados de alguns dos estados de *Program*. Em conseqüência, *STATE stepping (running)* indica que o estado *stepping* (da *Task*) é um subestado do estado *running* (de *Program*). *STATE stopped* corresponde à uma notação simplificada para *STATE stopped (stopped)*.

CLASS ProgramMenu

EVENT Step
EVENT = User.MouseSelect (Step)
- .Step

SERVICE Step
- Program.NextLineIntoCall
- .EnableStop

```
DEF EnableStop
   - .DisableAll
   - .Enable (stop)


CLASS ProgramWindow

EVENT BreakStop
   EVENT = task_id, cpu Target.BreakStop
   - .BreakStop  (task_id, cpu)

EVENT OutStepRange
   EVENT = task_id, cpu Target.OutStepRange
   - .OutStepRange  (task_id, cpu)

SERVICE BreakStop  (task_id, cpu)
   - Program.BreakStop (task_id, cpu)
   - .IfStopped

SERVICE OutStepRange  (task_id, cpu)
   - Program.OutStepRange (task_id, cpu)
   - .IfStopped

DEF IfStopped
   + program/status Program.Status
   CONSTRAINT STATE (Program) program/status = stopped
      .Stopped

DEF Stopped
   REQUIREMENT ProgramStop_UpdatesUserInterface
   - .UpdateStateRegion
   + source_line .CurrentLine_SelectedFrame_SelectedTask
   - SourceRegion.UpdateSourceFocus (source_line)
   - ProgramMenu.Enable ([cont, next, step, abort, detach])

DEF UpdateStateRegion
   - TaskArea.Update
   - StackArea.Update
   - SourceArea.Update

DEF CurrentLine_SelectedFrame_SelectedTask
   - Stack TaskSet.StackSelectedTask
   - Frame Stack.SelectedFrame
   - source_line Frame.CurrentSourceLine
```

ProgramMenu.Enable ([cont, next, step, abort, detach]), de *DEF Stopped*, utiliza uma tupla como argumento: uma simplicação correspondente a solicitações do mesmo serviço repetidas vezes para cada um dos argumentos da tupla.

```
CLASS Program
```

```
SERVICE NextLineIntoCall
   STATE stopped
      - TaskSet.NextLineIntoCall
      NEXTSTATE running

SERVICE BreakStop  (task_id, cpu)
   STATE running
      * debug_phase = + 1
      - TaskSet.BreakStop (task_id, cpu)
      - .IfStopped (task_id)

SERVICE OutStepRange  (task_id, cpu)
   STATE running
      * debug_phase = + 1
      - TaskSet.OutStepRange (task_id, cpu)
      - .IfStopped (task_id)

DEF IfStopped (task_id)
   REQUIREMENT StateProgram_IsFunctionOf_StateTask
   + task/status TaskSet.TaskStatus (task_id)
   CONSTRAINT STATE (Task) task/status = stopped
      - SourceSet.UpdateData
      NEXTSTATE stopped
   OTHER
      NEXTSTATE
```

A omissão do nome de estado após a palavra chave *NEXTSTATE* indica permanência no estado atual.

```
CLASS TaskSet

SERVICE NextLineIntoCall
   + Task active_tasks/SET [current/task_id]
   - Task.NextLineIntoCall

SERVICE BreakStop (task_id, cpu)
   + Task .IfNoTask
   - Task.BreakStop (cpu)
   - .IfStopped (task_id)

SERVICE OutStepRange (task_id, cpu)
   + Task .IfNoTask
   - Task.OutStepRange (cpu)
   - .IfStopped (task_id)

SERVICE TaskSet.TaskStatus (task_id) : task/status
   + Task active_tasks/SET [task_id]
   - task/status Task.Status

DEF IfNoTask : Task
   + Task active_tasks/SET [task_id]
   CONSTRAINT #Task
      REQUIREMENT NewTasks_CanBeActivatedOn_StopEvents
      * Task (task_id)
```

```
DEF IfStopped (task_id)
    + task/status .TaskStatus (task_id)
    CONSTRAINT STATE (Task) task/status = stopped
        * current/task_id = task_id
        * selected/task_id = task_id
            REQUIREMENT SelectedTask_CorrespondTo_CurrentTask_OnProgramStop


CLASS Task

SERVICE NextLineIntoCall
REQUIREMENT NextLine_CurrentFrame_IntoCalledInnerFrame
STATE stopped
    REQUIREMENT CurrentFrame_DefinesContextOf_StepCommand
    + current/Frame Stack.SelectedFrame
    * referred/Frame = current/Frame
    - .StepRange (Frame)
        REQUIREMENT CurrentLine_DefinesRangeExecutionOf_StepCommand
    NEXTSTATE stepping

SERVICE BreakStop (cpu)
STATE dragging
    * last_stop/CPU = CPU (cpu)
    + break_type .BreakType
    CONSTRAINT break_type
    = user_break
        REQUIREMENT ExecuteToHere_PassOver_UserBreaks
        - TargetProgram.ContinueExecution
        NEXTSTATE
    = current_task_artificial_break
        - .TargetStop_Reached
        NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
STATE nexting
    * last_stop/CPU = CPU (cpu)
    + break_type .BreakType
    CONSTRAINT break_type
    = user_break
        - .UserBreakStop
        NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
STATE nexting_inner
    * last_stop/CPU = CPU (cpu)
    + break_type .BreakType
    CONSTRAINT break_type
    = user_break
        - .UserBreak_WhileInnerFrame
        NEXTSTATE stopped
    = current_task_artificial_break
        + frame_reference .FrameReference (task_id, referred_frame, CPU)
        CONSTRAINT frame_reference
```

```
        = inner_frame
            - TargetProgram.ContinueExecution    ;Recursive_InnerFrame
            NEXTSTATE
        = same_frame
            + range_reference .RangeReference (task_id,CPU)
            CONSTRAINT range_reference
            = inner_range
                - .StepRange (referred/frame)  ; YetInto_StepRange
                NEXTSTATE nexting (running)
            = outer_range
                - .Update   ; NextLine_Reached
                NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
STATE stepping
    * last_stop/CPU = CPU (cpu)
    + break_type .BreakType
    CONSTRAINT break_type
    = user_break
        - .UserBreakStop
        NEXTSTATE stopped
    = another_task_artificial_break
        REQUIREMENT AnotherTask_ArtificialBreak_MustBePassedOver
        - TargetProgram.ContinueExecution
        NEXTSTATE
STATE stepping_inner
    ; NoSymbolic_InnerFrame
    * last_stop/CPU = CPU (cpu)
    + break_type .BreakType (task_id, cpu)
    CONSTRAINT break_type
    = user_break
        - .UserBreak_WhileInnerFrame
        NEXTSTATE stopped
    = current_task_artificial_break
        + frame_reference .FrameReference
        CONSTRAINT frame_reference
        = inner_frame
            REQUIREMENT RecursiveInnerFrame_MustBePassedOver
            - TargetProgram.ContinueExecution
            NEXTSTATE
        = same_frame
            + range_reference .RangeReference
            CONSTRAINT range_reference
            = inner_range
                ; YetInto_StepRange
                - .StepRange (referred/Frame)
                NEXTSTATE stepping
            = outer_range
                ; NextLine_Reached
                - .Update
                NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
```

```
    * last_stop/CPU = CPU (cpu)
    + break_type .BreakType
    CONSTRAINT break_type
    = user_break
        - .UserBreak_WhileInnerFrame
        NEXTSTATE stopped
    = current_task_artificial_break
        - .TargetStop_Reached
        NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE

SERVICE OutStepRange (cpu)
STATE nexting
    * last_stop/CPU = CPU (cpu)
    + frame_reference .FrameReference
    CONSTRAINT frame_reference
    = inner_frame, no_symbolic_ inner_frame
        - .ReturnFrom_InnerFrame
        NEXTSTATE nexting_inner
    = outer_frame, same_frame
        - .Update   ; NextLine_Reached
        NEXTSTATE stopped
STATE stepping
    * last_stop/CPU = CPU (cpu)
    + frame_reference .FrameReference
    CONSTRAINT frame_reference
    = inner_frame
        - .Reach_FirstExecutableLine_InnerFrame
        NEXTSTATE stepping_header
    = no_symbolic_ inner_frame
        REQUIREMENT NoSymbolic_InnerFrame_MustBePassesOver_OnStep
        - .ReturnFrom_InnerFrame
        NEXTSTATE stepping_inner
    = outer_frame, same_frame
        - .Update   ; NextLine_Reached
        NEXTSTATE stopped

SERVICE Update
    + program/debug_phase Program.DebugPhase
    CONSTRAINT program/debug_phase > debug_phase
        - Stack.Update (task_id)
        * debug_phase = program/debug_phase

DEF StepRange (Frame)
    + frame/Address Frame.Address
    + first_range/Address, last_range/Address .RangeAddress (Frame)
    - TargetProgram.StepRange (frame/Address, first_range/Address, last_range/Address)

DEF RangeAddress (Frame) : first_range/Address, last_range/Address
    + first_range/Address Frame.CurrentLineAddress
    + next_line/Address AddressSet.NextLineAddress (first_range/Address)

DEF BreakType : break_type
    + instruction_pointer/Address CPU.InstructionPointer
```

```
    + user_break BreakSet.UserBreak (instruction_pointer/Address)
    CONSTRAINT user_break
        % break_type = user_break
    OTHER  ; ArtificialBreak
            CONSTRAINT instruction_pointer/Address = artificial_break/Address
                ; ArtificialBreak_OfThisTask
                % break_type = task_break
            OTHER
                ; ArtificialBreak_OfAnotherTask
                % break_type = another_task_break

DEF TargetStop_Reached
    - BreakSet.RemoveBreak (artificial_break/Address)
    * #artificial_break/address
    - .Update

DEF UserBreakStop
    REQUIREMENT UserBreakStop_InterruptsAny_NextLineCommand
    - .Update

DEF UserBreak_WhileInnerFrame
    - BreakSet.RemoveBreak (artificial_break/Address)
    * #artificial_break/Address
    - .Update

DEF FrameReference : frame_reference
    + instruction_pointer/Address CPU.InstructionPointer
    + frame_pointer/Address CPU.FramePointer
    % frame_reference referred/Frame.FrameReference
      (instruction_pointer/Address, frame_pointer/Address)

DEF RangeReference : range_reference
    - first_range/Address, last_range/Address .RangeAddress (referred/Frame)
    CONSTRAINT instruction_pointer/Address < last_range/Address
        % range_reference = inner_range
    OTHER
        % range_reference = outer_range

DEF ReturnFrom_InnerFrame
    + return/Address .ReturnAddress
    - BreakSet.InsertBreak (return/Address)
    * artificial_break/Address = return/Address

DEF ReturnAddress : Address
    + stack_pointer/Address CPU.StackPointer
    + return/Address TargetProgram.Read (stack_pointer/Address)

DEF Reach_FirstExecutableLine_InnerFrame
    + first_executable_line/Address .FirstExecutable_LineAddress
    - BreakSet.InsertBreak (first_executable_line/Address)
    * artificial_break/Address = first_executable_line /Address

DEF FirstExecutableLineAddress : first_executable_line/Address
    + instruction_pointer/Address CPU.InstructionPointer
    + Block Line.Block (instruction_pointer/Address)
    + initial/Line Block.InitialLine
```

```
    + initial/Address Line.InitialAddress
    % first_executable_line/Address = initial/Address


CLASS BreakSet

SERVICE InsertBreak (Address)
    + program/status Program.Status
    CONSTRAINT program/status # loaded
        REQUIREMENT UserBreak_OnlyCanBeInsertedOnProgram_AfterProgramStarted
        + Break program_breaks/SET [Address]
        CONSTRAINT #Break
            * program_breaks/SET (Break) = + Break (Address)
        - Break.InsertOcurrence
        - .RemoveBreakData_WhenNoOcurrences (Break)    ; ExceptionHandler

SERVICE RemoveBreak (Address)
    + program/status Program.Status
    CONSTRAINT program/status # loaded
        + Break program_breaks/SET [Address]
        - Break.RemoveOcurrence
        - .RemoveBreakData_WhenNoOcurrences (Break)

DEF RemoveBreakData_WhenNoOcurrences (Break)
    + ocurrences_n Break.OcurrenceNumber
    CONSTRAINT #ocurrences_n
        * program_breaks/SET (Break) = - ~Break


CLASS Break

SERVICE Break (source_line)
    * #ocurrence_n

SERVICE InsertOcurrence
    CONSTRAINT ocurrence_n = 1
        ; FirstBreakInstance_InsertBreak_OnProgram
        - TargetProgram.InsertBreak (program_break/Address)
        * ocurrence_n = + 1

SERVICE RemoveOcurrence
    CONSTRAINT #ocurrence_n
        ; LastBreakInstanceRemoved_RemoveBreak_OnProgram
        - TargetProgram.RemoveBreak (program_break/Address)
        * ocurrence_n = - 1

SERVICE OcurrenceNumber : ocurrence_n
```

6.3.3 Primeira Visão Intermediária

O objetivo deste exemplo é mostrar a recuperação do diagrama de estados do serviço, mas para ilustrar a capacidade de recuperação iterativa seletiva, são apresentadas duas visões inter-

mediárias. A primeira visão intermediária recupera o serviço a nível de sistema (uma vez que o objetivo não é mostrar o diagrama de interações entre as classes de objetos) enfocando apenas a chamada dos serviços das classes de interface. A recuperação é orientada a partir das seguintes restrições:

- a interação entre as diversas classes (*ProgramMenu, ProgramWindow, Program, TaskSet* e *Task*) é aglutinada em um único serviço do sistema (pertencente a *SYSTEM*);
- os estados que não são referentes ao serviço em foco (*dragging, nexting e nexting_inner*) são retirados;
- somente as chamadas a serviços da classe de interface com o programa em execução na máquina alvo (*TargetProgram*) e das classes de interface com usuário (*ProgramMenu, ProgramWindow, SourceRegion*) são mantidas (bem como as definições que referenciam serviços de tais classes).

```
SYSTEM CHILL_Symbolic_Debbuger

SERVICE NextLineIntoCall
REQUIREMENT NextLine_CurrentFrame_IntoCalledInnerFrame
STATE stopped
EVENT = ProgramMenu.Step
    + frame .CurrentFrame_CurrentTask
    - .StepRange (frame)
    * referred/frame = frame
    - .EnableStop
    NEXTSTATE stepping (running)
STATE stepping (running)
EVENT
= task_id, cpu ProgramWindow.BreakStop
    + break_type .BreakType (task_id, cpu)
    CONSTRAINT break_type
    = user_break
        - .UserBreakStop (task_id)
        - .Stopped
        NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
= task_id, cpu ProgramWindow.OutStepRange
    + frame_reference
        .FrameReference (task_id, referred_frame, cpu)
    CONSTRAINT frame_reference
    = inner_frame
        - .Reach_FirstExecutableLine_InnerFrame (task_id, cpu)
        NEXTSTATE stepping_inner_header
    = no_symbolic_ inner_frame
        - .ReturnFrom_InnerFrame (task_id, cpu)
        NEXTSTATE stepping_inner
    = outer_frame, same_frame
        ; NextLine_Reached
        - .Update (task_id)
        - .Stopped
```

```
        NEXTSTATE stopped
STATE stepping_inner (running)
EVENT = task_id, cpu ProgramWindow. BreakStop
    + break_type .BreakType (task_id, cpu)
    CONSTRAINT break_type
    = user_break
        - .UserBreak_WhileInnerFrame (task_id)
        - .Stopped
        NEXTSTATE stopped
    = current_task_artificial_break
        + frame_reference .FrameReference (task_id, referred_frame, cpu)
        CONSTRAINT frame_reference
        = inner_frame
            - TargetProgram.ContinueExecution    ;Recursive_InnerFrame
            NEXTSTATE
        = same_frame
            + range_reference .RangeReference (task_id, cpu)
            CONSTRAINT range_reference
            = inner_range
                - .StepRange (referred/frame)  ; YetInto_StepRange
                NEXTSTATE stepping (running)
            = outer_range
                ; NextLine_Reached
                - .Update (task_id)
                - .Stopped
                NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
STATE stepping_header (running)
EVENT = task_id, cpu ProgramWindow. BreakStop
    + break_type .BreakType (task_id, cpu)
    CONSTRAINT break_type
    = user_break
        - .UserBreak_WhileInnerFrame (task_id)
        - .Stopped
        NEXTSTATE stopped
    = current_task_artificial_break
        - .TargetStop_Reached (task_id)
        - .Stopped
        NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE

DEF Stopped
    - .UpdateStateRegion
    + source_line .CurrentLine_SelectedFrame_SelectedTask
    - SourceRegion.UpdateSourceFocus (source_line)
    - ProgramMenu.Enable ([cont, next, step, abort, detach])

DEF UpdateStateRegion
    - TaskArea.Update
    - StackArea.Update
    - SourceArea.Update
```

DEF EnableStop
- *ProgramMenu.DisableAll*
- *ProgramMenu.Enable (stop)*

DEF UserBreakStop (task_id)
REQUIREMENT UserBreakStop_InterruptsAny_NextLineCommand
- *.Update (task_id)*

DEF Reach_FirstExecutableLine_InnerFrame (task_id, cpu)
+ *first_executable_line/address .FirstExecutable_LineAddress (cpu)*
- *TargetProgram.InsertBreak (first_executable_line/address)*
% *artificial_break/address = first_executable_line /address*
* *[task_id, artificial_break/address]*

DEF ReturnFrom_InnerFrame (task_id, cpu)
+ *return/address .ReturnAddress (cpu)*
- *TargetProgram.InsertBreak (return/address)*
% *artificial_break/address = return/address*
* *[task_id, artificial_break/address]*

DEF Update (task_id)
- *.UpdateTask (task_id)*
+ *source_line .CurrentLine_SelectedFrame_SelectedTask*
- *SourceRegion.UpdateSourceFocus (source_line)*
- *ProgramMenu.Enable ([cont, next, step, abort, detach])*

DEF UserBreak_WhileInnerFrame (task_id)
- *TargetProgram.RemoveBreak (artificial_break/address)*
* *#[task_id, artificial_break/address]*
- *.Update (task_id)*

DEF TargetStop_Reached (task_id)
- *TargetProgram.RemoveBreak (artificial_break/address)*
* *#[task_id, artificial_break/address]*
- *.Update (task_id)*

Os estados do serviço do sistema (*NextLineIntoCall*) provêm dos estados da classe *Task*, uma vez que ela é a classe para a qual ocorre o último repasse dos eventos gerados na interface (*step, break_stop* e *out_step_range*). Os estados da classe *Program* são atualizados em função da classe *Task*.

6.3.4 Segunda Visão Intermediária

A segunda visão intermediária é uma simplificação da primeira, a partir da aplicação das seguintes restrições:

- as referências às definições são expandidas;
- são retiradas as ações correspondentes à preparação de argumentos e de salvamento ou descarte de informação.

```
SYSTEM CHILL_Symbolic_Debbuger

SERVICE NextLineIntoCall
REQUIREMENT NextLine_CurrentFrame_IntoCalledInnerFrame
STATE stopped
EVENT = ProgramMenu.Step
    - TargetProgram.StepRange (frame/address, first_line/address, next_line/address)
    - ProgramMenu.DisableAll
    - ProgramMenu.Enable (stop)
    NEXTSTATE stepping (running)
STATE stepping (running)
EVENT
= task_id, cpu ProgramWindow.BreakStop
    CONSTRAINT break_type
    = user_break
        REQUIREMENT UserBreakStop_InterruptsAny_NextLineCommand
        - TaskArea.Update
        - StackArea.Update
        - SourceArea.Update
        - SourceRegion.UpdateSourceFocus (source_line)
        - ProgramMenu.Enable ([cont, next, step, abort, detach])
        NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
= task_id, cpu ProgramWindow.OutStepRange
    CONSTRAINT frame_reference
    = inner_frame
        ; Reach_FirstExecutableLine_InnerFrame
        - TargetProgram.InsertBreak (first_executable_line/address)
        NEXTSTATE stepping_inner_header
    = no_symbolic_ inner_frame
        ; ReturnFrom_InnerFrame
        - TargetProgram.InsertBreak (first_executable_line/address)
        NEXTSTATE stepping_inner
    = outer_frame, same_frame
        ; NextLine_Reached
        - TaskArea.Update
        - StackArea.Update
        - SourceArea.Update
        - SourceRegion.UpdateSourceFocus (source_line)
        - ProgramMenu.Enable ([cont, next, step, abort, detach])
        NEXTSTATE stopped
STATE stepping_inner (running)
EVENT = task_id, cpu ProgramWindow. BreakStop
    CONSTRAINT break_type
    = user_break
        ;UserBreak_WhileInnerFrame
        - TargetProgram.RemoveBreak (artificial_break/address)
        - TaskArea.Update
        - StackArea.Update
        - SourceArea.Update
        - SourceRegion.UpdateSourceFocus (source_line)
        - ProgramMenu.Enable ([cont, next, step, abort, detach])
```

```
        NEXTSTATE stopped
    = current_task_artificial_break
        CONSTRAINT frame_reference
        = inner_frame
            - TargetProgram.ContinueExecution    ;Recursive_InnerFrame
            NEXTSTATE
        = same_frame
            CONSTRAINT range_reference
            = inner_range
                ; YetInto_StepRange
                - TargetProgram.StepRange (frame/address, first_line/address, next_line/address)
                NEXTSTATE stepping (running)
            = outer_range
                ; NextLine_Reached
                - TaskArea.Update
                - StackArea.Update
                - SourceArea.Update
                - SourceRegion.UpdateSourceFocus (source_line)
                - ProgramMenu.Enable ([cont, next, step, abort, detach])
                NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
STATE stepping_header (running)
EVENT = task_id, cpu ProgramWindow. BreakStop
    CONSTRAINT break_type
    = user_break
        ; UserBreak_WhileInnerFrame
        - TargetProgram.RemoveBreak (artificial_break/address)
        - TaskArea.Update
        - StackArea.Update
        - SourceArea.Update
        - SourceRegion.UpdateSourceFocus (source_line)
        - ProgramMenu.Enable ([cont, next, step, abort, detach])
        NEXTSTATE stopped
    = current_task_artificial_break
        ; TargetStop_Reached
        - TargetProgram.RemoveBreak (artificial_break/address)
        - TaskArea.Update
        - StackArea.Update
        - SourceArea.Update
        - SourceRegion.UpdateSourceFocus (source_line)
        - ProgramMenu.Enable ([cont, next, step, abort, detach])
        NEXTSTATE stopped
    = another_task_artificial_break
        - TargetProgram.ContinueExecution
        NEXTSTATE
```

6.3.5 Visão Correspondente ao Diagrama de Transição de Estados

A última visão, obtida a partir da exclusão dos serviços de interface (só os eventos são mantidos), corresponde ao diagrama de estados do serviço. Os argumentos de retorno dos eventos foram eliminados por serem irrelevantes para a informação que se deseja mostrar.

```
SYSTEM CHILL_Symbolic_Debbuger

SERVICE NextLineIntoCall
REQUIREMENT NextLine_CurrentFrame_IntoCalledInnerFrame
STATE stopped
EVENT = ProgramMenu.Step
   NEXTSTATE stepping (running)
STATE stepping (running)
EVENT
= ProgramWindow.BreakStop
   CONSTRAINT break_type
   = user_break
      REQUIREMENT UserBreakStop_InterruptsAny_NextLineCommand
      NEXTSTATE stopped
   = another_task_artificial_break
      NEXTSTATE
= ProgramWindow.OutStepRange
   CONSTRAINT frame_reference
   = inner_frame
      ; Reach_FirstExecutableLine_InnerFrame
      NEXTSTATE stepping_inner_header
   = no_symbolic_ inner_frame
      ; ReturnFrom_InnerFrame
      NEXTSTATE stepping_inner
   = outer_frame, same_frame
      ; NextLine_Reached
      NEXTSTATE stopped
STATE stepping_inner (running)
EVENT = ProgramWindow. BreakStop
   CONSTRAINT break_type
   = user_break
      ;UserBreak_WhileInnerFrame
      NEXTSTATE stopped
   = current_task_artificial_break
      CONSTRAINT frame_reference
      = inner_frame
         ;Recursive_InnerFrame
         NEXTSTATE
      = same_frame
         CONSTRAINT range_reference
         = inner_range
            ; YetInto_StepRange
            NEXTSTATE stepping (running)
         = outer_range
            ; NextLine_Reached
            NEXTSTATE stopped
```

```
    = another_task_artificial_break
        NEXTSTATE
STATE stepping_header (running)
EVENT = ProgramWindow. BreakStop
    CONSTRAINT break_type
    = user_break
        ; UserBreak_WhileInnerFrame
        NEXTSTATE stopped
    = current_task_artificial_break
        ; TargetStop_Reached
        NEXTSTATE stopped
    = another_task_artificial_break
        NEXTSTATE
```

Embora o exemplo apresentado tenha ilustrado apenas a recuperação do diagrama de transição de estados, a recuperação do diagrama de interações de objetos pode ser obtida de forma similar. Neste caso, a recuperação pode ser orientada pela restrição a tipos de classes (por exemplo classes de interface), a classes de uma determinada camada de serviços, ou mesmo a algumas classes específicas do núcleo do sistema.

Pode-se observar que o diagrama de estados (ilustrado na Figura 30) necessita de informação complementar correspondente à ordem de ocorrência dos estados e eventos. Da mesma forma, o diagrama de interações (ilustrado na Figura 31) necessita de informação adicional para distinguir:

- informação de argumentos e de retorno dos serviços;
- eventos paralelos;
- seleção e iteração de serviços (eventualmente em vários níveis de profundidade).

Esta tendência tem sido observada em propostas de renomados metodologistas [Rumbaugh94b] [Embley92], com o objetivo de capturar aspectos de especificação adicionais. A recuperação seletiva (de forma automática ou iterativa) de tais diagramas, a partir da Especificação Unificada, constitui-se em uma proposta muito mais flexível, uma vez que o nível de abstração e o tipo de informação recuperada pode ser parametrizado ou selecionado iterativamente.

7. O Contexto da Especificação Unificada

Com o objetivo de situar a metodologia Especificação Unificada no contexto de outras propostas, são apresentadas:

- uma comparação com outras metodologias e com as premissas estabelecidas para avaliação das metodologias;
- a abrangência da metodologia;
- uma experiência de utilização.

7.1 Comparação com Outras Metodologias

A comparação dos conceitos propostos pela metodologia Especificação Unificada com os conceitos propostos pelas outras metodologias (resumidas neste trabalho) é organizada da seguinte forma:

- conceitos incorporados de outras metodologias;
- correspondência com conceitos de outras metodologias;
- contribuições da metodologia Especificação Unificada.

7.1.1 Conceitos Incorporados de Outras Metodologias

A metodologia Especificação Unificada corresponde ao amadurecimento de um trabalho cujo objetivo inicial era propor extensões à metodologia OOSE [Jacobson92]:

- uma representação mais formal para a especificação de requisitos;
- heurísticas adicionais para a decomposição da parte não reativa do sistema (uma vez que a decomposição da parte reativa através de casos de uso pode levar a serviços complexos, cuja decomposição necessita de heurísticas adicionais);
- refinar as heurísticas de concepção de seus três tipos de objetos (de interface, entidade e de controle).

A escolha da metodologia *OOSE* (também referenciada com o nome de *Objectory*) como base para este trabalho se deve ao poder heurístico decorrente da análise baseada em casos de uso e da caracterização de três tipos de objetos (de interface, entidade e de controle) propostas na metodologia *OOSE*. A escolha parece ter sido bastante acertada, segundo a opinião publicada recentemente (setembro de 1994) de um dos mais importantes metodologistas no paradigma orientado a objetos cujo texto [Rumbaugh94c] é transcrito abaixo.

"Most methodologists now agree that user-centered analysis is the best way to solve the right problem. Capturing the user's needs is a major focus of several metodologies, including Rubin and Goldberg's OBA and Jacobson's Objectory. In particular, Jacobson's use cases have been well received by just about every methodologist including us."

Por este motivo, características fundamentais da metodologia *OOSE* se encontram presentes na Especificação Unificada:

- especificação dos requisitos de um sistema a partir dos seus casos de uso (que correspondem a serviços de sistema na Especificação Unificada);
- determinação dos casos de uso a partir dos atores que se utilizam do sistema;
- caracterização de três tipos de objetos (de interface, entidade e de controle);
- descrição da interface faz parte da especificação de requisitos.

A metodologia *OOSE* utiliza-se de heurísticas de reuso [Johnson88] referenciadas em muitos outros trabalhos posteriores que abordam reusabilidade de software e que também foram incorporadas à Especificação Unificada:

- concepção de serviços primitivos para facilitar a composição de outros serviços;
- utilização de poucos argumentos na interface de um serviço;
- não utilização de parâmetros de controle como argumentos de serviços.

7.1.2 Correspondência a Conceitos de Outras Metodologias

O conceito de delegação tem correspondência com o de colaboração da metodologia *OOSD* [Wirfs-Brock90], no sentido de que em ambos a decomposição de um serviço se utiliza de outros serviços. A diferença básica está na concepção dos serviços utilizados como colaboradores. Na metodologia *OOSD*, para classe de objetos, determinada a partir de especificações em linguagem natural, são atribuídas responsabilidades; e para cada responsabilidade são identificadas colaborações. Na Especificação Unificada, delegação faz parte de uma heurística mais geral, baseada na identificação das classes responsáveis (ou de classes que as controlam) pela informação manipulada pelo serviço em decomposição.

Vinculação de requisitos tem sido proposta de diferentes formas:

- caracterização explícita de alternativas de projeto [Potts88] [Lubars91];
- vinculação de modelo de fragmentos (baseados em parágrafos ou seções retirados de especificações de requisitos em linguagem natural) a diagramas de relacionamentos de objetos ou a diagramas de estados [Lubars93].

Na Especificação Unificada, somente nomes de requisitos (não extraídos de parágrafos ou seções de linguagem natural e sem caracterizar alternativas de projeto) representando restrições ou propriedades adicionais são utilizadas para justificar a decomposição de serviços através de operadores de ordenação. Ou seja, adota-se a idéia de vincular requisitos diferindo, no entanto,

na forma de definir tais requisitos e no papel que eles exercem como guias na decomposição do sistema.

A distinção de tipos de serviços (solicitação de informação, armazenamento ou descarte de informação, e execução de serviço) tem semelhanças com a proposta de atividades de interações especiais (acessar, modificar, remover, destruir, somar, criar) entre objetos em diagramas de interação da metodologia OSA [Embley92]. No entanto, a sua utilização na Especificação Unificada é marcada por diferenças básicas:

- de motivação: decorre da incorporação de conceitos provenientes de diagramas de fluxo de dados (armazenagem e recuperação de informações em depósitos de dados) a uma linguagem textual baseada em diagramas de interação entre objetos (por considerá-la a representação gráfica fundamental para expressar a construção de sistemas de software como uma cooperação entre objetos);
- de objetivo: viabilizar a criação de filtros de recuperação de níveis mais altos de abstração a partir da representação da Especificação Unificada.

Finalmente, o mapeamento de conceitos em classes de objetos tem correspondência com a definição de visões da metodologia OSA [Embley92]. Mas a sua utilização na Especificação Unificada é substancialmente diferente, uma vez que somente um nível de abstração é mantido (acrescido de informação de mapeamento), a partir do qual visões são recuperadas. O esquema proposto na Especificação Unificada simplifica a manutenção da documentação do sistema de software, ao propor uma única forma de representação.

7.1.3 Contribuições da Metodologia Especificação Unificada

Uma metodologia de desenvolvimento de sistemas de software é caracterizada por dois aspectos ortogonais:

- sua linguagem de representação;
- seu corpo de heurísticas (que tende a crescer e a ser refinado à medida que a metodologia amadurece).

A metodologia Especificação Unificada distingue-se essencialmente de outras metodologias de análise e projeto orientadas a objetos ao propor uma forma de representação única a partir da qual são recuperadas visões correspondentes a níveis mais altos de abstração.

A representação de requisitos, em vez de definir classes de objetos, atributos e serviços baseados em substantivos, adjetivos e verbos de descrições textuais, utiliza restrições sobre a linguagem natural para definir nomes para conceitos (e seus relacionamentos), serviços e requisitos adicionais. A descrição da interface com os usuários e a caracterização de eventos compostos (a partir de eventos primitivos) são formalizadas como base para a concepção das classes de objetos de interface.

O mapeamento explícito de conceitos em classes de objetos e a descrição esquemática dos serviços do sistema (em função de seus estados e eventos) é mantido na Especificação Unificada para viabilizar a recuperação dos conceitos e dos serviços do sistema.

A representação da Especificação Unificada corresponde a uma linguagem textual baseada na formalização de construções do diagrama de interação entre objetos, acrescido de informação proveniente do diagrama de fluxo de dados e do diagrama de transição de estados. A partir da linguagem de representação da Especificação Unificada podem ser recuperadas, de forma seletiva (automática ou interativa), os diagramas que constituem a representação gráfica das metodologias orientadas a objeto.

O diagrama de interação de estados, do qual provêm a descrição de solicitação de serviço com suas listas de parâmetros e de resultados, e os operadores de ordenação de serviços, é a base para representar um sistema descrito como uma cooperação de objetos; pelo diagrama trafega todo o fluxo de dados do sistema (devido ao encapsulamento de dados do paradigma orientado a objetos [Rumbaugh94b]). Do diagrama de fluxo de dados provêm as construções que distinguem entre solicitação de informações, processamento da informação, descarte ou armazenamento de informação (que exercem um papel fundamental na decomposição de serviços). Do diagrama de transição de estados provêm as informações de estados, de eventos e de próximo estado (cuja utilização é justificada somente para representar a assincronia de interação com usuários ou sistemas externos).

Algumas abstrações (conjunto e iteração sobre os elementos do conjunto) são definidas para isolar o domínio do problema do domínio computacional. Definições de trechos de serviços são utilizadas para distinguir níveis de abstração e facilitar a recuperação de visões.

As classes de interface são mapeadas a partir das regiões que compõem a interface com os usuários. As classes do núcleo do sistema são mapeadas a partir dos conceitos e da informação manipulada pelos serviços do sistema.

Para decompor a parte reativa (na qual o sistema é decomposto pelos serviços que presta) e a parte não reativa (na qual os serviços, decorrentes da decomposição da parte reativa, são decompostos em função da informação que manipulam) de sistemas de software é definida uma heurística básica.

Adicionalmente, heurísticas de alocação de controle são propostas como refinamento às heurísticas propostas pela metodologia OOSE [Jacobson92] para classes de interface, classes agregadoras e classes específicas de controle. O acesso a classes agregadas é recomendado em relação ao repasse de serviços pelas classes agregadoras.

Diagramas gráficos têm sidos acrescidos de grande quantidade de informação [Embley92] [Rumbaugh94b] com o objetivo de capturar aspectos de especificação adicionais (desconsiderados anteriormente). A recuperação de visões, correspondentes a estes diagramas (com representação gráfica associada), com a possibilidade de selecionar iterativamente a informação a ser recuperada (e portanto representada nos diagramas), pode ser considerada uma alternativa mais flexível e

definitiva aos constantes acréscimos de tipos de informação aos diagramas de especificação. Acrescente-se a isto o fato de que a recuperação de visões elimina a distância representacional entre os diagramas utilizados para especificar as várias fases de desenvolvimento de software.

A recuperação iterativa e seletiva de visões juntamente com o rastreamento de requisitos (a partir da vinculação dos requisitos) tem conseqüências fundamentais na capacidade de fornecimento de informação para entendimento do sistema bem como na definição de ações de manutenção corretiva e evolutiva.

A forma de representação serve de base para a geração automática de implementação (apesar de requerer estudos suplementares) em uma linguagem orientada a objetos (como C++), desde que supridas informações adicionais de tipagem e parametrização de mapeamento (o que pode ser considerado como acréscimo marginal em relação à informação necessária para o mapeamento).

7.2 Comparação com as Premissas

Neste trabalho, metodologias de análise e projeto orientadas a objeto foram avaliadas segundo duas premissas propostas (que se distinguem dos critérios de avaliação propostos em *surveys* de metodologias orientadas a objeto): adequação representacional e poder heurístico.

Adequação representacional foi definida como:

- captura concisa, não ambígua e consistente dos requisitos do sistema;
- manutenção do vínculo entre os artefatos gerados nos vários níveis de abstração (que depende basicamente de dois fatores: continuidade representacional e decomposição do sistema guiada e justificada pelos seus requisitos).

Poder heurístico foi definido em função da capacidade do corpo de heurísticas da metodologia em guiar a decomposição do sistema em artefatos que atendam os requisitos do sistema, promovam reuso e restrinjam a propagação de alterações por decorrência de ações de manutenção.

A metodologia proposta neste trabalho deve atender as premissas acima e responder as seguintes questões:

- que elementos compõem uma especificação de requisitos e como expressá-la concisamente?
- como mapear requisitos em classes de objetos, procurando explicitar apenas o mapeamento necessário?
- como manter o vínculo com os requisitos em cada um desses mapeamentos?
- existe equivalência entre construções utilizadas nas diversas fases de especificação (requisitos, análise, projeto e implementação)?
- é possível recuperar visões distintas das classes de objetos a partir de uma única representação (evitando manutenção de vários níveis e eliminando a quebra de continuidade representacional)?

- é possível especificar um sistema de software como uma cooperação de objetos em todos os níveis de abstração?

A análise que segue propõe-se a justificar porque a metodologia Especificação Unificada pode ser considerada como uma alternativa de resposta a estas questões.

Um sistema de software é definido a partir dos serviços que presta. Cada serviço do sistema é definido a partir da composição de outros serviços (da mesma forma, nas fases de análise e projeto). A composição de serviços utiliza operadores de seqüência, seleção e de iteração (que se fazem presentes em todos os níveis de abstração do desenvolvimento do sistema). A definição de tais serviços utiliza-se do jargão estabelecido pelo domínio do problema (conceitos). Restrições ou propriedades adicionais à definição de serviços podem ser caracterizadas através de requisitos adicionais.

Estes elementos (conceitos, serviços e requisitos adicionais) são representados por nomes cuja sintaxe corresponde a restrições sobre a linguagem natural, mantendo-se apenas as palavras necessárias ao entendimento (separadas por *underscores* para distinguir palavras minúsculas ou aumentar a legibilidade de nomes longos), sem substituí-las por abreviações (com exceção dos casos em que a abreviação faz parte do jargão do domínio do sistema). Identificados de forma concisa e legível, estes três elementos são suficientes para representar requisitos de sistemas de software. Concisão e legibilidade na forma de representação dos requisitos são premissas para análises de completitude, não ambiguidade e consistência de requisitos.

A captura dos requisitos é considerada adequada em função dos seguintes aspectos:

- abrangente quanto ao poder de expressão: baseada em restrições de linguagem natural;
- concisa: captura mínima informação necessária para sua legibilidade, sem perda de informação com abreviações;
- genérica: qualquer requisito pode ser expresso através de conceitos, serviços e requisitos adicionais;
- essencial: um sistema é definido pelos serviços que presta.

A decomposição dos serviços do sistema é guiada pelos requisitos adicionais e pela informação manipulada pelos serviços. A aplicação de operadores de ordenação de serviços é associada, sempre que cabível, ao requisito que a justifica. As classes responsáveis pelas informações manipuladas pelo serviço, em decomposição, são mapeadas a partir dos conceitos. O mapeamento só necessita ser explicitado quando um conceito é associado a mais de uma classe (uma classe mapeada a partir de um conceito conserva o seu nome, dispensando mapeamento explícito).

A continuidade representacional é garantida pela utilização de uma única forma de representação mapeada a partir dos requisitos do sistema (conceitos, serviços do sistema e requisitos adicionais). A manutenção de informação de mapeamento explícito associada a construções que distinguem tipos de serviços (solicitação de informação, solicitação de execução de serviço, descarte e armazenamento de informação) e níveis de abstrações distintos (definições) permite recuperar de forma iterativa e seletiva visões associadas a diferentes modelos de representação (de

classes de objetos, de interação entre objetos e de transição de estados) em diferentes níveis de abstração.

A rastreabilidade de requisitos da Especificação Unificada suporta manutenção corretiva e evolutiva, sem perda de consistência com os requisitos existentes e com recursos de composição em função das alterações ou adições necessárias. As falhas detectadas ou os novos requisitos para evolução são consideradas informações de entrada para a fase de manutenção corretiva e evolutiva. Aspectos relacionados à detecção de falhas ou à avaliação de necessidade de evolução não são considerados pela metodologia.

Se o procedimento for de manutenção corretiva, a falha reportada deve ser analisada a partir da visão do nível de abstração mais detalhado da especificação (juntamente com as informações adicionais para implementação). Detectado o motivo da falha, uma análise *bottom-up* deve ser feita, do nível mais detalhado até o nível de requisitos, para avaliar a partir de que nível de abstração devem ser feitas as correções.

Se o procedimento for de manutenção evolutiva, deverá ser feita uma análise *top-down* comparando os requisitos do sistema com os novos requisitos, a serem descritos como alterações ou adições dos conceitos, serviços e requisitos adicionais do sistema (de forma a permitir o rastreamento das classes e dos serviços afetados, nos níveis mais baixos de abstração).

Novos serviços ou novas classes de objetos, decorrentes da manutenção corretiva ou evolutiva, deverão ser implementados a partir da composição dos serviços e classes existentes. A recuperação de tais componentes é facilitada pelo mecanismo de visões de níveis de abstração. Se não for possível compor os novos serviços totalmente a partir dos serviços existentes, novos serviços elementares deverão ser detalhados. A manutenção, desta forma, mantém a consistência com os requisitos existentes, buscando explorar ao máximo a composição das alterações ou adições.

Uma heurística é proposta como base para a alocação de controle na decomposição de serviços associados à parte reativa e não reativa de sistemas de software. Heurísticas adicionais refinam a concepção das classes de interface, do núcleo do sistema e especificamente de controle, como proposta pela metodologia *OOSE* [Jacobson92].

A metodologia Especificação Unificada agrega construções e heurísticas que viabilizam a concepção de objetos primitivos, fatorados, genéricos e vinculados aos requisitos que os justificam. Essa somatória de fatores promove a concepção de um sistema mais robusto em relação a ações de manutenção corretiva e evolutiva, concorrendo para minimizar a propagação de alterações.

A especificação de requisitos baseada nos serviços prestados pelo sistema, decomposta em serviços de classes mapeadas a partir da descrição da interface e dos conceitos, define a representação do sistema como uma cooperação de objetos em todos os níveis de abstração.□

7.3 Abrangência da Metodologia Especificação Unificada

A metodologia Especificação Unificada abrange os seguintes aspectos do desenvolvimento de sistemas de software:

- especificação dos requisitos dos sistemas;
- decomposição (análise e projeto) da parte reativa dos sistemas;
- decomposição da parte não reativa dos sistemas;
- especificação e rastreamento de requisitos de tempo real;
- suporte à recuperação e ao entendimento de artefatos reusáveis dos sistemas.

A especificação dos requisitos dos sistemas de software corresponde à descrição dos conceitos (vocabulário do domínio do sistema), serviços de sistemas (requisitos funcionais de utilização do sistema do ponto de vista dos usuários), requisitos adicionais (que permitem definir restrições ou propriedades que complementam a descrição dos serviços) e da interface. A descrição da interface engloba a descrição dos elementos da interface e dos eventos a eles associados.

A decomposição da parte reativa de sistemas de software corresponde ao mapeamento dos requisitos nas classes de interface. As classes de interface são responsáveis pela troca de informações do sistema com o seu meio ambiente e sua concepção está basicamente associada às necessidades de tratamento dos eventos identificados nos serviços de sistema.

A decomposição da parte não reativa de sistemas de software corresponde ao detalhamento dos serviços decorrentes da decomposição da parte reativa. Neste caso, a decomposição passa a ser guiada pelas necessidades de tratamento da informação utilizada pelo sistemas, uma vez que a influência dos eventos como heurística de decomposição se esgota na parte reativa do sistemas.

Restrições de tempo real, tais como tempo máximo de resposta ou periodicidade de execução, podem ser expressas através de requisitos adicionais e portanto, associadas a serviços ou a seqüências de serviços. A recuperação de visões correspondentes a diagramas de interações entre objetos permite rastrear requisitos de tempo real, em decorrência do fato de que tais diagramas ordenam seqüências de interações ao longo do eixo do tempo.

O reuso de artefatos de um sistema de software aplicativo, em função das necessidades decorrentes de ações de manutenção corretiva ou evolutiva, é fortemente dependente do suporte à recuperação e ao entendimento dos componentes do sistema. A metodologia Especificação Unificada provê este suporte através de sua linguagem de representação (que permite associar conceitos e requisitos aos elementos que compõem a especificação do sistema) e do mecanismo de recuperação de visões.

A metodologia Especificação Unificada não abrange os seguintes aspectos do desenvolvimento de sistemas de software:

- especificação de sistemas concorrentes;

- projeto de sistemas.

Sistemas de software de tempo real são, em geral, baseados em concorrência de processos. O conceito de processo é ortogonal ao conceito de objeto: um processo pode envolver vários objetos e um objeto pode estar alocado a vários processos. O comportamento associado aos casos de uso (ou serviços de sistema) provê a base para a decomposição de um sistema de software em processos [Jacobson92]. A especificação de sistemas concorrentes demanda as seguintes extensões à metodologia Especificação Unificada:

- heurísticas de mapeamento dos serviços de sistema em processos;
- notação de representação da definição, criação e remoção de processos;
- representação da comunicação e sincronização entre processos.

A partir das extensões para representação de processos o mecanismo de recuperação de visões terá informação suficiente para recuperar diagramas de interações de processos. Tais diagramas se diferenciam dos diagramas de interações de objetos por representar processos e sinais em substituição aos objetos e mensagens.

Extensões também são necessárias para especificar a arquitetura e a decomposição de sistemas de software (atividades englobadas no projeto de sistemas), particularmente importante para o desenvolvimento de grandes sistemas de software. Essas atividades independem, em geral, do paradigma utilizado [Rumbaugh91].

7.4 Experiência de Utilização da Metodologia Especificação Unificada

A metodologia Especificação Unificada foi aplicada em algumas partes do *CSD*. O software do *CSD* é composto de várias partes com características distintas: controle da interface (a nível de recepção de eventos e *display* de informação), controle do *CSD*, núcleo do *CSD*, monitor e biblioteca de depuração remotos.

O controle da interface, a nível de recepção de eventos e *display* da informação, foi implementado utilizando uma linguagem específica (interpretada ou compilada por um gerador de interfaces gráficas baseado no padrão *Motif*) e adicionalmente utilizando a linguagem C para a implementação de algumas rotinas. A função desta parte se restringe à ativação de *callbacks* em função da recepção de eventos e ao *display* de janelas gráficas e de informação na interface.

O controle do *CSD* foi projetado segundo a metodologia Especificação Unificada, implementado na linguagem C++ e testado com base na sua especificação. A cada janela da interface, ou a cada região de janela com características específicas de entrada e saída, foi associada uma classe de objetos da interface. Alguns dos serviços dessas classes são ativadas como *callbacks* pelo controle da interface a nível de eventos. As principais funções dessa parte do *CSD* são:

- converter informação de interface para informação entendida pelo núcleo do *CSD* por ocasião da recepção de eventos e vice-versa, e por ocasião do *display* de informação na interface;

- controlar a atualização da interface, geralmente em função de informação solicitada ao núcleo, quando ocorre parada ou terminação de execução de programa em depuração;
- solicitar, ou eventualmente controlar, a execução de serviços do núcleo do *CSD*.

O núcleo do *CSD* foi implementado na linguagem C++ baseado no paradigma orientado a objetos mas, sem prévia realização de análise e projeto, pois grande parte da sua implementação já havia sido realizada quando a metodologia Especificação Unificada começou a ser utilizada no *CSD*. Suas principais funções são:

- armazenar e prover informações estáticas sobre o programa em depuração (linhas executáveis e símbolos de programa fonte) obtidas a partir da conversão da tabela de símbolos gerada pelo compilador (com informações para depuração simbólica);
- armazenar e prover informações dinâmicas sobre o programa em execução (pilha de execução, *breakpoints* instalados pelo usuário);
- controlar a execução remota do programa em depuração.

O núcleo do *CSD*, excetuando o tratamento de variáveis e expressões e a comunicação com o programa em execução, foi reprojetado a partir da metodologia Especificação Unificada sem o objetivo de alterar a implementação em função desse reprojeto mas, sim, de testar as construções da linguagem de representação da Especificação Unificada de forma mais abrangente. Esse reprojeto permitiu refinar as construções suportadas pela Especificação Unificada mas não serviu como teste para o desenvolvimento de sistemas de software porque utilizou informação previamente detalhada na implementação.

O monitor remoto e a biblioteca de depuração (chamada pelo programa em execução para comunicação com o núcleo do *CSD*) foram escritos em linguagem *CHILL*. Suas principais funções são: instalação e controle do programa em execução remota e detecção de parada do programa.

A aplicação da metodologia Especificação Unificada no desenvolvimento e teste da parte de controle do *CSD* foi realizado de forma abrangente. A especificação gerada foi utilizada para recuperar visões de níveis mais altos de abstração (manualmente devido à falta do ambiente de suporte à metodologia), de forma a orientar o desenvolvimento e os testes do controle do *CSD*. O *CSD* ainda não foi liberado para os usuários e, portanto, ainda não foi possível avaliar o impacto da utilização da Especificação Unificada em ações de manutenção corretiva e evolutiva.

O refinamento da metodologia Especificação Unificada, tanto na sua forma de representação quanto nas suas heurísticas, trouxe alguns desafios (de abrangência e de consistência) devido ao fato de estar ocorrendo em paralelo com sua primeira aplicação em um sistema de software real. Essas dificuldades iniciais foram superadas à medida que a Especificação Unificada foi se estabilizando. A aplicação da Especificação Unificada no desenvolvimento completo e na manutenção de um sistema de software de, pelo menos, médio porte, é o próximo passo em direção ao objetivo de atingir sua maturidade.

8. Conclusões

8.1 Resultados Alcançados

Como base para o proposta da metodologia Especificação Unificada foram realizadas as seguintes atividades:

- análise das vantagens e fraquezas das abordagens de reuso existentes, com o objetivo de caracterizar as necessidades associadas ao reuso de software aplicativo;
- definição de premissas para caracterização de metodologias de análise e projeto orientadas a objetos e análise comparativa de reconhecidas metodologias segundo tais premissas, com o objetivo de caracterizar os pontos a serem cobertos por uma proposta de metodologia;
- apresentação de um questionamento adicional de pontos a serem cobertos pela proposta de metodologia.

A forma de representação da Especificação Unificada é única para as fases de especificação de requisitos, análise e projeto. As construções utilizadas pela linguagem são detalhadas e ilustradas para prover a representação de requisitos (com especificação para interfaces gráficas com o usuário, devido a sua larga utilização atual em sistemas reativos), das classes e serviços de análise e projeto (em geral, as classes de projeto associadas ao encapsulamento de estruturas de dados tem sua representação abstraída pela utilização da classe especial *SET*).

Heurísticas de decomposição de sistemas de software são propostas com ênfase na cobertura da parte não reativa dos sistemas (coberta apenas superficialmente pelas propostas existentes).

O papel fundamental do mecanismo de Recuperação de Visões na utilização da metodologia Especificação Unificada é ilustrado com exemplos extraídos de sua aplicação no depurador simbólico *CHILL* (*CSD*). A opção pela ilustração da aplicação da metodologia a partes de um sistema de software real de médio porte, em vez da costumeira ilustração com *toy problems*, tem como objetivo fornecer uma amostragem do grau de amadurecimento da metodologia proposta.

A caracterização de metodologia Especificação Unificada no contexto de outras propostas e das premissas definidas previamente, da sua abrangência e da experiência da sua utilização se constitui em informação essencial para a avaliação da contribuição científica do presente trabalho.

8.2 Sugestões para Trabalhos Futuros

Para a continuidade deste trabalho pode-se propor esforços direcionados aos seguintes aspectos:

- geração automática de implementação;
- ambiente de suporte;
- refinamento da metodologia através da sua utilização;

- avaliação do impacto de ações de manutenção;
- extensões à metodologia.

O estudo e a definição do mapeamento entre a linguagem de representação da Especificação Unificada e uma linguagem de implementação orientada a objetos (por exemplo C++) tem como objetivo viabilizar a geração automática de implementação a partir da representação da Especificação Unificada e de informação complementar (a ser mantida em um documento separado, para não se misturar com a especificação) constituída de informação de tipagem e de *defaults* de mapeamento (específicos para a linguagem de programação escolhida).

Um ambiente de desenvolvimento de software, baseado na metodologia Especificação Unificada, deve ser projetado e desenvolvido para dar suporte à representação dos requisitos iniciais e da Especificação Unificada, à recuperação seletiva (automática ou interativa) de visões e à geração automática da implementação.

O refinamento da linguagem de representação e do corpo de heurísticas da metodologia Especificação Unificada deve ser realizado através da sua utilização, apoiada pelo seu ambiente de suporte, na especificação e desenvolvimento de sistemas de software no paradigma orientado a objetos.

A partir da utilização da metodologia Especificação Unificada no desenvolvimento de sistemas de software deve ser feito um estudo e uma avaliação das conseqüências decorrentes de ações de manutenção corretiva e evolutiva em relação à estabilidade e capacidade de evolução dos sistemas em questão.

Extensões à metodologia Especificação Unificada podem englobar suporte a projeto de sistemas e suporte a sistemas concorrentes. A utilização da metodologia para a especificação de sistemas concorrentes e de tempo real é fundamental para refinar a forma de representação e as heurísticas específicas que forem acrescentadas à metodologia para a representação e a decomposição de tais sistemas.

A Figura 32 ilustra os trabalhos futuros propostos à metodologia Especificação Unificada e o papel fundamental do ambiente de suporte à metodologia no seu estado atual e futuro.

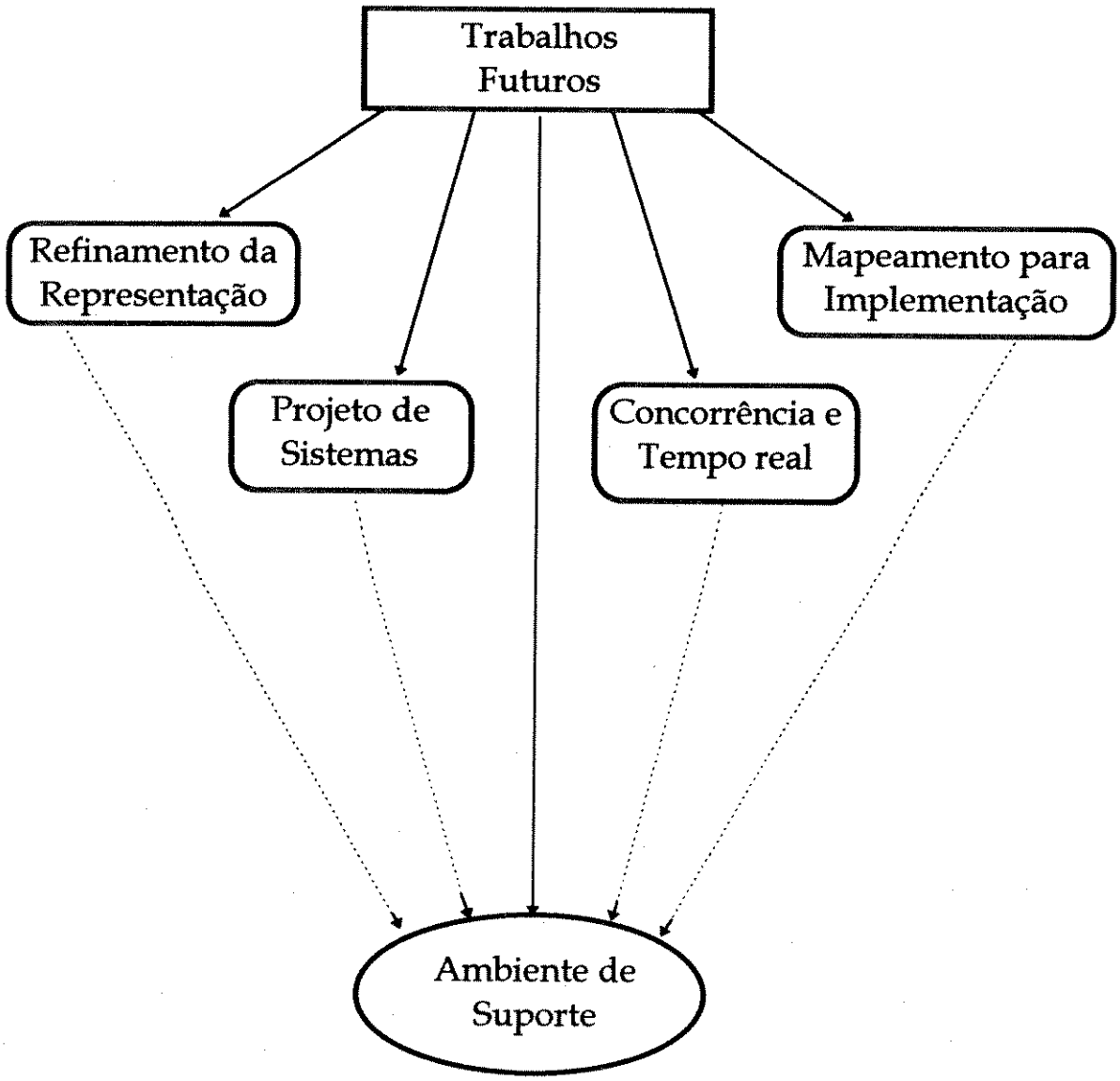

Figura 32: Trabalhos Futuros propostos e o Papel Fundamental do Ambiente de Suporte

Referências

[Abbott83] R. Abbott;
Program Design by Informal English Descriptions;
Communications of ACM, vol. 26, n. 11.

[Alabios88] B. Alabios;
Transformation of Data Flow Analysis Models to Object Oriented Design;
SIGPLAN Notices, vol. 23, n. 11.

[Batista93] J. Batista Jr;
Relatório Técnico : Metodologias de Análise e Projeto Orientado a Objeto;
Biblioteca do CPqD-TELEBRÁS, 1993.

[Beck89] K. Beck, W. Cunningham;
A Laboratory for Teaching Object-Orriented Thinking;
SIGPLAN Notices, vol. 24, n. 10.

[Biggerstaff87] T. J. Biggerstaff, C. Richter;
Reusability Framework, Assessment and Directions;
IEEE Software, March 1987, pp 41-9.

[Booch86] G. Booch;
Object Oriented Development;
IEEE Transactions on Software Engineering, February 1986.

[Booch91] G. Booch;
Object Oriented Design (with applications);
The Benjamin/Cummings Publishing Company, 1991.

[Constantine92] L. Constantine, M. Page-Jones, I. Jacobson, J. Palmer;
From Events to Objects: The Heresy of Event-Orientation in a World of Objects;
Panel OOPSLA'92, pp 295-7.

[Chaves94] G. L. M. Chaves;
CSD - X11 Based Source Level Debugger;
CHILL Project, Telebrás R&D Center, October, 1994.

[Coad91] P. Coad, E. Yourdon;
Object-Oriented Analysis;
Yourdon Press, 1991.

[DeMarco79] T. DeMarco;
Structured Analysis and System Specification;
Prentice-Hall, 1979.

[Embley92] D. W. Embley, B. D. Kurtz, S. N. Woodfield;
Object-Oriented System Analysis (A Model-Driven Approach);
Yourdon Press, 1992;

[Frakes88] W. B. Frakes, P. B. Gandel;
Representing Reusable Software: Survey and Model;
Software Productivity Consortium, December, 1989.

[Freeman83] P. Freeman;

Reusable Software Engineering: Concepts and Research Directions;
Proc. of Workshop on Reusability in Programming, September 1993, pp 63-76;

[Freeman87] P. Freeman;
A Conceptual Analysis of the Draco Approach to Constructing Software Systems;
IEEE Transactions on Software Engineering, July 1987, pp 830-44;

[Hoydalsvik93] G. M. Hoydalsvik, G. Sindre;
On the purpose of Object-Oriented Analysis;
Proceedings OOPSLA'93, pp 240-50.

[Jackson75] M. Jackson;
Principles of Program Design;
Academic Press, 1975.

[Jacobson92] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard;
Object-Oriented Software Engineering (A Use Case Driven Approach);
Addison-Wesley Publishing Company, 1992.

[Johnson88] R. E. Johnson, B. Foote;
Design Reusable Classes;
Journal of Object-Oriented Programming, June/July, pp 22-35.

[Krueger92] C. W. Krueger;
Software Reuse;
ACM Computing Surveys, June 1992.

[Lubars91] M. D. Lubars;
Representing Design Dependencies in an Issue-Based Style;
IEEE Software, July 1991, pp 81-89.

[Lubars92] M. D. Lubars, G. Meredith, C. Potts, C. Richter;
Object-Oriented Analysis for Evolving Systems;
Proc. of the 14th Int. Conf. on Software Enginnering, 1992, pp 173-85.

[Lubars93] M. Lubars, C. Potts, C. Richter;
Developing Initial OOA Models;
Proc. of the 15th Int. Conf. on Software Enginnering, 1993, pp 255-64.

[McMenamin84] S. M. McMenamin, J. F. Palmer;
Essential Systems Analysis;
Yourdon Press, 1984.

[Neighbors84] J. M. Neighbors;
The Draco Approach to Constructing Software from Reusable Components;
IEEE Transactions on Software Engineering, September 1984, pp 564-74.

[Parnas72] D. Parnas;
On the criteria to be used in decomposing systems into modules;
Communications of ACM, Feb 72, pp 1053-58.

[Potts88] C. Potts, G. Bruns;
Recording the Reasons for Design Decisions;
Proc. of the 10th Int. Conf. on Software Enginnering, 1988, pp 418-27.

[Rubin92] K. S. Rubin, A. Goldberg;
Object Behavior Analysis;

Communications of the ACM, Sep 92, pp 48-62.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen;
Object-Oriented Modeling and Design;
Prentice Hall, 1991.

[Rumbaugh94a] J. Rumbaugh;
The life of an object model;
Journal of Object-Oriented Programming, March-April 1994.

[Rumbaugh94b] J. Rumbaugh;
Going with the flow (Flow graphs in their various manifestations);
Journal of Object-Oriented Programming, June1994.

[Rumbaugh94c] J. Rumbaugh;
Getting started (Using use cases to capture requirements);
Journal of Object-Oriented Programming, September 1994.

[Shlaer92] S. Shlaer, S. Mellor;
Object Lifecycle (Modeling the World in States);
Yourdon Press, 1992.

[Stroustrup88] B. Stroustrup;
What is Object-Oriented Programming?;
IEEE Software, May 1988.

[Yourdon79] E. Yourdon, L. Constantine;
Structured Design;
Prentice-Hall, 1979.

[Warnier74] J. Warnier;
Logical Construction of Programs;
Van Nostrand Reinhold, 1974.

[Wegner89] P. Wegner;
Capital-intensive Software Technology;
Software Reusability, vol. I, 1989, pp 247-56.

[Wirfs-Brock90] R. Wirfs-Brock, B. Wilkerson, L. Wiener;
Designing Object-Oriented Software;
Prentice Hall, 1990.

Apêndice A: Sintaxe da Linguagem de Representação da Especificação Unificada

<system> ::= <system_header> <interface_description> <requirement_mapping>
<service_layers>
<system_header> ::= SYSTEM <system_name>
<system_name> ::= <name1>
<interface_description> ::= { <layout_window> <mouse_window> }*

<layout_window> ::= <layout_header> { <layer> | <partition> }+
<layout_header> ::= LAYOUT <window_name>
<window_name> ::= <name1>
<layer> ::= <layer_header> <internal_region>
<partition> ::= <partition_header> <internal_region>
<layer_header> ::= LAYER [<interface_element>]
<partition_header> ::= PARTITION [<interface_element>]
<internal_region> ::= { <tab> (<layer> | <partition>) }*
<interface_element> ::= <interface_element_name_list> [":" { <interface_element_name>
| (SET "(" <interface_element_name> ")") }]
<interface_element_name_list> ::= <interface_element_name> { ","
<interface_element_name> }*
<interface_element_name> ::= <name2>

<mouse_window> ::= <mouse_header> <mouse_button>
<mouse_header> ::= MOUSE <window_name>
<mouse_button> ::= (LEFT | MIDDLE | RIGHT) <mouse_function_name> <mouse_body>
<mouse_function_name> ::= <name1>
<mouse_body> ::= { <tab> (<interface_element_name> | <interface_menu>) }+
<interface_menu> ::= <menu_name> "(" <interface_element_name> ") " "=" "["
<menu_option_name_list> "] "
<menu_name> ::= <name3>
<menu_option_name_list> ::= <menu_option_name> { "," <menu_option_name> }*
<menu_option_name> ::= <name1>

<requirement_mapping> ::= <concept_set> <concept_mapping_set> <system_services_set>
<internal_state_set>
<concept_set> ::= CONCEPT <concept_name_list>
<concept_name_list> ::= <concept_name> { "," <concept_name> }*
<concept_name> ::= <name1>

<concept_mapping_set> ::= { MAPPING <concept_name> ":" <classe_name_list> }
<classe_name_list> ::= <classe_name> { "," <classe_name> }*
<classe_name> ::= <name1>
<system_services_set> ::= { <system_service> }+
<system_service> ::= SERVICE <service_name> { <event_alternative> }*
<service_name> ::= <name1>
<event_alternative> ::= STATE <state_name> { EVENT <event_name> }+
<state_name> ::= <name1>
<event_name> ::= <name1>
<internal_state_set> ::= { <internal_state> }*
<internal_state> ::= STATE <concept_name> INTERNAL <concept_name>

<service_layers> ::= { [<service_layer_header>] { <object_classes> }+ }+
<service_layer_header> ::= SERVICE LAYER <layer_name>
<layer_name> ::= <name1>
<object_class> ::= <class_header> <class_data> <class_services>

<class_header> ::= CLASS <class_name> [<template>] [":" <inherited_class_name>
[<template>]]
<inherited_class_name> ::= <name1>
<template> ::= "<" <argument_name_list> ">"
<argument_name_list> ::= <argument_name> { "," <argument_name> }*

<class_data> ::= { <enumerated_set> | <data> }*
<enumerated_set> ::= "%" <enumerated_set_name> "=" "[" <element_name_list> "]"
<enumerated_set_name> ::= <name3>
<element_name_list> ::= <element_name> { ","<element_name> }*
<element_name> ::= <name3>
<data> ::= "." [<relationship_name> ":"] <type>
<type> ::= [<prefix_name> "/"] <radical_name>
<relationship_name> ::= <name3>
<prefix_name> ::= <name3>
<radical_name> ::= <attribute_name> | <class_name> | <data_set>
<attribute_name> ::= <name3>
<data_set> ::= SET "(" <class_name> [":" <index_name>] ")"
<index_name> ::= <name2>

<class_services> ::= { <public_service> }+ { <internal_service> }* { <definition_service> }*

{ <definition_service_with_nextstate> }*

<public_service> ::= <service_header> <service_body>

<internal_service> ::= <internal_header> <service_body>

<definition_service> ::= <definition_header> <definition_body>

<service_header> ::= SERVICE <service_name> [<service_arguments>]

<internal_header> ::= INTERNAL <internal_name> [<service_arguments>]

<internal_name> ::= <name1>

<definition_header> ::= DEF <definition_name> [<service_arguments>]

<definition_name > ::= <name1>

<service_arguments> ::= "(" <parameter_type_list> ")" "[" ": " <return_type_list>]

<parameter_type_list> ::= <parameter_type> { "," <parameter_type> }*

<parameter_type> ::= <type>

<return_type_list> ::= <return_type> { "," <return_type> }*

<return_type> ::= <type>

<service_body> ::= [<comment>] [<requirement>] ({ <indented_action> }+ | { <state> }+
| { <state_finished_by_definition> }+)

<comment> ::= ";" <comment_name>

<comment_name> ::= <name4>

<requirement> ::= REQUIREMENT <requirement_name>

<requirement_name> ::= <name4>

<state> ::= <state_header> <state_transition>

<state_header> ::= STATE <state_name> ["(" <super_state_name> ")"]

<super_state_name> ::= <name1>

<state_transition> ::= { <indented_action> | <indented_nextstate> }+

<state_finished_by_definition> ::= <state_header> <state_transition_finished_by_definition>

<state_transition_finished_by_definition> ::= { <indented_action> | <indented_nextstate>
| <definition_service_with_nextstate> }+

<definition_service> ::= <definition_header> <definition_body>

<definition_body> ::= { <indented_action> }+

<definition_service_with_nextstate> ::= <definition_header> <definition_body_with_nextstate>

<definition_body_with_nextstate> ::= { <indented_action> | <nextstate> }+

<indented_nextstate> ::= <tab> NEXTSTATE [<state_name>]

<indented_action> ::= <tab> <action>

<action> ::= <comment> | <requirement> | <design> | <definition> | <selection> | <iteration>
| <service_request> | <information_reading> | <information_save>

```
                | <information_discard> | <exception>
<design> ::= DESIGN <requirement_name>
<definition> ::= "%" ( <service_request> | <attribution> )
<information_reading> ::= "+" ( <service_request> | <set_indexation> )
<information_save> ::= "*" ( <construction> | <attribution> | <set_insertion> )
<information_discard> ::= "*" ( <remotion> | <set_remotion> )
<exception> ::= EXCEPTION <attribution>

<selection> ::= <selection_header> { <selection_alternative> }+ [ <other_alternative> ]
<selection_header> ::= CONSTRAINT <constraint_argument_list>
<constraint_argument_list> ::= <constraint_argument> { "," <constraint_argument> }*
<constraint_argument> ::= <comparison reference negation>
<comparison> ::= <reference comparator> ( <radical_name> | <condition_expression> )
<comparator> ::= "=" | "<" | ( "<" "=" ) | ">" | ( ">" "=" ) | "#"
< condition_expression> ::= expression
<negation> ::= "#" <reference>
<selection_alternative> ::= "=" ( <alternative_name> | ( <comparator> <expression> ) )
                            <alternative_body>
<alternative_name> ::= <name2>
<other_alternative> ::= OTHER <alternative_body>
<alternative_body> ::= { <indented_action> }+ [ <nextstate> ]

<iteration> ::= <iteration_header> <iteration_body>
<iteration_header> ::= ITERATION ALL "(" [ <prefix_name> "/" ] [ <class_name> "." ]
                       <set_reference> ")" [ <condition_expression> ]
<iteration_body> ::= { <indented_action> }+

<service_request> ::= [ <return_type_list> ] [ <class_name> ] "." <service_name>
                      [ "(" <parameter_type_list> ")" ]
<set_indexation> ::= <indexed_element_name> <set_reference>
<indexed_element_name> ::= <name2>
<set_reference> ::= <prefix_name> "/" SET "(" ( <attribute_name> | <class_name> ) ")"
<reference> ::= [ <prefix_name> "/" ] <reference_radical>
<reference_radical> ::= <class_name> | <attribute_name>
<attribution> ::= <reference> "=" ( <expression> | <reference> )
<set_indentation> ::= <class_name> <set_reference>
<construction> ::= <class_name> "(" <parameter_type_list> ")"
<destruction> ::= "~" <construction>
```

<set_insertion> ::= <prefix_set> "=" "+ " (<construction> | <prefix_attribute>)
<set_remotion> ::= <prefix_set> "=" " -" (<destruction> | <prefix_attribute>)
<prefix_set> ::= [<sufix_name> "/"] <set_reference>
<prefix_attribute> ::= [<sufix_name> "/"] <attribute_name>

<name1> ::= { <capital_word> | <upper_case_word> }+
<name2> ::= <name1> | <name3>
<name3> ::= <lower_case_word> { "_" <lower_case_word> }*
<name4> ::= <capital_word> { "_" <capital_word> }*

<capital_word> ::= "A-Z" { "a-z" }*
<upper_case_word> ::= { "A-Z" }*
<lower_case_word> ::= { "a-z" }*
<tab> ::= "/t"