Universidade Federal de Pernambuco
Centro de Informática

Detalhando o projeto arquitetural no desenvolvimento de software orientado a agentes: O caso Tropos

Por
Carla Taciana Lima Lourenço Silva
Dissertação de Mestrado

Recife,

Fevereiro de 2003

Carla Taciana Lima Lourenço Silva

Detalhando o projeto arquitetural no desenvolvimento de software orientado a agentes: O caso Tropos

Dissertação apresentada à Coordenação da Pos-Graduação em Ciência da Computação do Centro de Informática, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Orientador: Jaelson Freire Brelaz de Castro

Recife,

Fevereiro de 2003

Agradecimentos

Agradeço a Deus pelo apoio espiritual em todos os momentos e por permitir a realização deste sonho.

À Salete, minha mãe, o meu muito obrigada por tudo! Sem a sua presença na minha vida, eu não teria o suporte para chegar até aqui. Esta conquista é nossa!

A Jaelson, meu orientador, meus profundos agradecimentos por este período de crescimento pessoal e “lapidação” profissional, no qual eu encontrei o meu caminho de realização.

Aos meus colegas do Centro de Informática, eu agradeço pela amizade e pelos inesquecíveis momentos de descontração.

Resumo

O desenvolvimento orientado a agentes é bastante recente, no entanto, este novo paradigma tem sido utilizado cada vez mais em aplicações industriais, tais como telecomunicações e comércio eletrônico. Entre as principais preocupações para a consolidação desse novo paradigma, destacamos a necessidade de técnicas, notações e ferramentas adequadas para suportar o desenvolvimento de sistemas orientados a agentes. Neste sentido, o projeto Tropos está desenvolvendo uma abordagem de desenvolvimento orientado a agentes centrada em requisitos que visa suportar todas as fases do desenvolvimento de software orientado a agentes.

Tropos definiu vários estilos arquiteturais que focam em processos organizacionais e nos requisitos não-funcionais do software. Esses estilos arquiteturais organizacionais foram descritos inicialmente com a notação i*. Contudo, o uso do i* como uma linguagem de descrição arquitetural (ADL) é inadequado, pois apresenta algumas limitações sobretudo no que diz respeito ao detalhamento comportamental do projeto arquitetural.

Reconhecendo que a *Unified Modeling Language* (UML) pode ser estendida para atuar como uma linguagem de descrição arquitetural, apresentamos neste trabalho um conjunto de extensões baseadas na UML *Real Time* para representar os estilos arquiteturais organizacionais. Representar esses estilos arquiteturais em UML permitirá que o engenheiro de software insira mais detalhe ao projeto arquitetural do sistema. A fim de validar esta proposta, é apresentado um estudo de caso que utiliza o *framework* Tropos no desenvolvimento de um sistema de software multi-agentes para uma aplicação de comércio eletrônico.

Abstract

Agent-oriented development is quite recent, however, this new paradigm has been successfully used in industrial applications, such as telecommunications and e-commerce. Among the several concerns required for the consolidation of this new paradigm, we highlight the need for suitable techniques, notations and tools to support the agent-oriented software development. In this sense, the Tropos project is developing an approach for agent-oriented development centered into requirements aiming to support all phases of agent-oriented software development.

Tropos has defined a number of architectural styles focusing on both software organizational processes and non-functional requirements. These organizational architectural styles have been described initially using the i* notation. However, the use of i* as an architectural description language is not suitable, since it presents some limitations to describe the detailed behaviour required for architectural design.

Recognizing the *Unified Modeling Language* (UML) can be extended to act as an architectural description language, we present in this dissertation a set of extensions based on UML Real Time especially tailored for the representation of organizational architectural styles. This will enable software engineer to insert more detail to the system architectural design. In order to validate this proposal, a case study is presented using the Tropos framework for developing an agent-oriented software system tuned for an e-commerce application.

Conteúdo

Capítulo 1 - Introdução 1

1.1 Motivações 2
1.2 Estágio Atual da Engenharia de Software Orientada a Agentes 2
1.3 Escopo da Dissertação 4
1.4 Objetivos e Abordagem 5
1.5 Contribuição 6
1.6 Estrutura da dissertação 7

Capítulo 2 - Engenharia de Software Orientada a Agentes 9

2.1 Introdução 10
2.2 Características de Agentes 12
2.3 Agentes versus Objetos 16
2.4 Tipos de Sistemas Baseados em Agentes 20
2.5 Importância dos Sistemas Multi-Agentes na Engenharia de Software 22
2.6 Áreas de Aplicação 23
2.7 Desafios e Obstáculos do Desenvolvimento Orientado a Agentes 26
2.8 Considerações Finais 31

Capítulo 3 - Metodologias Orientadas a Agentes 33

3.1 Introdução 34
3.2 A metodologia GAIA 36
3.3 A metodologia AUML 38
3.3.1 O Comportamento Interno de um Agente 41
3.3.2 Sistemas Multi-Agentes como Estruturas Sociais 44
3.4 A metodologia MESSAGE/UML 45
3.5 A metodologia *MaSE* 48
3.6 A metodologia *TROPOS* 50
3.7 Considerações Finais 55

Capítulo 4 - Arquitetura de Software 58

4.1 Introdução 59
4.2 Definições de Arquitetura de Software 59
4.3 Conceitos Básicos 61
4.4 Projeto Arquitetural 62
4.4.1 Visões Arquiteturais 64
4.5 Estilos Arquiteturais 66
4.6 Documentação e Descrição da Arquitetura 68
4.6.1 Práticas Recomendadas 68
4.6.2 Linguagens para Descrição de Arquitetura 69
4.6.3 Representação Arquitetural em UML 70

4.7 Considerações Finais 75

Capítulo 5 - Estilos Arquiteturais Organizacionais em UML 77

5.1 Introdução 78
5.2 A Fase de Projeto Arquitetural da Metodologia Tropos 79
5.3 Mapeando i* para UML-RT 92
5.4 Estilos Arquiteturais Organizacionais em UML 96
5.5 Considerações Finais 100

Capítulo 6 - Estudo de Caso 102

6.1 Introdução 103
6.2 O Sistema de Comércio Eletrônico *Medi@* 103
6.2.1 Requisitos Iniciais 104
6.2.2 Requisitos Finais 106
6.2.3 Projeto Arquitetural 109
6.3 Considerações Finais 123

Capítulo 7 - Conclusões e Trabalhos Futuros 125

7.1 Contribuições do Trabalho 126
7.2 Trabalhos Futuros 128

Capítulo 8 - Referências Bibliográficas 129

Apêndice A 136

Apêndice B 162

Índice de Figuras

Figura 2.1 - Visão detalhada de um agente....13
Figura 2.2 - A visão de um agente como uma caixa-preta....14
Figura 2.3 - Comparação entre objetos e agentes....17
Figura 2.4 - Emergência....19
Figura 3.1 - Conceitos de Análise....36
Figura 3.2 - Relacionamentos entre os modelos Gaia....37
Figura 3.3 - Um Protocolo de Interação de Agente genérico expressado como um pacote modelo....40
Figura 3.4 - Estrutura interna de um agente....42
Figura 3.5 - Diagrama de classes para especificar a estrutura interna de um agente....43
Figura 3.6 - Ontologia Consolidada....45
Figura 3.7 - Conceitos MESSAGE....47
Figura 3.8 - Metodologia MaSE....49
Figura 3.9 - Cobertura das fases de desenvolvimento do software....56
Figura 4.1 - Um diagrama de classes de Cápsula....74
Figura 4.2 - Um diagrama de classes de Cápsula detalhado....74
Figura 4.3 - Um diagrama de colaboração de Cápsulas....75
Figura 5.1 - União Estratégica (Joint-Venture)....81
Figura 5.2 - Estrutura em 5 (Structure in 5)....84
Figura 5.3 - Integração Vertical (Vertical Integration)....85
Figura 5.4 - Apropriação (Co-optation)....86
Figura 5.5 - Comprimento de Braço (Arm's Length)....86
Figura 5.6 - Tomada de Controle (Takeover)....87
Figura 5.7 - Pirâmide (Pyramid)....88
Figura 5.8 - Contratação Hierárquica (Hierarchical Contracting)....89
Figura 5.9 - Oferta (Bidding)....90
Figura 5.10 - Estrutura Plana (Flat Structure)....91
Figura 5.11 - Mapeando uma dependência entre atores para UML....92
Figura 5.12 - Mapeando uma dependência de meta para UML....94
Figura 5.13 - Mapeando uma dependência de meta-soft para UML....94
Figura 5.14 - Mapeando uma dependência de recurso para UML....95
Figura 5.15 - Mapeando uma dependência de tarefa para UML....95
Figura 5.16 - Estilo União Estratégica em UML-RT....96

Figura 5.17 - Protocolos e portas que representam a dependência de meta Delegação de Autoridade do estilo União Estratégica 98

Figura 5.18 - Protocolos e portas que representam a dependência de meta-soft Valor Agregado do estilo União Estratégica....98

Figura 5.19 - Protocolos e portas que representam a dependência de tarefa Coordenação do estilo União Estratégica....99

Figura 5.20 - Protocolos e portas que representam a dependência de recurso Intercâmbio de Recursos do estilo União Estratégica 99

Figura 5.21 - Protocolos e portas que representam as demais dependências do estilo União Estratégica100

Figura 6.1 - Modelo i de Dependência Estratégica da Loja de Mídia*.... 105

Figura 6.2 - Modelo i de Razão Estratégica da Loja de Mídia*.... 105

Figura 6.3 - Modelo i de Dependência Estratégica do sistema Medi@*.... 107

Figura 6.4 - Modelo i de Razão Estratégica d o sistema Medi@*.... 108

Figura 6.5 - Grafo de Interdependência de Metas-soft.... 111

Figura 6.6 - A arquitetura União Estratégica do sistema Media@ 112

Figura 6.7 - Refinamento da cápsula Frente de Loja 113

Figura 6.8 - Refinamento da Cápsula Gerente Comum.... 114

Figura 6.9 - Refinamento da Cápsula Processador de Fatura.... 116

Figura 6.10 - Refinamento da Cápsula Retaguarda de Loja 116

Figura 6.11 - O refinamento da arquitetura do sistema Medi@.... 117

Figura 6.12 - Diagrama de seqüência para o contexto Fazer Pedido.... 118

Figura 6.13 - Diagrama de seqüência para o contexto Fazer Pedido de Item Não Disponível.... 119

Figura 6.14 - Diagrama de seqüência para o contexto Pesquisar Item. (a) Buscar item por palavra-chave. (b) Navegar no catálogo da Loja de Mídia.... 120

Figura 6.15 - Diagrama de seqüência refinando a Frente de Loja para o contexto Fazer Pedido.... 121

Figura 6.16 - Diagrama de seqüência refinando o Processador de Fatura para o contexto Fazer Pedido 122

Figura A.1 - Estilo Arquitetural Organizacional Leilão representado em UML-RT 136

Figura A.2 – Protocolo Serviço/Produto representando uma dependência de recurso entre as cápsulas Leiloeiro e Expositor.... 136

Figura A.3 - Protocolo MelhorOfertaPossivel representando uma dependência de meta-soft entre as cápsulas Expositor e Leiloeiro 137

Figura A.4 - Protocolo OfertaMaior representando uma dependência de meta-soft entre as cápsulas Leiloeiro e Comprador 2 137

Figura A.5 - Protocolo NenhumaOfertaMaior representando uma dependência de meta-soft entre as cápsulas Comprador n e Leiloeiro 137

Figura A.6 - Protocolo IniciarOfertaNoMenorPreço representando uma dependência de meta-soft entre as cápsulas Comprador 1 e Leiloeiro 138

Figura A.7 - Estilo Arquitetural Organizacional Apropriação representado em UML-RT.... 138

Figura A.8 - Protocolo ServicosExternos representando uma dependência de tarefa entre as cápsulas Contratador 1 e Apropriado 1.... 139

Figura A.9 - Protocolo RecursoExterno representando uma dependência de recurso entre as cápsulas Contratador n e Apropriado 1 .... *139*

Figura A.10 - Protocolo ProverCapital representando uma dependência de meta-soft entre as cápsulas Apropriado 2 e Contratador 1 .... *139*

Figura A.11 - Protocolo CompartilharConhecimento representando uma dependência de meta entre as cápsulas Contratador 1 e Contratador n *140*

Figura A.12 - Protocolo Suporte representando uma dependência de tarefa entre as cápsulas Apropriado n e Contratador n *140*

Figura A.13 - Estilo Arquitetural Organizacional Integração Vertical representado em UML-RT *141*

Figura A.14 - Protocolo ObterInstruções representando uma dependência de meta entre as cápsulas Provedor e Organizador *141*

Figura A.15 - Protocolo AcessoDiretoAoConsumidor representando uma dependência de meta entre as cápsulas Atacadista e Varejista *142*

Figura A.16 - Protocolo FornecimentoEmMassa representando uma dependência de meta entre as cápsulas Varejista e Atacadista *142*

Figura A.17 - Protocolo FornecerProdutos representando uma dependência de meta entre as cápsulas Consumidor e Varejista *143*

Figura A.18 - Protocolo EntregarProdutos representando uma dependência de meta entre as cápsulas Varejista e Organizador *143*

Figura A.19 - Protocolo DescobrirProdutos representando uma dependência de meta entre as cápsulas Consumidor e Organizador *143*

Figura A.20 - Protocolo AdquirirProdutos representando uma dependência de meta entre as cápsulas Consumidor e Organizador *144*

Figura A.21 - Protocolo ProdutosDeQualidade representando uma dependência de meta-soft entre as cápsulas Atacadista e Provedor *144*

Figura A.22 - Protocolo AcessoLargoAoMercado representando uma dependência de meta-soft entre as cápsulas Provedor e Atacadista *145*

Figura A.23 - Protocolo AvaliaçaoDoMercado representando uma dependência de meta-soft entre as cápsulas Atacadista e Organizador *145*

Figura A.24 - Protocolo InteresseNosProdutos representando uma dependência de meta-soft entre as cápsulas Varejista e Consumidor *145*

Figura A.25 - Estilo Arquitetural Organizacional Estrutura em Cinco representado em UML-RT. *146*

Figura A.26 - Protocolo GerênciaEstratégica representando uma dependência de meta-soft entre as cápsulas Agência Intermediária e Cúpula. *146*

Figura A.27 - Protocolo Controle representando uma dependência de meta entre as cápsulas Agência Intermediária e Coordenação. *147*

Figura A.28 - Protocolo Logística representando uma dependência de meta entre as cápsulas Agência Intermediária e Coordenação. *147*

Figura A.29 - Protocolo Padronizar representando uma dependência de tarefa entre as cápsulas Núcleo Operacional e Coordenação *147*

Figura A.30 - Protocolo Supervisionar representando uma dependência de tarefa entre as cápsulas Agência Intermediária e Coordenação *148*

Figura A.31 - Protocolo ServiçoNaoOperacional representando uma dependência de meta entre as cápsulas Núcleo Operacional e Suporte *148*

Figura A.32 - Estilo Arquitetural Organizacional Contratação Hierárquica representado em UML-RT . 149

Figura A.33 - Protocolo DadoBruto representando uma dependência de recurso entre as cápsulas Observador e Contratador 1 149

Figura A.34 - Protocolo Rotear representando uma dependência de tarefa entre as cápsulas Mediador e Contratador 2 150

Figura A.35 - Protocolo DadoBruto representando uma dependência de tarefa entre as cápsulas Contratador 3 e Mediador 150

Figura A.36 - Protocolo Coordenar representando uma dependência de tarefa entre as cápsulas Contratador n e Controlador 150

Figura A.37 - Protocolo Autoridade representando uma dependência de meta-soft entre as cápsulas Negociador e Executivo 151

Figura A.38 - Protocolo DecisoesEstrategicas representando uma dependência de meta-soft entre as cápsulas Ponderador e Executivo 151

Figura A.39 - Protocolo Compatibilizar representando uma dependência de meta entre as cápsulas Mediador e Negociador 152

Figura A.40 - Protocolo ResolverConflito representando uma dependência de meta entre as cápsulas Ponderador e Controlador 152

Figura A.4 1 - Estilo Arquitetural Organizacional Comprimento de Braço representado em UML-RT 153

Figura A.42 - Protocolo ManterAutonomia representando uma dependência de meta-soft entre as cápsulas Colaborador 2 e Colaborador 1 153

Figura A.43 - Protocolo ExaminarConhecimento representando uma dependência de meta-soft entre as cápsulas Colaborador n e Colaborador 2 154

Figura A.44 - Protocolo CapitalCompetitivo representando uma dependência de meta-soft entre as cápsulas Colaborador 1, Colaborador 2, Colaborador 3 e Colaborador n 154

Figura A.45 - Protocolo ObjetivosCorporativos representando uma dependência de meta-soft entre as cápsulas Colaborador 3 e Colaborador 1 154

Figura A.46 - Protocolo AvaliacaoDeDesempenho representando uma dependência de tarefa entre as cápsulas Colaborador n e Colaborador 3 154

Figura A.47 - Estilo Arquitetural Organizacional Tomada de Controle representado em UML-RT 155

Figura A.48 - Protocolo ManipularTarefas representando uma dependência de tarefa entre as cápsulas Tomador de Controle e Agência n 155

Figura A.49 - Protocolo Controle representando uma dependência de meta entre as cápsulas Agência 1 e Tomador de Controle 156

Figura A.50 - Protocolo Delegação de Autoridade representando uma dependência de meta entre as cápsulas Tomador de Controle e Agência 2 156

Figura A.51 - Protocolo FornecerRecursos representando uma dependência de recurso entre as cápsulas Tomador de Controle e Agência 3 156

Figura A.52 - Estilo Arquitetural Organizacional Pirâmide representado em UML-RT 157

Figura A.53 - Protocolo DelegarResponsabilidades representando uma dependência de meta entre as cápsulas Gerente e Cúpula 157

Figura A.54 - Protocolo AutoridadeEstrategica representando uma dependência de meta entre as cápsulas Supervisor e Cúpula 157

Figura A.55 - Protocolo RotearDelegacao representando uma dependência de tarefa entre as cápsulas Operador 1 e Gerente 158

Figura A.56 - Protocolo ResolverConflitos representando uma dependência de tarefa entre as cápsulas Operador 2 e Gerente *158*

Figura A.57 - Protocolo Coordenar representando uma dependência de tarefa entre as cápsulas Tomador Operador 3 e Supervisor *158*

Figura A.58 - Protocolo Monitorar representando uma dependência de tarefa entre as cápsulas Operador n e Supervisor *159*

Figura A.59 - Estilo Arquitetural Organizacional Estrutura Plana representado em UML-RT *159*

Figura A.60 - Protocolo ManterAutonomia representando uma dependência de meta entre as cápsulas Agência 1 e Agência 3 .... *160*

Figura A.61 - Protocolo CompartilharConhecimento representando uma dependência de meta-soft entre as cápsulas Agência 1 e Agência 2 .... *160*

Figura A.62 - Protocolo Suporte representando uma dependência de tarefa entre as cápsulas Agência 3 e Agência 2 .... *160*

Figura A.63 - Protocolo ManipularTarefas representando uma dependência de tarefa entre as cápsulas Agência n e Agência 2 .... *160*

Figura A.64 - Protocolo IntercambioDeRecursos representando uma dependência de recurso entre as cápsulas Agência 3 e Agência n *161*

Figura B.1 - Protocolo Carro representando uma dependência de recurso entre as cápsulas Processador de Fatura e Frente de Loja *162*

Figura B.2 - Protocolo Usabilidade representando uma dependência de meta-soft entre as cápsulas Frente de Loja e Gerente Comum *162*

Figura B.3 - Protocolo Conta representando uma dependência de recurso entre as cápsulas Retaguarda de Loja e Processador de Fatura *163*

Figura B.4 - Protocolo Confidencialidade representando uma dependência de meta-soft entre as cápsulas Processador de Fatura e Gerente Comum *163*

Figura B.5 - Protocolo Integridade representando uma dependência de meta-soft entre as cápsulas Frente de Loja e Gerente Comum *163*

Figura B.6 - Protocolo ConfirmarCompra representando uma dependência de tarefa entre as cápsulas Processador de Fatura e Frente de Loja *164*

Figura B.7 - Protocolo Item representando uma dependência de recurso entre as sub-cápsulas Carro de Compra e Catálogo On-line *164*

Figura B.8 - Protocolo ConsultarCatalogo representando uma dependência de tarefa entre as sub-cápsulas Navegador de Item e Catálogo On-line *164*

Figura B.9 - Protocolo Selecionar Item representando uma dependência de recurso entre as sub-cápsulas Carro de Compra e Navegador de Item *165*

Figura B.10 - Protocolo SolicitarPagamento representando uma dependência de tarefa entre as sub-cápsulas Processador de Pedido e Processador de Conta *165*

Figura B.11 - Protocolo ProcessarRecibo representando uma dependência de tarefa entre as sub-cápsulas Processador de Pedido e Processador de Recibo *165*

Figura B.12 - Protocolo Cliente representando uma dependência de recurso entre as sub-cápsulas Carro de Compra e Criador de Perfil *166*

Figura B.13 - Protocolo Atualização representando uma dependência de meta-soft entre as cápsulas Frente de Loja e Gerente Comum *166*

Figura B.14 - Protocolo Autorização representando uma dependência de meta-soft entre as cápsulas Processador de Fatura e Gerente Comum *166*

Figura B.15 - Protocolo Entrega representando uma dependência de recurso entre as cápsulas Retaguarda de Loja e Processador de Fatura *167*

Figura B.16 - Protocolo AvaliarSelec representando uma dependência de meta entre as cápsulas Retaguarda de Loja e Frente de Loja *167*

Figura B.17 - Protocolo Manutenibilidade representando uma dependência de meta-soft entre as cápsulas Frente de Loja e Gerente Comum *167*

Figura B.18 - Protocolo TempoResposta representando uma dependência de meta-soft entre as cápsulas Processador de Fatura e Gerente Comum *168*

Figura B.19 - Protocolo Observar representando uma dependência de meta-soft entre as sub-cápsulas Gerente de Disponibilidade, Controle de Segurança, Gerente de Adaptação e Monitor *168*

Índice de Tabelas

Tabela 6.1 - Catálogo de Correlação *110*

Capítulo 1 - Introdução

Este capítulo apresenta as principais motivações para realização desta dissertação. Em seguida, descreve o estágio atual na engenharia de software orientada a agentes. As seções seguintes apontam o escopo da dissertação, os objetivos e a abordagem utilizada, as principais contribuições e finalmente, a estrutura do trabalho.

1.1 Motivações

Atualmente, as aplicações industriais de software têm crescido no tamanho e na complexidade, fazendo com que seu projeto e construção se torne uma tarefa bastante difícil. Isso tem motivado os engenheiros de software a obterem um melhor entendimento das características de um software complexo, reconhecendo a interação como um de seus aspectos mais importantes. De fato, muitos pesquisadores buscam novos paradigmas para aumentar nossa habilidade de modelar, projetar e construir complexos sistemas de software de natureza distribuída [Wooldridge02].

Um dos mais promissores paradigmas do momento é a chamada orientação a agentes. Um agente é um sistema computacional capaz de ações flexíveis e autônomas em um ambiente dinâmico, aberto e imprevisível [Jennings03a]. A orientação a agente oferece um nível mais alto de abstração na forma de pensar sobre as características e os comportamentos dos sistemas de software.

Espera-se que sistemas orientados a agentes sejam mais poderosos, mais flexíveis e mais robustos do que os sistemas de software convencionais [Yu01]. É a naturalidade e a facilidade com que uma variedade de aplicações pode ser caracterizada em termos de agentes que levam pesquisadores e desenvolvedores a ficarem tão entusiasmados sobre o potencial dessa abordagem [Jennings03a].

1.2 Estágio Atual da Engenharia de Software Orientada a Agentes

Visando demonstrar a eficácia da abordagem orientada a agente, o ideal seria demonstrar quantitativamente que a adoção desse paradigma ajuda a melhorar, de acordo com alguns conjuntos padrões de métricas de software, o processo de desenvolvimento de software. Entretanto, devido à imaturidade da área, esses dados simplesmente não estão disponíveis (assim como também não há dados precisos para paradigmas mais estabelecidos, tal como a orientação a objetos). Mesmo assim, há argumentos qualitativos que nos faz acreditar que uma abordagem orientada a agentes será de muito benefício para a engenharia de certos sistemas de software complexos. Esses argumentos surgiram de uma década de experiência no uso da tecnologia de agente para construir aplicações do mundo-real em larga-escala e

em uma grande variedade de domínios industriais e comerciais [Agentlink03]. Note que não estamos sugerindo que o desenvolvimento baseado em agentes é uma solução mágica (*Silver Bullet*), pois não há nenhuma evidência que venha a sugerir que eles representarão uma melhoria em ordem de magnitude na produtividade da engenharia de software. Entretanto, acredita-se que para certas classes de aplicação, uma abordagem orientada a agentes pode melhorar significantemente o processo de desenvolvimento de software [Jennings03a].

No momento, existem algumas barreiras para que haja uma maior aceitação da tecnologia de agente. É necessário estabelecer melhor a relação da tecnologia de agente com a tecnologia antecedente (tecnologia orientada a objetos), apresentando essa nova tecnologia como uma extensão incremental de métodos conhecidos e confiáveis [Odell00b]. Também é preciso entender as situações em que soluções baseadas em agentes são apropriadas [Jennings03a]. Contudo, uma das maiores lacunas é a ausência de metodologias sistemáticas que possibilitem aos projetistas especificar e estruturar claramente suas aplicações como sistemas orientados a agentes. Ainda há uma ausência de ferramentas disponíveis na indústria para suportar o desenvolvimento de sistemas orientados a agentes. De fato, metodologias preliminares e ferramentas de software para ajudar a difundir sistemas orientados a agentes estão começando a ser propostas [Wooldridge02]. Acredita-se firmemente que a tecnologia de agente tenha potencial para se tornar uma solução de engenharia de software popular (da mesma maneira que a tecnologia orientada a objeto se tornou).

Entre os desafios atuais, nesta dissertação estaremos abordando a necessidade de se ter um meio sistemático de analisar o problema, exercitar de que forma ele pode ser mais bem estruturado como um sistema orientado a agentes e então determinar quais agentes individuais devem ser desenvolvidos e integrados [Jennings03a]. É natural que a proposta de metodologias de desenvolvimento para a construção de sistemas orientados a agentes esteja se tornando uma área promissora na pesquisa da engenharia de software.

Até agora, técnicas de desenvolvimento de software tradicionalmente têm sido direcionadas à implementação no sentido de que o paradigma de programação dita as técnicas usadas agora no projeto e na análise de requisitos [Odell00b]. Surgem as primeiras metodologias

voltadas especificamente para o desenvolvimento orientado a agentes oriundas da inteligência artificial [Caire01][Wooldridge00][Odell00b][DeLoach01]. Algumas adaptam engenharia do conhecimento e outras técnicas. Poucas são orientadas a requisitos no sentido de que a mesma seja baseada nos conceitos usados durante as fases iniciais do processo de desenvolvimento, conforme advogado pela engenharia de requisitos [Castro02].

Outro impedimento para que haja uma maior aceitação da tecnologia de agente é de cunho social. Tanto os indivíduos quanto as organizações precisarão se tornar mais acostumados e confiantes com a noção de agentes de software autônomos que possam trabalhar em seu benefício [Jennings00]. Essa dissertação é um passo nesta direção.

1.3 Escopo da Dissertação

Esta dissertação tem o foco na carência de notações, ferramentas e metodologias específicas para o desenvolvimento de software orientado a agentes. De fato, buscamos a padronização de linguagens para modelar sistemas orientados a agentes, já que, como visto na seção anterior, para que o paradigma orientado a agentes se torne uma tecnologia popular é necessário que ele seja bem servido de ferramentas e metodologias sistemáticas de desenvolvimento.

Em particular enfocaremos a proposta Tropos, que propõe uma metodologia de desenvolvimento de software e um framework de desenvolvimento que são baseados em conceitos usados para modelar requisitos iniciais. Tropos suporta cinco fases do desenvolvimento de software: Requisitos Iniciais, Requisitos Finais, Projeto Arquitetural, Projeto Detalhado e Implementação.

É importante salientar que Tropos utiliza a notação i* [Yu95] para representar seus modelos tanto na fase de requisitos quanto na fase arquitetural. No entanto, essa notação não é largamente aceita pelos profissionais de software, já que ela é recente e só agora começa a ser reconhecida como sendo adequada para representar requisitos. Também crítico para a sua adoção é o seu limitado suporte ferramental. Além disso, ela não é adequada para representar algumas informações detalhadas que algumas vezes são

requeridas no projeto arquitetural, tal como o conjunto de sinais que são trocados entre os componentes arquiteturais, bem como a seqüência válida desses sinais (protocolo).

Por outro lado, a *Unified Modeling Language* – UML [Rumbaugh99] vem sendo proposta para representação da arquitetura de sistemas simples e complexos. Como uma linguagem de descrição arquitetural, a UML pode prover meios de representar decisões de projeto. Isso pode levar a modelos arquiteturais onde descrevemos elementos de projeto de alto-nível do sistema e seus conectores, suportando diferentes pontos de vista do sistema em construção. Além disso, ela é suportada por um grande número de provedores de ferramentas.

Neste contexto, enfatizaremos a fase de projeto arquitetural da metodologia Tropos, que definiu estilos arquiteturais organizacionais [Kolp01a][Kolp01b][Kolp02] baseados em conceitos e alternativas de projeto vindos da pesquisa em gerência da organização, usadas para modelar coordenação de *stakeholders*[1] de negócio, indivíduos, físicos ou sistemas sociais. Desta perspectiva, o sistema de software é como uma organização social de componentes autônomos coordenados que interagem a fim de atingir metas específicas e possivelmente comuns.

1.4 Objetivos e Abordagem

Neste trabalho propomos o uso de uma extensão da UML para acomodar os conceitos e características atualmente usadas para representar arquiteturas organizacionais em Tropos. Tal extensão, a UML Real-Time [Selic98][Selic99], é destinada para sistemas em tempo real e está sendo usada para modelar arquitetura de software.

Esta pesquisa é motivada pelos seguintes fatores:

- A Engenharia de Software Orientada a Agentes, por ser ainda recente, é desprovida de notações, ferramentas e metodologias de desenvolvimento;

[1] Stakeholders: são pessoas ou organizações que serão afetadas pelo sistema e tem influência, direta ou indireta, sobre os requisitos do sistema – usuários finais, gerentes e outros envolvidos no processo organizacional influenciados pelo sistema, engenheiros responsáveis pelo desenvolvimento e manutenção do sistema, clientes da organização que usará o sistema para fornecer algum serviço, etc [Kotonya97].

- Agentes podem ser considerados como uma forma de evolução dos objetos;

- A UML é bem servida de ferramentas CASE, sendo bastante popular na representação de artefatos de engenharia em software orientado a objetos. Também provê um conjunto rico e extensível de elementos de construção;

- A UML, através de extensões tais como a UML-RT, pode ser usada para representar a arquitetura de sistemas tanto simples como complexos.

- A metodologia de desenvolvimento orientada a agentes TROPOS tem demonstrado a necessidade de detalhar melhor a sua fase arquitetural.

Para validar esta proposta, será realizado um estudo de caso onde a metodologia Tropos, utilizando a UML-RT na fase de projeto arquitetural, será aplicada no desenvolvimento de um sistema de software orientado a agentes na área de comércio eletrônico.

1.5 Contribuição

Esta dissertação fornece várias contribuições para a área de Engenharia de Software Orientado a Agentes, em particular elas estão diretamente relacionadas com a fase de projeto arquitetural da metodologia Tropos. As contribuições mais significativas são:

- Usar os estilos arquiteturais organizacionais definidos pelo Tropos nos permitirá construir arquiteturas mais flexíveis, com componentes fracamente acoplados, que podem evoluir e mudar continuamente para acomodar novos requisitos, permitindo a solicitação de sistemas complexos mais flexíveis.

- Usar estilos arquiteturais organizacionais, descritos mais precisamente em UML, nos permitirá representar informação tais como os sinais de comunicação trocados pelos componentes que compõem a arquitetura. Observe que este tipo de informação não é permitido atualmente na notação adotada no Tropos (baseada somente em i*).

- Detalhar a descrição dos catálogos arquiteturais possibilitará uma maior facilidade de adoção da proposta Tropos por aqueles que são familiarizados com a UML.

Como benefícios adicionais em usar UML para modelar também a arquitetura detalhada em Tropos, podemos destacar [Medvidovic00]:

- Representação Comum de Modelos: Armazenar no mesmo repositório de modelos de informação de tipos diferentes de visão (UML e não-UML).
- Conjunto de ferramentas reduzido para manipulação de modelos: Estar apto a usar elementos UML para representar artefatos não-UML nos permite usar conjuntos de ferramentas UML existentes para criar tais visões.
- Forma unificada de cruzamento referenciado de informação de modelo: Possuir informação de modelo armazenada em uma única localização física nos permite ainda cruzamento referenciado desta informação. Cruzamento referenciado é útil para manter o rastreamento entre artefatos das fases de projeto detalhado e projeto arquitetural no Tropos.

1.6 Estrutura da dissertação

Além deste capítulo introdutório, esse trabalho consiste de mais seis capítulos, são eles:

Capítulo 2 – Engenharia de Software Orientada a Agentes

Nesse capítulo são mostradas as principais motivações para usar a abordagem de agentes, assim como apresenta algumas definições e conceitos relacionados com esta tecnologia. Em seguida fazemos uma comparação entre objetos e agentes, mostramos a importância de sistemas orientados a agentes, as armadilhas da Engenharia de Software Orientada a Agentes e os obstáculos que esta nova área vem sofrendo.

Capítulo 3 – Metodologias Orientadas a Agentes

Nesse capítulo são mostradas as principais metodologias atualmente em desenvolvimento para suportar a ESOA, bem como algumas propostas em explorar a UML para desenvolver sistemas orientados a agentes.

Capítulo 4 – Arquitetura de Software

Nesse capítulo é fornecida uma visão geral da arquitetura de software como a descrição de um sistema, que é o resultado da soma de partes menores e da forma como estas partes se relacionam e cooperam para realizar as tarefas do sistema.

Capítulo 5 – Modelando Estilos Arquiteturais Organizacionais em UML

Nesse capítulo é apresentada a principal contribuição desse trabalho, o uso da UML para acomodar os conceitos e características usadas para representar arquiteturas organizacionais em Tropos, de modo a adicionar mais detalhes aos seus modelos arquiteturais.

Capítulo 6 – Estudo de Caso

Nesse capítulo é mostrado um estudo de caso real com o objetivo de validar a nossa proposta. Será realizado o desenvolvimento de um sistema de software orientado a agentes para uma aplicação e-commerce. Nesse contexto, será avaliada a adequação da UML como notação para a fase de projeto arquitetural da metodologia Tropos.

Capítulo 7 – Conclusões e Trabalhos Futuros

Nesse capítulo são mostradas as conclusões obtidas durante o desenvolvimento desse trabalho, assim como as principais contribuições que ele fornece para Engenharia de Software Orientada a Agentes. Serão mostrados alguns possíveis trabalhos futuros. Assim como será feito um direcionamento para futuras pesquisas nesta área.

Capítulo 2 - Engenharia de Software Orientada a Agentes

Este capítulo aborda a engenharia de software orientada a agentes. Inicialmente mostraremos as motivações para o uso desse novo paradigma, seguido dos aspectos e propriedades que caracterizam um agente. Também abordaremos as motivações e a importância dos sistemas baseados em agentes para a engenharia de software. Além disso, apresentaremos as principais áreas de aplicação dos softwares orientados a agentes. Finalmente, apontamos os desafios e obstáculos enfrentados atualmente no desenvolvimento de sistemas orientados a agentes

2.1 Introdução

Projetar e construir software industrial de alta qualidade é difícil. De fato, assume-se que a tarefa de desenvolver tais projetos está entre as mais complexas tarefas de construção realizadas por humanos. Diante disso, vários paradigmas de implementação foram propostos na literatura (ex., programação estruturada, programação declarativa, programação orientada a objetos e baseada em componentes). Esta evolução incremental dos paradigmas visa tornar a engenharia de software mais gerenciável bem como permite aumentar a complexidade das aplicações que podem ser construídas.

As empresas de desenvolvimento de sistemas de grande porte vêm apresentando grandes problemas e desejam produzir software de qualidade. Na visão do usuário, software de qualidade é aquele que além de satisfazer as suas necessidades, é feito no custo e prazo combinados. Já na visão do projetista de software, um bom software é aquele que apresenta as seguintes características:

- Maior flexibilidade - Possibilita satisfazer novos requisitos de negócio (funcionalidade) de forma fácil e rápida.
- Melhor adaptabilidade - Possibilita personalizar uma aplicação para vários usuários, usando várias alternativas para oferecer os serviços da aplicação com o mínimo de impacto no seu núcleo.
- Melhor manutenibilidade - Possibilita alterar partes de uma aplicação, de modo que as outras partes sofram um impacto mínimo.
- Melhor reusabilidade - Possibilita montar aplicações únicas e dinâmicas rapidamente.
- Melhor aproveitamento do legado - Possibilita reusar a funcionalidade de sistemas legados em novas aplicações.
- Melhor interoperabilidade - Possibilita que duas aplicações que executam em plataformas diferentes troquem informações.

- Melhor escalabilidade - Possibilita distribuir e configurar a execução da aplicação de modo a satisfazer a vários volumes de transação.

- Melhor robustez - Possibilita implementar soluções de software com menos defeitos.

Além disso, o projetista de software considera um bom processo de desenvolvimento de software aquele que apresenta as seguintes características:

- Menor tempo de desenvolvimento – Possibilita construir novos sistemas de forma mais rápida e com baixo orçamento.

- Menor risco - Possibilita todas as características citadas para um software de qualidade, sem ter o risco de ter projetos fracassados.

Resumindo, as empresas de grande porte desejam software de qualidade [ISO96], isto é: Funcionalidade, Manutenibilidade, Usabilidade, Eficiência, Confiabilidade, Portabilidade, etc.

Embora a orientação a objetos esteja bastante popular no mundo do desenvolvimento de software atualmente, ela não está resolvendo os problemas apresentados pelas grandes empresas de desenvolvimento de sistemas. De fato, muitos projetos de grande porte que são baseados em objetos estão fracassando recentemente. Aparentemente, a OO não possui todo o aparato necessário para desenvolver software com as características citadas anteriormente, já que ela é apenas uma tecnologia habilitadora (*Enabling Technology)*. Isto significa dizer que os benefícios oferecidos pela orientação a objetos não são automáticos, já que dependem do uso correto da tecnologia. Além disso, muitas extensões a orientação a objetos ainda precisam ser feitas para que as características apontadas para um software de qualidade venham a ser realizadas com menos esforço pelos engenheiros de software.

Portanto, nos últimos anos os pesquisadores têm averiguado novas abordagens, tal como o paradigma de agentes, de forma a melhorar significantemente o processo de desenvolvimento de software. Técnicas orientadas a agentes representam um novo meio de analisar, projetar e construir sistemas de software complexos. O uso da abordagem

orientada a agentes é baseado nos seguintes argumentos: (i) o aparato conceitual de sistemas orientados a agentes é adequado para construir soluções de software para sistemas complexos e (ii) as abordagens orientadas a agentes representam um verdadeiro avanço sobre o atual estado da arte na engenharia de sistemas complexos [Jennings03a].

De fato, técnicas orientadas a agentes são adequadas para desenvolver sistemas complexos de software porque:

- As decomposições orientadas a agente são uma maneira efetiva de repartir o espaço do problema de um sistema complexo;
- As abstrações chave presentes no modo de pensar orientado a agentes são um meio natural de modelar sistemas complexos;
- A filosofia orientada a agente para identificar e gerenciar relacionamentos organizacionais é apropriada para lidar com as dependências e interações que existem em um sistema complexo.

Portanto, é necessário primeiro caracterizar o que são agentes. Na próxima seção, apresentaremos os principais conceitos relacionados a um agente.

2.2 Características de Agentes

No momento, há um contínuo debate sobre o que exatamente constitui um agente, não havendo ainda uma definição padrão. No entanto, encontra-se na literatura várias definições para o termo agente, que incluem:

> "Um agente é aquele que age, exerce um poder ou produz um efeito [Bueno90]".
>
> "Internamente, um agente é descrito por uma função f, que captura percepções e recebe mensagens como entrada, gerando saídas na forma de execução de ações e envio de mensagens. O mapeamento f em si não é diretamente controlado por uma autoridade externa [Müller96]".

> "Um agente é aquele que possui um modelo simbólico do mundo explicitamente representado e cujas decisões (por exemplo, sobre qual ação executar) são tomadas via raciocínio simbólico [Wooldridge95b]".

> "Internamente, um agente é um componente computacional com qualidades mentais atribuídas, que incluem crença, desejo, meta, intenção e compromisso. O agente de computação toma decisões de resolução, planejamento e execução baseadas na manipulação simbólica destas qualidades mentais, a fim de se comprometer com suas obrigações para realizar uma tarefa final [Li00]".

> "Agentes são unidades de software que podem lidar com mudanças ambientais e com os vários requisitos de redes abertas através de características como autonomia, mobilidade, inteligência, cooperação e reatividade [Tahara99]"

> "Um agente é um sistema que possui autonomia, reatividade, pró-atividade e habilidade social [Wooldridge95a]"

Nesta dissertação direcionamos a conceituação do que seriam agentes de software para a seguinte definição [Wooldridge01a]:

> "Um agente é um sistema computacional encapsulado que está situado em algum ambiente e é capaz de ação flexível autônoma neste ambiente, a fim de alcançar seus objetivos de projeto"

Na figura a seguir temos a proposta por [Flake01] que detalha o interior de um agente:

Figura 2.1 - Visão detalhada de um agente

Para os projetistas de sistemas de informação, agentes precisam de propriedades adicionais, que podem ser possuídas em várias combinações. São elas:

- Autonomia – capaz de agir sem intervenção externa direta. O agente tem algum grau de controle sobre seu estado interno e suas ações são baseadas em suas próprias experiências.
- Interatividade – capaz de se comunicar com o ambiente e com outros agentes (Figura 2.2).

Figura 2.2 - A visão de um agente como uma caixa-preta

- Adaptatividade – capaz de responder a outros agentes e/ou a seu ambiente em alguma proporção. As formas mais avançadas de adaptação permitem que um agente modifique seu comportamento baseado em sua experiência.
- Sociabilidade – capaz de interagir de forma amistosa ou prazerosa.
- Mobilidade – capaz de se transportar de um ambiente para outro.
- Representatividade – capaz de agir em benefício de alguém ou algo, isto é, age em interesse, como um representante, ou em benefício de alguma entidade.
- Pró-atividade – capaz de orientar-se à meta, ter propósito. O agente não reage simplesmente ao ambiente.

- Inteligência – capaz de possuir o estado formalizado por conhecimento (isto é, crenças, metas, planos, afirmações). A interação com os outros agentes é através de linguagem simbólica.
- Racionalidade – capaz de escolher uma ação baseando-se em metas internas e no conhecimento de que uma ação particular o deixará mais próximo de suas metas.
- Imprevisibilidade – capaz de agir de formas não completamente previsíveis, mesmo se todas as condições iniciais são conhecidas. Capacidade de comportamento não-determinístico.
- Continuidade Temporal – capaz de ser um processo que executa continuamente.
- Caráter – capaz de possuir personalidade e estado emocional críveis.
- Transparência e responsabilidade – capaz de ser transparente quando necessário e ainda prover um registro das atividades sob demanda.
- Coordenação – capaz de executar alguma atividade em um ambiente compartilhado por outros agentes. As atividades são freqüentemente coordenadas através de planos, fluxos de trabalho ou algum outro mecanismo de gerência de processo.
- Cooperação – capaz de cooperar com outros agentes a fim de atingir um objetivo comum; são agentes não adversários que obtêm sucesso ou falham juntos. (Colaboração é outro termo usado como sinônimo de cooperação)
- Competição – capaz de coordenar com outros agentes exceto no caso em que o sucesso de um agente implica na falha de outros (oposto de cooperativo).
- Robustez – capaz de lidar com erros e dados incompletos de forma robusta.
- Confiabilidade – capaz de ser confiável.

Grande parte da comunidade orientada a agentes concorda que o agente, para sistemas de informação, não é útil sem pelo menos as três primeiras propriedades acima, isto é, autonomia, interatividade e adaptatividade. Outros requerem que o agente possua todas as

propriedades listadas acima, porém em proporções variadas [Odell00a]. Uma classificação comum de agentes é a noção fraca e forte de agência. Na noção fraca de agência, os agentes têm sua própria vontade (autonomia), são capazes de interagir uns com os outros (habilidade social), respondem a estímulos (reatividade) e tomam iniciativa (pró-atividade). Na noção forte de agência, as noções fracas de agência são preservadas com a adição de que agentes podem se mover de um ambiente para outro (mobilidade), são confiáveis (veracidade), fazem o que é dito para ser feito (benevolência) e operam de maneira ótima para atingir metas (racionalidade) [Tveit01].

É importante sermos capazes de comparar esse novo paradigma com o paradigma antecedente, isto é, a orientação a objetos. Na próxima seção, apresentaremos os aspectos comparáveis entre agentes e objetos, destacando suas semelhanças e diferenças.

2.3 Agentes versus Objetos

Sabemos que objetos estão sendo usados com sucesso como abstrações para entidades passivas no mundo real (por exemplo, uma casa), mas agentes são considerados como um possível sucessor de objetos, já que eles podem melhorar as abstrações de entidades ativas [Tveit01].

Agentes e objetos são comparáveis em vários aspectos. De fato, há semelhanças óbvias entre agentes e objetos, porém também existem diferenças significantes entre eles [Wooldridge99].

Tanto objetos quanto agentes têm identidade, estado e comportamento próprios, além de interfaces através das quais eles podem se comunicar entre si e com outras entidades. Ambos podem ser de qualquer tamanho, entretanto diretrizes de desenvolvimento freqüentemente sugerem a construção de agentes e objetos pequenos [Odell00a].

Não é necessário – de fato, não é factível – que um agente conheça tudo. Ao invés de ser onisciente e onipotente, sistemas grandes baseados em agentes percebem e agem localmente. Analogamente, um objeto geralmente só interage com aqueles objetos aos quais ele está ligado [Odell99].

Contudo, objetos e agentes diferem em vários aspectos (Figura 2.3). Por exemplo, agentes ao invés de invocar métodos uns sobre os outros, fazem requisições (através de atos de comunicação) para que ações sejam executadas. Além disso, o foco de controle com respeito à decisão sobre se uma ação é executada ou não, difere em sistemas baseados em agentes. Na orientação a objetos, a decisão sobre a execução de uma ação está com o objeto que invoca o método. Já na orientação a agentes, esta decisão se encontra com o agente que recebe a requisição para executar a ação, pois é baseada no seu estado interno. Esta distinção entre objetos e agentes foi quase resumida no seguinte slogan: Objetos fazem de graça; agentes fazem por interesse [Odell99].

O conhecimento de um agente pode ser representado de uma maneira que não é facilmente traduzida em um conjunto de atributos, como ocorre em objetos (Figura 2.3). Além disso, mesmo se o estado de um agente estiver disponível publicamente, pode ser difícil decifrá-lo ou até mesmo entendê-lo [Odell99].

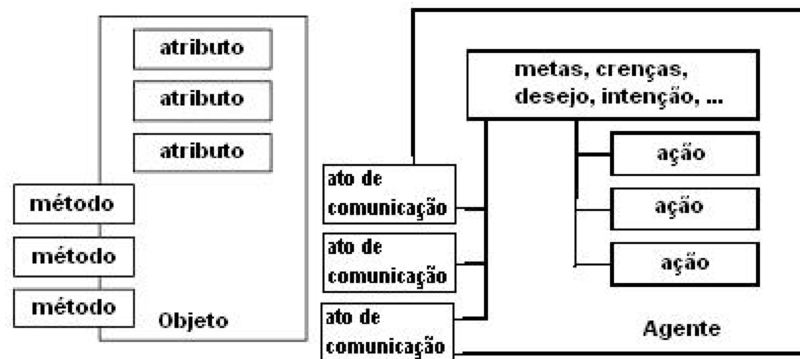


Figura 2.3 - Comparação entre objetos e agentes

As linguagens orientadas a objetos atuais não permitem que um objeto "anuncie" suas interfaces. Em contraste, um agente pode empregar mecanismos para publicar seus serviços. Outro mecanismo provê agentes representantes especializados, através dos quais outros agentes se tornam conhecidos para vários propósitos, mas desconhecidos para o resto da população de agente.

O modelo de comunicação de agente é geralmente assíncrono. Isso significa que não há fluxo predefinido de controle de um agente para outro. Conseqüentemente, as linguagens

orientadas a agente têm de suportar processamento paralelo. Já as linguagens OO proeminentes não suportam diretamente essa característica. Portanto, compartilhamos a visão de que o paradigma de objeto nos forçou a repensar sobre as formas apropriadas de interação (métodos de acesso ao invés de manipulação/introspecção direta), enquanto os agentes nos forçaram a encarar as implicações temporais da interação (mensagens ricas ao invés de invocação remota de método) [Odell99].

Agentes podem reagir não só a invocações de método específicas, mas também a eventos observáveis no ambiente. Considera-se que agentes estejam continuamente ativos e tipicamente envolvidos em um *loop* infinito de observação do seu ambiente. Dessa forma, eles podem atualizar o seu estado interno e selecionar uma ação para executar [Wooldridge02]. Um sistema multi-agentes é inerentemente *multi-threaded*, onde se assume que cada agente tem pelo menos uma linha (*thread*) de controle. Em contraste, objetos estão quietos a maioria do tempo e se tornando ativos apenas quando seus serviços são requisitados. Essa requisição é feita por meio de invocação de seus métodos sob a linha de controle do objeto que invocou [Odell99].

Agentes utilizam uma poderosa linguagem para se comunicar uns com os outros. Eles podem representar informação complexa de crença-desejo-intenção, usar inferência e mudar seu comportamento baseado no que eles aprenderam. Já objetos, usam um conjunto fixo de mensagens na comunicação. Conjunto este definido pelas múltiplas interfaces que o objeto pode suportar [Odell00a].

Na orientação a objetos, objetos são criados por uma classe e, uma vez criados, nunca devem mudar suas classes ou se tornar instâncias de classes múltiplas (exceto por herança). No caso de agentes, a abordagem é mais flexível. Pelo menos em muitos sistemas baseados em agentes, os agentes correspondem aproximadamente à instância de objetos ou indivíduos, porém não há uma noção de classe fortemente tipada. Assumindo um papel ou outro, o agente ainda é a mesma entidade, porém ele possui um conjunto de características diferentes. Além disso, um agente pode atuar papéis diferentes em domínios diferentes. As linguagens orientadas a objeto não suportam diretamente esses mecanismos de dependência de domínio que são necessários para ambientes baseados em agentes. A abordagem orientada a objeto de classe única é eficiente e confiável, mas a abordagem dinâmica e

múltipla provê flexibilidade e modelos mais próximos aos muitos modelos do nosso mundo.

Sabemos que as características possuídas por um objeto são definidas por sua classe. Um agente, além de possuir características definidas por sua classe, ainda pode adquirir ou modificar suas próprias características. Nesse caso, se um agente tem a habilidade de aprender, ele pode mudar seu próprio comportamento dinamicamente e agir diferentemente de qualquer outro agente.

Sistemas baseados em agentes podem suportar diretamente conceitos como regras, restrições, metas, crenças, desejos e responsabilidades. Embora sistemas baseados em objetos sejam construídos para incluir estes conceitos (particularmente as regras IF-THEN de sistemas peritos), eles não são diretamente suportados pela orientação a objetos tradicional.

A interação de muitos agentes individuais pode dar abertura a um efeito secundário, no qual um grupo de agentes se comporta como uma entidade única. Esse fenômeno chama-se emergência (Figura 2.4) e caracteriza-se por produzir um comportamento maior do que a soma dos comportamentos dos agentes individuais. Exemplos desse fenômeno podem ser vistos em sociedades encontradas na natureza tais como um bando de patos ou um cordão de formigas. Agentes só precisam possuir poucas regras simples para produzir a emergência, ao contrario de objetos tradicionais que, por não interagirem sem um *thread* de controle de mais alto nível, não correm o risco de provocar este fenômeno [Odell99].

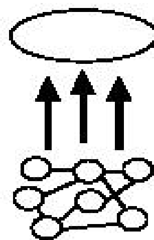


Figura 2.4 - Emergência

Embora não haja nenhum modelo interno que caracterize todos os agentes, considera-se que os agentes (inteligentes) são mais funcionais (espertos) do que objetos comuns. Esses

agentes incluem componentes de raciocínio complexo, tais como motores de inferência ou até mesmo redes neurais.

Os advogados do paradigma de objeto podem sugerir que objetos fazem coisas que agentes comumente não fazem, como por exemplo, herdam comportamento. Essa não é comumente vista como um aspecto de agente, embora agentes não sejam impossibilitados de suportar algumas formas de herança [Odell00a].

Em sistemas com grande numero de agentes ou objetos, cada um pode ser pequeno quando comparado ao sistema inteiro. Entretanto, um agente individual pode ter menos impacto em um sistema do que um objeto. O comportamento do sistema multi-agentes tende a ficar estável, desprezando as variações de performance ou a morte de qualquer agente. Já na orientação a objetos se um objeto é perdido em um sistema, uma exceção é levantada [Odell99].

Nessa seção vimos que agentes podem ser definidos como uma extensão de objetos. Um objeto é algo que encapsula sua identidade (quem), seu estado (o que) e seu comportamento (como). Um objeto ativo encapsula sua própria linha de controle (quando). Já um agente possui todas estas características e adiciona algo a mais. Um agente também encapsula o porque que ele faz algo, ou seja, sua motivação. Na próxima seção apresentamos os tipos de sistemas baseados em agentes encontrados hoje na engenharia de software orientada a agentes.

2.4 Tipos de Sistemas Baseados em Agentes

Um sistema baseado em agentes pode conter um ou mais agentes. Existem casos em que um único agente é apropriado. Um bom exemplo é a classe de sistemas conhecida como assistentes especialistas, no qual um agente age como um especialista que auxilia um usuário de computador na execução alguma tarefa. Entretanto, quando adotamos uma visão orientada a agentes do mundo, se torna claro que um único agente é insuficiente. A maioria dos problemas requer ou envolve agentes múltiplos [Jennings00]. O caso multi-agentes – onde o sistema é projetado e implementado com vários agentes interagindo – é mais geral e mais interessante do ponto de vista da engenharia de software. Os sistemas multi-agentes

são idealmente apropriados para representar problemas que têm múltiplos métodos de resolução, múltiplas perspectivas e/ou múltiplas entidades de resolução. Tais sistemas têm as vantagens tradicionais da resolução de problemas concorrentes e distribuídos, mas incluem as vantagens dos padrões de interações sofisticados. Exemplos comuns de tipos de interação incluem: cooperação (trabalhar junto para um fim comum); coordenação (organizar a atividade de resolução do problema de forma que interações danosas sejam evitadas ou interações beneficentes sejam exploradas); e negociação (vindo de um acordo que é aceitável para todas as partes envolvidas).

A pesquisa em sistemas multi-agentes se preocupa com o comportamento de uma coleção de agentes autônomos, possivelmente pré-existentes, visando resolver um dado problema. Um sistema multi-agentes pode ser definido como uma rede fracamente acoplada de solucionadores de problema que trabalham juntos para solucionar problemas que estão além de suas capacidades ou conhecimentos individuais. Estes solucionadores de problemas – agentes - são autônomos e podem ser heterogêneos por natureza. As características de um sistema multi-agentes são [Jennings00]:

- Cada agente tem informações ou capacidades incompletas para solucionar um dado problema. Dessa forma, cada agente tem um ponto de vista limitado;
- Não há controle global no sistema;
- O dado é descentralizado; e
- A computação é assíncrona.

Na maioria dos casos, agentes agem para atingir objetivos em benefício de indivíduos ou de empresas. Assim, quando agentes interagem, há tipicamente algum contexto organizacional em destaque. Este contexto ajuda a definir a natureza dos relacionamentos entre os agentes.

Para suportar sistemas multi-agentes, um ambiente apropriado precisa ser estabelecido, devendo [Odell00a]:

- prover uma infra-estrutura de comunicação específica e protocolos de interação;

- ser tipicamente aberto e não possuir um projetista centralizado ou uma função de controle *top-down*;
- Conter agentes que sejam autônomos, adaptativos e coordenativos.

Na próxima seção destacaremos a importância e a motivação dos sistemas multi-agentes.

2.5 Importância dos Sistemas Multi-Agentes na Engenharia de Software

Uma questão óbvia a perguntar é por que sistemas multi-agentes são vistos como uma nova direção importante na engenharia de software. A resposta é clara [Odell99]:

Agentes são importantes/úteis porque:

- provêm uma maneira de pensar sobre o fluxo de controle em um sistema altamente distribuído,
- oferecem um mecanismo que permitem um comportamento emergente ao invés de uma arquitetura estática,
- codificam melhores práticas de como organizar entidades colaborativas concorrentes.

Existem várias razões que justificam o aumento de interesse na pesquisa de sistemas multi-agentes [Wooldridge01a]:

- Distribuição de dado e controle - Para muitos sistemas de software o controle global é distribuído em vários nós de computação freqüentemente distribuídos geograficamente. A fim de fazer com que esses sistemas trabalharem efetivamente, esses nós devem ser capazes de interagir autonomamente uns com os outros, ou seja, eles devem ser agentes.
- Sistemas legados - Uma maneira natural de incorporar sistemas legados em modernos sistemas de informação distribuídos é *"agenticiando-os"*, isto é, envolvendo-os em uma camada de agente, que permitirá que eles interajam com outros agentes.

- Sistemas abertos - Muitos sistemas são abertos no sentido de que não é possível saber, em tempo de projeto, quais componentes compreenderão exatamente o sistema, nem como esses componentes interagem entre si. Para operarem efetivamente, estes sistemas devem ser capazes de tomar decisão autônoma flexível, ou seja, devem ser compostos por agentes.

Uma outra razão inclui a habilidade de prover robustez e eficiência aos sistemas de software [Odell00a]:

- Um agente poderia ser construído para fazer tudo (onipotente), mas agentes complexos representam um gargalo para velocidade, confiabilidade, manutenibilidade, etc. Dividir funcionalidade entre muitos agentes permite que a aplicação seja modular, flexível, modificável e extensível.

- Conhecimento especializado não é freqüentemente disponível em um único agente (onisciência). O conhecimento que é distribuído em várias fontes (agentes) pode ser integrado para uma maior completude quando for necessário.

Nesse contexto, sistemas orientados a agentes estão sendo cada vez mais usados na indústria como, por exemplo, em aplicações voltadas para telecomunicações e comércio eletrônico. Na próxima seção abordaremos as principais áreas onde os sistemas orientados a agentes estão presentes atualmente.

2.6 Áreas de Aplicação

A tecnologia de agente está ultrapassando rapidamente as universidades e os laboratórios de pesquisa e está começando a ser usada para resolver problemas do mundo-real em uma escala de aplicações industriais e comerciais. Aplicações em uso existem hoje e novos sistemas estão sendo desenvolvidos cada vez mais. Nesta seção nós identificamos as principais áreas onde as abordagens baseadas em agente estão sendo usadas e provemos indicadores para alguns exemplos de sistemas nestas áreas [Agentlink03].

Aplicações industriais – Aplicações industriais de tecnologia de agente estão entre as primeiras que foram desenvolvidas, e hoje, agentes estão sendo aplicados em uma larga escala de sistemas industriais:

- Fabricação: Sistemas nessa área incluem projeto de configuração de produtos de fabricação, projeto colaborativo, cronograma e controle de operações de fabricação, controle de fabricação de um robô e determinação de seqüências de produção para um fábrica [Jennings03b].

- Controle de processo: Sistemas para monitoração e diagnóstico de faltas em plantas de força nuclear, controle de espaço-nave, controle climático, controle de processamento de bobina de aço, gerencia de transporte elétrico e controle de acelerador de partícula.

- Telecomunicações: Sistemas baseados em agentes podem ser construídos e incluem controle de rede, transmissão e chaveamento, gerência de serviço e gerência de rede e para prover serviços melhores, mais rápidos e mais confiáveis.

- Controle de Tráfico Aéreo: Agentes são usados para representar tanto os equipamentos de aviação quanto vários sistemas de controle de tráfego aéreo em operação.

- Sistemas de transporte: O domínio de gerência de tráfico e transporte é adequado para uma abordagem baseada em agente por causa da sua natureza geograficamente distribuída.

Aplicações Comerciais – Enquanto aplicações industriais tendem a ser altamente complexas, indicando sistemas que operam em áreas de nicho comparativamente pequeno, as aplicações comerciais, especialmente aquelas preocupadas com gerência da informação, tendem a ser orientadas muito mais para o mercado.

- Gerência de informação: Mecanismos de busca são realizados por agentes, que agem autonomamente para buscar na *web* em benefício de algum usuário. De fato esta é provavelmente uma das áreas mais ativas para aplicações de agente. Outras

áreas incluem um assistente pessoal que aprende sobre os interesses do usuário e com base neles compila um jornal pessoal, um agente assistente para automatizar várias tarefas de usuário em um *desktop* de computador, um agente de procura de *home page*, um assistente de navegação na *web* e um agente especialista de localização.

- Comércio eletrônico: Algumas tomadas de decisão comerciais podem ser colocadas nas mãos de agentes. Um aumento da quantidade de comércio está sendo empreendido por agentes. Entre estas aplicações comerciais encontra-se um agente que descobre os *Compact Discs* mais baratos, um assistente pessoal de compra capaz de buscar lojas on-line para disponibilizar o produto e informação sobre o preço, um mercado virtual para comercio eletrônico e vários catálogos interativos baseados em agentes.

- Gerência de processo de negócio: Organizações têm procurado desenvolver vários sistemas de Tecnologia da Informação para dar assistência a vários aspectos da gerência de processos de negócio. Outras aplicações nesta área incluem um sistema de gerência de cadeias fornecedoras, um sistema para gerência de fluxos de trabalho heterogêneos e um sistema baseado em agentes móveis para gerência de fluxo de trabalho interorganizacional.

Aplicações de entretenimento – Agentes têm um papel obvio nos jogos de computador, cinema interativo, e aplicações de realidade virtual: tais sistemas tendem a ser cheios de caracteres animados semi-autônomos, que podem naturalmente ser implementados como agentes.

Aplicações médicas – Recentes aplicações incluem a área de monitoração de pacientes e plano de saúde. [Jennings01].

Diante do uso crescente dessa nova tecnologia alguns obstáculos e desafios foram encontrados no desenvolvimento de software orientado a agentes, os quais serão apresentados a seguir.

2.7 Desafios e Obstáculos do Desenvolvimento Orientado a Agentes

Depois de destacar os benefícios potenciais da engenharia de software orientada a agentes, apresentamos nesta seção algumas das desvantagens inerentes à construção de software usando esta nova tecnologia. O conjunto de problemas exposto a seguir é diretamente atribuído às características do software orientado a agente e são intrínsecos a abordagem. Entretanto, projetistas tem encontrado meios de contornar estes problemas, já que sistemas de agente robustos e confiáveis têm sido construídos.

O fato de que agentes têm de agir perseguindo seus objetivos enquanto mantém uma interação contínua com seu ambiente torna difícil projetar software capaz de manter um equilíbrio entre comportamento pró-ativo e reativo. Para atingir este equilíbrio é necessário que a tomada de decisão seja sensível ao contexto, resultando num grau significante de imprevisibilidade sobre quais objetivos o agente perseguirá, em quais circunstancias e quais os métodos que serão usados para atingir os objetivos escolhidos [Jennings00].

As interações de agentes apresentam um nível de sofisticação e força. Contudo, como agentes são autônomos, os padrões e os efeitos de suas interações são imprevisíveis. Em resumo, tanto a natureza (uma requisição simples versus uma negociação prolongada) como a conseqüência de uma interação pode não estar determinada desde o início do projeto do sistema multi-agentes [Jennings00].

Outro aspecto da imprevisibilidade no projeto de sistemas orientados a agentes refere-se a noção de comportamento emergente. Foi reconhecido que a composição interativa de agentes resulta no fenômeno comportamental que não pode ser entendido de forma geral exclusivamente em termos do comportamento dos agentes individuais.

Além desses problemas específicos da tecnologia de agentes, também identificamos várias armadilhas que ocorrem repetidamente no desenvolvimento de sistemas multi-agentes, além de outros que são comuns a qualquer nova tecnologia. Eles são classificados geralmente em cinco grupos [Tveit01]: Política, Conceitual, Análise e Projeto, Nível de Agente e Nível de Sociedade. A seguir descreveremos esses grupos em detalhe:

Política

- Apresentar com entusiasmo excessivo as soluções de agente ou falhar em entender onde agentes podem ser aplicados utilmente

A tecnologia de agente representa uma inovação e uma importante maneira de conceituar e implementar software, mas é importante entender suas limitações.

- Não tome agentes como sendo uma religião ou dogma

Embora haja muitas aplicações usando agentes, eles não são uma solução universal. Há muitas aplicações para as quais os paradigmas convencionais de desenvolvimento de software (tal como programação OO) são mais apropriados.

<u>*Conceitual*</u>

- Esquecer que está desenvolvendo software

A ausência de técnicas testadas e confiáveis para dar assistência no desenvolvimento de software multi-agentes encoraja os projetistas a esquecerem que estão na verdade desenvolvendo software. Nesse caso, a falha do projeto não é por causa dos problemas específicos de agente, mas porque a boa prática da engenharia de software foi ignorada.

- Esquecer que está desenvolvendo software com múltiplas linhas de controle (*multi-threaded*)

Por natureza, sistemas multi-agentes tendem a ter múltiplas linhas de controle (tanto num agente quanto numa sociedade de agente). Assim, na construção de software multi-agentes, é vital não ignorar as lições aprendidas da comunidade de sistemas concorrentes e distribuídos, pois os problemas inerentes aos sistemas com múltiplas linha de controle não desaparecem apenas porque você adotou uma abordagem baseada em agente.

<u>*Análise e Projeto*</u>

- Não saber porque necessita de uma solução orientada a agentes

Pode não haver entendimento de como agentes podem ser usados para melhorar seus produtos nem como eles podem capacitá-los a gerar novas linhas de produto, etc.

- Querer construir soluções genéricas para problemas exclusivos

Tipicamente se manifesta no planejamento de uma arquitetura que supostamente possibilita que uma classe completa de sistemas seja construída, quando o que realmente é solicitado é uma solução feita sob encomenda direcionada para um único problema.

- Acreditar que usar agentes é a única solução (*Silver Bullet*)

A tecnologia de agente é um paradigma de software novo, emergente e ainda não testado na sua essência. Portanto, ela não pode ser considerada uma solução mágica. Isto seria perigosamente ingênuo. Há bons argumentos que indicam que a tecnologia de agente levará a melhorias no desenvolvimento de software distribuído complexo. Mas estes argumentos ainda não foram quantitativamente comprovados.

- Decidir se quer uma arquitetura própria de agentes

Quando se inicia um projeto multi-agentes, tende-se a imaginar que nenhuma arquitetura de agente existente atende aos requisitos do seu problema. Portanto, há uma tentação para projetar uma arquitetura a partir do zero. Isto é freqüentemente um erro, pois deve-se estudar as várias arquiteturas descritas na literatura antes de construir uma própria.

- Gastar todo o tempo implementando infra-estrutura

Um dos maiores obstáculos para o grande uso da tecnologia de agente é que não há nenhuma plataforma que forneça toda a infraestrutura básica (para manipulação de mensagem, rastreamento, monitoração, gerência em tempo de execução, etc.) requerida para criar sistemas multi-agentes. Como resultado, quase todos os projetos de sistemas multi-agentes têm tido uma porção significante dos recursos disponíveis devotados a implementar esta estrutura. Conseqüentemente, resta pouco tempo, energia, ou até

mesmo entusiasmo para trabalhar ou nos agentes em si, ou nos aspectos sociais/cooperativos do sistema.

- Os agentes interagem muito livremente ou de forma desorganizada

Os comportamentos dinâmicos de sistemas multi-agentes são complexos e podem ser caóticos. Se um sistema contém muitos agentes, então as dinâmicas podem se tornar muito complexas de se gerenciar efetivamente. Alguma forma de estruturar a sociedade de agentes é necessária para reduzir a complexidade do sistema, para aumentar o sistema eficientemente e para modelar de forma mais exata o problema que está sendo lidado.

Nível de Agente

- Os agentes usam muita Inteligência Artificial

Ao se focar demais em aspectos de "inteligência", o *framework* de construção de um agente fica muito sobrecarregado com técnicas experimentais (interfaces de linguagem natural, planejadores, provadores de teorema, sistemas de manutenção de razão, etc.). Para ser útil sugere-se construir agentes com um mínimo de técnicas Inteligência Artificial. De fato, alguns pesquisadores chegam a afirmar que, dependendo da aplicação, somente 1% de inteligência artificial é necessário [Jennings03a].

Nível de Sociedade

- O projeto não explora concorrência

Se há a necessidade de uma única linha de controle num sistema, então a utilidade de uma solução baseada em agente deve ser seriamente questionada.

- Ver agentes em qualquer lugar

Quando se adota este ponto de vista, a tendência é concluir o uso de agentes para tudo – incluindo para entidades computacionais de granulação muito fina. Porém a sobrecarga

gerada para gerenciar agentes e a comunicação inter-agentes irá pesar mais do que os benefícios de uma solução baseada em agente.

- Ter pouquíssimos agentes

Alguns projetistas criam um sistema que completamente falha em explorar o poder oferecido pelo paradigma de agente e desenvolvem uma solução com um número muito pequeno de agentes fazendo todo o trabalho. Tais soluções tendem a não atender o teste de coerência da engenharia de software, que requer que um módulo de software tenha uma única função coerente.

A seguir estarão expostos os obstáculos chave que devem ser superados para que a engenharia de software orientada a agentes se torne popular [Wooldridge01b]:

- Organização da relação entre agentes e outros paradigmas – Não está claro como o desenvolvimento de sistemas multi-agentes irá coexistir com outros paradigmas de software, tal como o desenvolvimento orientado a objetos.

- Metodologias orientadas a agentes – embora várias metodologias preliminares de análise e projeto orientados a agentes estejam sendo propostas, há comparativamente pouco consenso entre elas e nenhum acordo sobre os tipos de conceitos que a metodologia deveria suportar [Luck03].

- Engenharia para sistemas abertos – precisa-se de um melhor entendimento de como fazer engenharia em sistemas abertos. Em tais sistemas, acredita-se que é essencial ser capaz de reagir a eventos imprevistos, explorar oportunidades onde estes eventos surgem, e conseguir dinamicamente acordos com componentes do sistema cuja presença não poderia ser predita em tempo de projeto.

- Engenharia de escalabilidade – precisa-se de um melhor entendimento de como fazer engenharia de forma segura e previsível em sistemas que consistem de um grande número de agentes interagindo dinamicamente uns com os outros a fim de atingir suas metas.

Nesta seção nossa intenção não foi sugerir que o desenvolvimento baseado em agentes é mais complexo e inclinado a erro do que as abordagens tradicionais de engenharia de software. Ao invés disso, nós reconhecemos que há certas armadilhas que são comuns às soluções de agente – assim como há certas armadilhas comuns às soluções baseadas em objetos. Reconhecendo essas armadilhas, não podemos garantir o sucesso de um projeto de desenvolvimento baseado em agentes, mas podemos pelo menos eliminar algumas das fontes de falha mais óbvias na sua construção. A seguir apresentamos as considerações finais acerca da engenharia de software orientada a agentes.

2.8 Considerações Finais

As empresas que desenvolvem software de grande porte estão apresentando grandes problemas com os atuais paradigmas de desenvolvimento. Isto é constatado, por exemplo, pela dificuldade de expressar certos conceitos como objetos (por exemplo, abstrações de entidades ativas) para tipos específicos de aplicações. Além disso, a orientação a objeto demanda de um esforço considerável dos engenheiros de software para produzir software de qualidade. Nesse contexto, a orientação a agentes surge como uma nova abordagem para construir sistemas complexos e está começando a ser usada em aplicações industriais e comerciais, deixando de existir apenas nas universidades e laboratórios de pesquisa.

Neste capítulo vimos que existem várias diferenças entre o paradigma de objetos e o paradigma de agentes. No entanto, uma forma bem mais importante de diferenciar agentes de objetos não é baseada em nenhum aspecto em particular, mas no fato de que agentes adicionam diferenças suficientes a objetos para estarem em um nível mais alto (ou diferente) de abstração. Não estamos afirmando que a orientação a agentes é uma solução universal para a construção de software e muito menos que ela irá melhorar em ordem de magnitude o processo de desenvolvimento de software. Entretanto, o uso desta tecnologia na indústria tem demonstrado que as técnicas orientadas a agentes levam a melhorias no desenvolvimento de software distribuído complexo.

Infelizmente, os benefícios prometidos pelo paradigma de agentes ainda não podem ser completamente obtidos porque não existe até o momento um processo de desenvolvimento padronizado para construir software orientado a agentes. Atualmente, muitos deles são

construídos de forma *ad-hoc* e outros utilizam metodologias ainda em desenvolvimento. Nesta direção, o próximo capítulo irá abordar as principais metodologias orientadas a agentes em desenvolvimento hoje a fim de suportar engenharia de sistemas multi-agentes e tornar o paradigma de agente amplamente difundido.

Capítulo 3 - Metodologias Orientadas a Agentes

Este capítulo aborda a necessidade de metodologias de desenvolvimento orientado a agentes e apresenta as principais metodologias propostas para a engenharia de software orientada a agentes.

3.1 Introdução

A engenharia de software orientada a agentes nos apresenta um novo nível de abstração para construir sistemas de software. Essa nova abstração permite que o engenheiro de software projete um sistema em termos de agentes que interagem entre si. No entanto, hoje em dia os projetistas de software ainda não pode explorar todos os benefícios oferecidos pelo paradigma de agentes devido à ausência de notações diagramáticas, metodologias e ferramentas de projeto reconhecidas para o desenvolvimento orientado a agentes [Bergenti00].

Para gerenciar com sucesso a complexidade associada ao desenvolvimento, manutenção e distribuição de sistemas multi-agentes, um conjunto de técnicas e ferramentas de engenharia de software é solicitado ao longo do ciclo de vida do software. Vantagens óbvias serão alcançadas se uma linguagem de agente puder ser facilmente integrada aos processos e ferramentas de engenharia de software existentes.

Assim como acontece com metodologias tradicionais, a análise orientada a agentes começa da definição de requisitos e metas de um sistema. O domínio de aplicação é estudado e modelado, os recursos computacionais disponíveis e as restrições tecnológicas são listados, as metas e os alvos fundamentais da aplicação são planejados, e assim por diante. A partir daí, as metas globais da aplicação são tipicamente decompostas em sub-metas menores, até que elas se tornem gerenciáveis [Zambonelli00].

Em particular, ao lidar com a análise de um agente, não estamos mais limitados a definir funcionalidades computáveis, como nas metodologias orientadas a procedimento, ou conjuntos de serviços relacionados para serem encapsulados em um objeto ou componente. Ao invés disso, a análise orientada a agentes deve identificar as responsabilidades de um agente, já que, por definição, agentes têm metas que são perseguidas de forma pró-ativa e autônoma. Em outras palavras, a análise deve identificar as atividades que os agentes têm de realizar para executar uma ou mais tarefas, que podem incluir permissões de acesso específicas e influência do ambiente circunvizinho, assim como interações com os outros agentes na aplicação. O projeto se preocupa com a representação dos modelos abstratos resultantes da fase de análise em termos das abstrações de projeto providas pela

metodologia. Na engenharia de software orientada a agente, isto significa que responsabilidades, tarefas e protocolos de interação deveriam ser mapeados em agentes, interações de alto-nível e organizações.

A construção de sistemas multi-agentes não é fácil, pois tem-se todos os problemas dos sistemas distribuídos e concorrentes tradicionais e as dificuldades adicionais que surgem dos requisitos de flexibilidade e interações sofisticadas. Além disso, qualquer metodologia para engenharia de software orientada a agentes deve prover abstrações adequadas e ferramentas para modelar não só as tarefas individuais, mas também as tarefas sociais dos agentes. Nesse sentido, várias metodologias têm sido propostas nos últimos anos para suportar a construção de sistemas multi-agentes [Zambonelli00].

Durante o desenvolvimento dessas propostas, foi observado que metodologias tradicionais de análise e projeto são pouco adequadas para sistemas multi-agentes por causa do mau casamento entre os respectivos níveis de abstração nas várias fases do desenvolvimento. No entanto, essa incompatibilidade foi desprezada e algumas propostas usam técnicas de modelagem e metodologias orientadas a objetos como sua base. Como exemplo destas propostas podemos citar as metodologias Gaia [Wooldridge00], AUML [Odell01], MESSAGE/UML [Caire01] e MaSE [DeLoach01].

Outra abordagem, chamada Tropos [Castro02], é baseada nos conceitos usados durante a análise de requisitos iniciais e tenta modelar e implementar sistemas multi-agentes de um ponto de vista orientado a requisitos, visando reduzir tanto quanto possível o mau casamento de impedância entre o sistema e seu ambiente.

Em particular, essas metodologias usam a *Unified Modeling Language* (UML), uma linguagem oriunda da comunidade orientada a objetos, para desenvolver requisitos e/ou projetos para sistemas baseados em agente. Embora a UML tenha suas raízes no mundo orientado a objetos, ela tem evoluído para uma linguagem de modelagem gráfica de propósito geral que pode ser usada na engenharia de muitos tipos diferentes de sistemas de software. Uma das vantagens da UML é a sua capacidade de acomodar o inteiro ciclo de vida do desenvolvimento de software. Além disso, ela é suportada por muitas ferramentas comerciais, que provêem a base para muito do desenvolvimento de software das

organizações atualmente [Papasimeon01]. Dessa forma, uma parte da comunidade de pesquisa na área de desenvolvimento de sistemas multi-agentes tem dedicado grande esforço em propor extensões da UML com a finalidade de acomodar muitos dos aspectos de agentes [Odell01].

A seguir analisaremos algumas das metodologias citadas. Elas estão sendo projetadas como parte de um esforço contínuo em prover suporte metodológico e ferramental para a construção de Sistemas multi-agentes numa dimensão de larga-escala

3.2 A metodologia GAIA

A metodologia Gaia [Wooldridge00] é aplicável a grande conjunto de sistemas multi-agentes, além de lidar com aspectos de nível macro (sociedade) e de nível micro (agente) dos sistemas. Gaia é baseada na visão de que um sistema multi-agentes se comporta como uma organização computacional que consiste de vários papéis interagindo. Ela permite que um analista vá sistematicamente do estabelecimento de requisitos até um projeto que seja suficientemente detalhado a ponto de ser implementado diretamente. Além disso, Gaia toma emprestada parte da terminologia e notação da análise e projeto orientados a objetos.

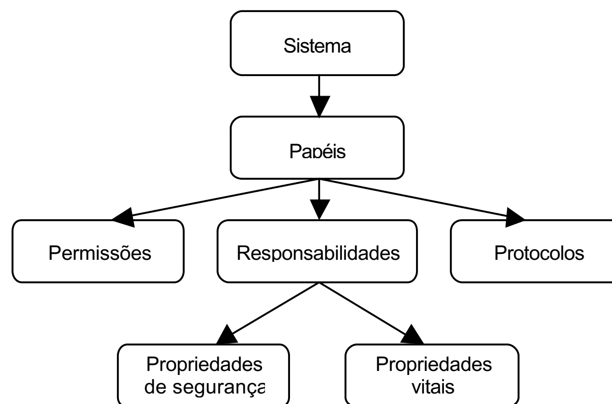


Figura 3.1 - Conceitos de Análise

Em particular, Gaia encoraja um projetista a pensar na construção de sistemas baseados em agentes como um processo de projeto de uma estrutura organizacional. Seus principais conceitos podem ser divididos em duas categorias: abstratas e concretas. Entidades abstratas são aquelas usadas durante a análise para conceituar o sistema, mas que não têm

necessariamente qualquer realização direta no sistema. Os conceitos abstratos (Figura 3.1) incluem Papéis, Permissões, Responsabilidades, Protocolos, Atividades, Propriedades Vitais e Propriedades de Segurança. As entidades concretas, por outro lado, são usadas no processo de projeto e tipicamente terão realização direta no sistema executável. Os conceitos concretos incluem Tipos de Agente, Serviços e Conhecimentos.

O objetivo do estágio de análise é o de desenvolver um entendimento do sistema e a sua estrutura (sem referência a nenhum detalhe de implementação). No caso de Gaia, este entendimento é capturado na organização do sistema. Uma organização é vista como uma coleção de papéis, que se mantêm em certos relacionamentos uns com os outros. Além disso, estes papéis participam de padrões sistemáticos e institucionalizados de interação com outros papéis. Assim, o modelo de organização Gaia é compreendido de dois outros modelos (Figura 3.2): o modelo de papéis, que identifica os papéis chave no sistema; e o modelo de interações, que identifica as dependências e os relacionamentos entre os vários papéis numa organização multi-agentes.

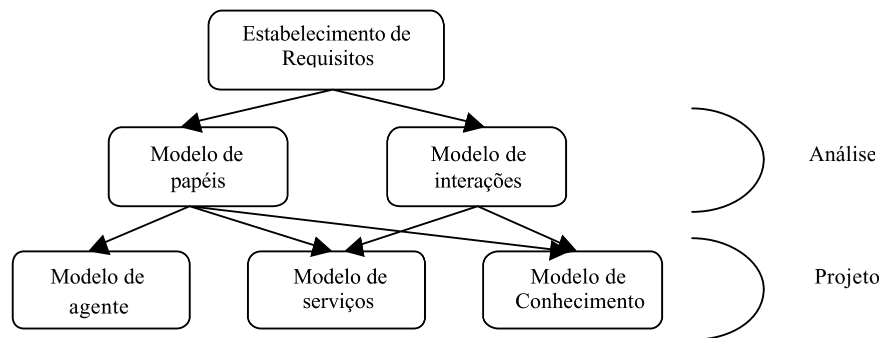


Figura 3.2 - Relacionamentos entre os modelos Gaia

A idéia de um sistema como uma sociedade é útil quando pensamos sobre o próximo conceito: papéis. Um papel é definido por quatro atributos: responsabilidades, permissões, atividades e protocolos. As responsabilidades determinam funcionalidades e, conseqüentemente, são os atributos chaves associados a um papel. Elas são divididas em dois tipos: propriedades vitais (*liveness*) e propriedades de segurança (*safety*). As propriedades vitais intuitivamente estabelecem que “alguma coisa boa aconteça”. Elas descrevem aquelas situações que um agente pode provocar em determinadas condições

ambientais. Em contraste, propriedades de segurança são invariáveis. Intuitivamente, uma propriedade de segurança estabelece que "nada de ruim aconteça" (isto é, que uma situação aceitável seja mantida durante todos os estados da execução).

Um papel tem um conjunto de permissões necessárias para realizar suas responsabilidades. Permissões são "direitos" associados a um papel. Elas identificam os recursos que estão disponíveis para aquele papel de modo que ele possa realizar suas responsabilidades. Permissões tendem a ser recursos de informação. As atividades de um papel são computações associadas a ele. Elas podem ser realizadas por um agente sem que ele precise interagir com outros agentes. Finalmente, um papel também é identificado por vários protocolos que definem a maneira com que ele interage com outros papéis.

O processo de projeto Gaia envolve a geração de três modelos (Figura 3.2). O modelo de agente identifica os tipos de agente que vão formar o sistema e as instâncias de agente geradas a partir destes tipos. O modelo de serviços identifica os principais serviços que são requisitados para realizar o papel do agente. Por fim, o modelo de conhecimento documenta as linhas de comunicação entre os diferentes agentes.

A metodologia Gaia está preocupada em como uma sociedade de agentes coopera para realizar as metas do sistema. Na verdade, como um agente realiza seus serviços está além do escopo de Gaia e irá depender do domínio da aplicação.

3.3 A metodologia AUML

Agent UML (AUML) [Odell01] é uma metodologia de análise e projeto que estende a UML para representar agentes. Ela sintetiza uma preocupação crescente das metodologias de software baseado em agentes com o aumento da aceitação da UML para o desenvolvimento de software orientado a agentes.

Agentes são vistos como objetos ativos, exibindo autonomia dinâmica (capacidade de ação pró-ativa) e autonomia determinística (autonomia para recusar uma solicitação externa). O objetivo da AUML é fornecer uma semântica semi-formal e intuitiva através de uma notação gráfica amigável para o desenvolvimento de sistemas orientados a agentes.

A AUML fornece extensões da UML, adicionando uma representação em três camadas para os protocolos de interação de agentes, que descrevem um padrão de comunicação com uma seqüência permitida de mensagens entre agentes e as restrições sobre o conteúdo destas mensagens.

Os protocolos de agente são representados por uma abordagem em camada (Figura 3.3) que definem:

- modelos e pacotes: onde o protocolo como um todo é tratado como uma entidade independente. Um pacote é uma agregação conceitual de seqüências de interação. Um modelo identifica entidades que não se limitam ao pacote, mas que precisam ser limitadas quando o modelo do pacote estiver sendo instanciado;
- diagramas de seqüência e colaboração: fornecem uma visão alternativa das interações entre os agentes (captura a dinâmica inter-agente); e
- diagramas de estados e de atividades: detalham o comportamento interno do agente quando este estiver executando os protocolos.

Além dessas extensões, o autor também propõe outras modificações para representar várias noções comumente empregadas pela comunidade de agentes. Expressar os papéis que um agente pode exercer no curso de uma interação com outros agentes é essencial para modelar os sistemas baseados em agentes. Assim, embora os diagramas de colaboração atualmente não permitam representar os papéis de um agente durante uma interação, cada ato de comunicação poderia ser rotulado com o papel responsável por sua requisição. Os diagramas de atividades também podem ser modificados para representar papéis de um agente, associando cada atividade com o nome do papel classificador apropriado. Além disso, a mudança de papéis pode ser representada em diagramas de atividades através de notas.

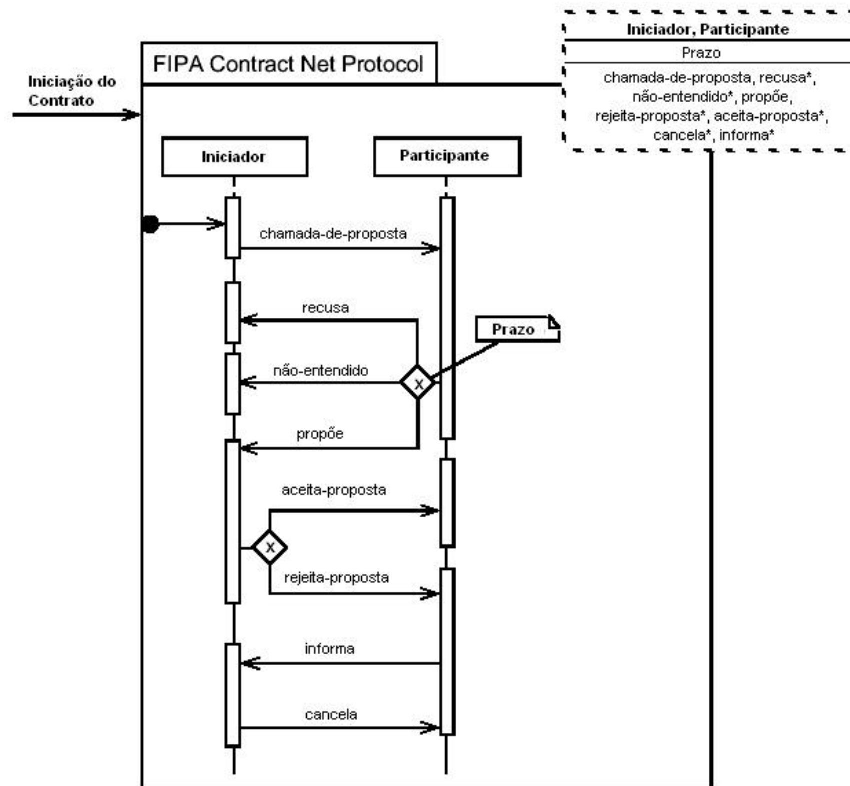


Figura 3.3 - Um Protocolo de Interação de Agente genérico expressado como um pacote modelo

Uma importante propriedade de um agente é a mobilidade e uma forma de representá-la é usando extensões dos diagramas de distribuição para indicar caminhos de mobilidade e declarações locais. A fim de representar a situação em que um agente em si é uma interface entre o mundo externo e uma implementação baseada em agente propõe-se usar uma pequena extensão de pacotes UML. Sabendo que os agentes tomam emprestadas muitas analogias da natureza, encontramos também uma proposta de extensão dos diagramas de classes, seqüência e atividades para representar clonagem, mitose e reprodução de agentes, além de representar relacionamentos parasitas e simbióticos entre eles. Vimos no Capítulo 2 que a interação de muitos agentes pode provocar um fenômeno chamando de emergência. Este fenômeno pode ser representado usando uma extensão do diagrama de classes.

A AUML estende a UML em alguns aspectos e fornece modelos para capturar o aspecto dinâmico das interações inter-agente. Contudo, a metodologia ainda não fornece extensões

para capturar o mapa cognitivo do agente (individual/estrutura estática) ou a organização estrutural de agentes (sistema/estrutura estática). Dessa forma, podemos verificar a cobertura da modelagem AUML relativa aos seguintes aspectos:

- Agente Individual/Estrutura Estática: A AUML não fornece extensões da UML que permitam representar o mapa cognitivo de um agente. Nenhum modelo de representação adicional é fornecido para capturar o conhecimento que um agente possui do mundo e de outros componentes do seu mapa cognitivo. Este conhecimento é capturado através de aspectos como desejos, intenções e mecanismos de mobilidade.

- Agente Individual/Dinâmico: o terceiro nível do protocolo de interação de agentes captura o processamento interno do agente e fornece um mecanismo pra representar a dinâmica do mapa cognitivo do agente. A AUML não configura uma arquitetura cognitiva específica e representa a dinâmica interna do agente usando diagramas de atividades e de estados.

- *Sistema Social/Estrutura Estática:* **A** AUML configura a hierarquia de classes incluídas no modelo UML e adiciona a noção do agente aplicando uma regra específica. Nenhum outro mecanismo é fornecido para capturar a estrutura do sistema.

- *Sistema Social/Dinâmico:* os dois primeiros níveis do protocolo de interação de um agente relatam a dinâmica do sistema e podem ser usados tanto para o modelo conceitual quanto para o modelo de projeto. No primeiro nível, modelos e pacotes fornecem soluções reusáveis para o sequenciamento de mensagens.

Entretanto, existem algumas propostas que ainda estão sob análise para serem incorporadas na AUML futuramente e serão mostradas a seguir.

3.3.1 O Comportamento Interno de um Agente

A proposta de [Bauer01a] se foca em um novo subconjunto de extensões da UML baseada em agentes destinado a especificar o comportamento interno de um agente e sua relação

com o comportamento externo. Este tópico diminui o "*gap*" existente entre a definição do protocolo de interação de um agente e o seu comportamento interno.

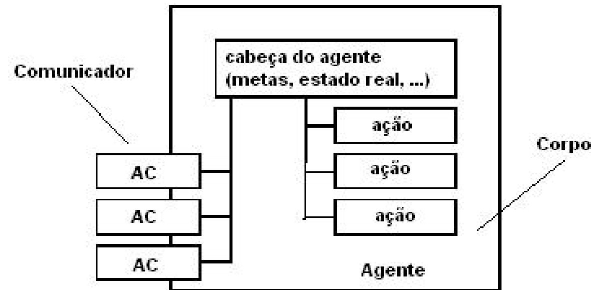


Figura 3.4 - Estrutura interna de um agente

Essa proposta define que um agente consiste de três partes (Figura 3.4): o comunicador, a cabeça e o corpo de um agente. O comunicador de um agente é responsável por sua comunicação física (através de atos de comunicação (AC)). A funcionalidade principal de um agente é implementada no seu corpo. A cabeça de um agente é a sua "chave-geradora". Seu comportamento tem que ser especificado na forma de um autômato de cabeça. Em particular, este autômato relaciona as mensagens de entrada com o estado interno, as ações, os métodos e as mensagens de saída do agente. Estas últimas são chamadas de comportamento reativo do agente. Além disso, o autômato de cabeça de um agente define o seu comportamento pró-ativo, isto é, ele automaticamente dispara ações, métodos e mudanças de estado dependendo do seu estado interno. Conforme a Figura 3.5, o diagrama de classes é estendido através de estereótipos e uma classe é divida em:

- Papéis e Descrições: descrevem o comportamento e um conjunto de propriedades, interfaces, descrições de serviço que caracterizam uma classe de agentes e os papéis que ela suporta.
- Descrição de estado: semelhante aos atributos, mas define a classe Fórmulas Bem Formadas (*Well Formed Formula* - *wff*) para todos os tipos de descrição lógica de estado, independente da lógica em questão. Também adiciona um estereótipo de persistência àqueles de visibilidade para caracterizar que um atributo é persistente.

- Ações: definem ações pró-ativas e reativas. A semântica de uma ação é definida por pré-condições, pós-condições, efeitos e constantes.
- Métodos: definem métodos como na UML convencional, eventualmente com pré-condições, pós-condições, efeitos e constantes.
- Capacidades: definem serviços e podem ser descritas informalmente ou usando diagramas de classe.
- Atos de comunicação (AC): definem o tipo de mensagem trocada e suas informações adicionais, como origem, destino e conteúdo. Atos de comunicação são recebidos/enviados no contexto de algum protocolo de interação. O ato de comunicação que foi enviado ou recebido pode ser alguma classe ou instância concreta.

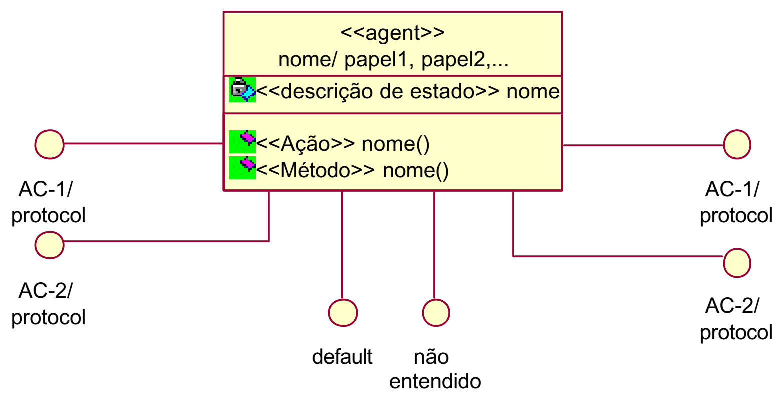


Figura 3.5 - Diagrama de classes para especificar a estrutura interna de um agente

Esta proposta também afirma que os diagramas de seqüência e colaboração são adequados para definir o comportamento concreto da cabeça de um agente. Comportamento este especialmente baseado nas ações, métodos e mudanças de estado do agente. Já os diagramas de estados e atividades são mais adequados para uma especificação mais abstrata do comportamento da cabeça de um agente.

3.3.2 Sistemas Multi-Agentes como Estruturas Sociais

A proposta apresentada por [Parunak01] combina em um *framework* teórico geral vários modelos organizacionais existentes para agentes, incluindo AALAADIN [Ferber98], teoria da dependência, protocolos de interação e *holonics*[2]. Esta proposta mostra como a UML pode ser estendida para capturar os conceitos que incluem Grupo, Papel, Dependência e Atos de fala. O objetivo foi conseguir uma sintaxe coerente para descrever estruturas organizacionais para serem usadas na análise, especificação e projeto de sistemas multi-agentes. É uma abordagem comportamental e define papéis em termos de características acessíveis para um observador externo, ao invés daquelas disponíveis apenas no interior do agente. Grupos e papéis são ilustrados usando dois recursos da UML: o diagrama de classes e a raia. Raias horizontais especificam a instanciação de um papel, enquanto raias verticais especificam um relacionamento de agregação entre grupos, como também os papéis que compreendem estes grupos. Outra forma de expressar os relacionamentos entre grupos é usando o diagrama de classes. As interações que podem ocorrer entre as entidades presentes no diagrama de classes podem ser representadas usando o diagrama de seqüência. Estes diagramas definem papéis como padrões de interação. Esta proposta ainda define o grafo de atividade de fluxo de objetos que modela um grupo como uma entidade de processamento e remove detalhes para prover uma visão de alto-nível dos componentes do sistema. Dessa forma, este diagrama também permite modelar grupos de agentes como agentes.

Em resumo, o modelo definido por essa proposta é baseado em AALAADIN, porém com três extensões:

- Papéis não são ontologicamente primitivas, mas são definidas como padrões recorrentes de dependências e ações.
- A definição de um grupo inclui não apenas um conjunto de agentes que ocupam papéis, mas também um ambiente através do qual eles interagem.

[2] Holonics é uma abordagem "bottom up" centralizada e provê princípios que garantem um maior agrupamento de reatividade e manipulação da complexidade do sistema.

- O requisito de AALAADIN que estabelece a interação de grupos apenas através de membros identificados é relaxado no caso de grupos não analisados, que possuem a permissão de ocupar papéis em grupos de nível maior conforme o modelo *holonic.*

A ontologia consolidada por este *framework* é mostrada na Figura 3.6.

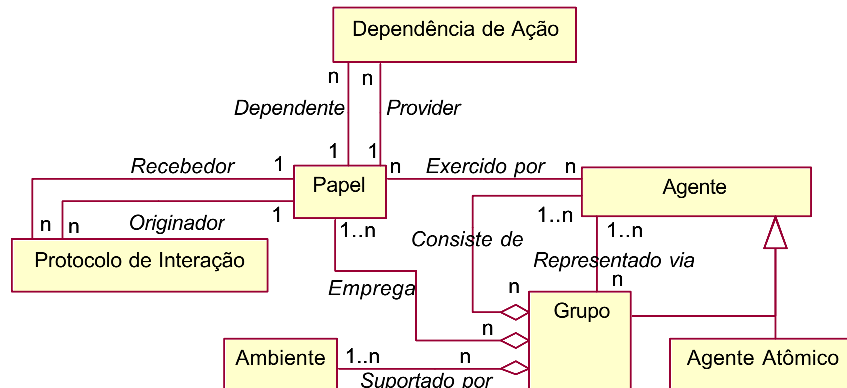


Figura 3.6 - Ontologia Consolidada

Na próxima seção apresentamos uma metodologia de desenvolvimento orientada a agentes que incorpora as melhores práticas da engenharia de software presentes na época de sua concepção.

3.4 A metodologia MESSAGE/UML

A metodologia MESSAGE (*Metodology for Engineering Systems of Software Agents*) [Caire01] cobre a análise e o projeto de sistemas multi-agentes usando conceitos bem definidos e uma notação baseada em UML.

Os conceitos da UML são usados para modelar as entidades MESSAGE em um nível detalhado (ou micro). Isto é, de um ponto de vista estrutural eles são objetos com atributos e operações realizadas por métodos. Já de um ponto de vista comportamental eles são máquinas de estado. A maioria dos conceitos de nível de conhecimento, definidos em MESSAGE, se divide nas seguintes categorias: Entidade Concreta, Atividade e Entidade de Estado Mental. Os principais tipos de Entidade Concreta são: Agente, Organização, Papel e Recurso. Um Agente é definido como uma entidade autônoma atômica que é capaz de

executar alguma função útil. A capacidade funcional é capturada pelos serviços do Agente. Uma organização é um grupo de Agentes que trabalham juntos para algum propósito comum. Um papel descreve as características externas de um Agente em um contexto particular. A distinção entre Papel e Agente é análoga a que existe entre Interface e Classe na orientação a objetos. Um recurso é usado para representar entidades não-autônomas como bancos de dado ou programas externos usados por Agentes.

Os principais tipos de Atividade são: Tarefa, Interação e Protocolo de Interação. Uma tarefa é uma unidade de atividade no nível de conhecimento com um único responsável por executá-la. Os conceitos de Interação e Protocolo de Interação são emprestados da Metodologia Gaia. Uma Interação possui mais de um participante e um propósito que os participantes visam atingir. Um Protocolo de Interação define um padrão de troca de mensagens associado a uma Interação.

Atualmente, há um tipo de Entidade de Estado Mental em foco chamado Meta. Uma Meta associa um Agente a uma Situação. Uma Situação é um nível de conhecimento análogo ao de um estado do mundo. Outros conceitos simples, porém importantes usados em MESSAGE são: Entidade De Informação e Mensagem.Uma Entidade de Informação é um objeto encapsulando um pedaço de informação, enquanto uma Mensagem é um objeto comunicado entre dois agentes.

MESSAGE ainda usa os conceitos comportamentais da UML tais como Ação, Evento e Estado, para definir as propriedades, interações e processos físicos da sua visão do mundo. A Figura 3.7 proporciona uma visão geral informal centrada em agente de como estes conceitos são inter-relacionados.

MESSAGE define várias de visões de modelo de análise que focam em subconjuntos de entidades e conceitos de relacionamento. A Visão de Organização mostra Entidades Concretas no sistema e seu ambiente, além dos relacionamentos entre elas. A Visão de Meta/Tarefa mostra Metas, Tarefas, Situações e as dependências entre elas. A Visão de Agente/Papel foca nos Agentes e Papéis individuais. A Visão de Interação mostra as informações que detalham cada interação entre os agentes/papéis. Por fim, a Visão de

Domínio mostra os conceitos específicos de domínio e as relações que são relevantes para o sistema em desenvolvimento.

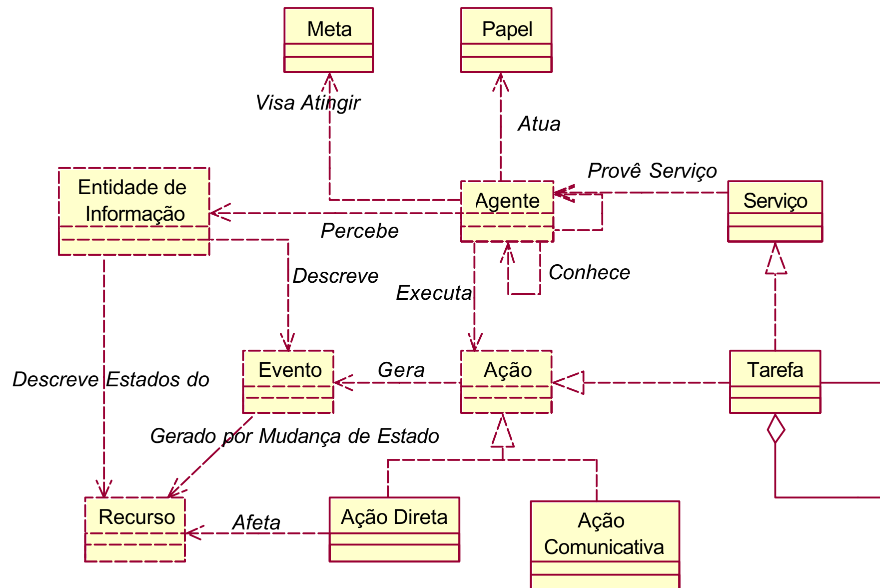


Figura 3.7 - Conceitos MESSAGE

O processo de análise se inicia definindo o nível mais alto de decomposição como nível 0 e utiliza uma abordagem de refinamento dos modelos produzidos neste nível. Estes modelos representam uma visão geral do sistema, seu ambiente e sua funcionalidade global. Aqui o processo de modelagem começa construindo as visões de Organização e Meta/Tarefa, que servem como entrada para a criação de visões de Agente/Papel e Domínio. A visão de Interação é construída usando a entrada dos outros modelos. O nível subseqüente define a estrutura e o comportamento das seguintes entidades: organização, agentes, tarefas e entidades do domínio da meta. Os próximos níveis devem ser definidos para analisar com aspectos específicos do sistema que lidam com requisitos funcionais e não-funcionais.

MESSAGE pode utilizar várias estratégias para refinar os modelos do nível “0”. As abordagens centradas na organização foca em analisar propriedades gerais, tais como a estrutura do sistema, os serviços oferecidos, tarefas e metas globais, papéis principais e recursos. Os agentes necessários para atingir as metas aparecem naturalmente durante o processo de refinamento. As abordagens centradas em agente focam na identificação dos

agentes necessários para prover a funcionalidade do sistema. A organização mais adequada é identificada de acordo com os requisitos do sistema. As abordagens orientadas a interação sugerem o refinamento progressivo de cenários de interação que caracterizam o comportamento interno e externo da organização e dos agentes. As abordagens de decomposição de meta/tarefa são baseadas na decomposição funcional. Conseqüentemente, observando a estrutura geral de metas e tarefas na visão de Meta/Tarefa tomar decisões acerca dos agentes e da organização mais adequada para atingir tais metas/tarefas.

MESSAGE argumenta que as diferentes visões do sistema deixam o analista livre para escolher a estratégia mais apropriada. Os diagramas de MESSAGE estendem os diagramas de classe e atividade UML. Um modelo de análise mais completo seria alcançado com a notação UML existente e os diagramas de seqüência AUML [Odell01] para descrever interações de papel/agente.

Embora as metodologias até agora apresentadas sejam baseadas em práticas e conceitos bem consolidados na engenharia de software, elas não cobrem o ciclo completo de desenvolvimento orientado a agentes. Por exemplo, a metodologia AUML só cobre a fase de projeto, enquanto que as metodologias Gaia e MESSAGE cobrem as fases de análise e projeto. Contudo, já existem metodologias mais completas, conforme veremos nas próximas seções.

3.5 A metodologia *MaSE*

O Instituto de Tecnologia da Força Aérea Americana está construindo a metodologia MaSE (Multiagent Systems Engineering) e a ferramenta associada agentTool [DeLoach01], que suportam grande parte do ciclo de vida do desenvolvimento de um sistema multi-agente. MaSE começa da representação textual do sistema e segue de forma estruturada até a sua implementação e combina vários modelos preexistentes em uma única metodologia estruturada.

MaSE define sistemas multi-agentes em termos de classes de agentes e organizações de agentes, que por sua vez são definidas em termos dos agentes que se comunicam através de conversação. Há duas fases básicas em MaSE (Figura 3.8): análise e projeto. A primeira

fase envolve três passos: captura de metas, aplicação de casos de uso e refinamento de papéis.

O primeiro passo, Capturar Metas, identifica os requisitos do usuário e os transforma em metas de alto-nível do sistema. Depois de definir as metas no nível de sistema, os casos de uso são extraídos e máquinas de estado são definidas no passo de aplicação de casos de uso. Este passo define um conjunto inicial de papéis do sistema e caminhos de comunicação. Usando as metas do sistema e os papéis identificados nos casos de uso, o conjunto inicial de papéis é refinado e estendido e as tarefas para realizar cada meta são definidas no passo de refinamento de papéis.

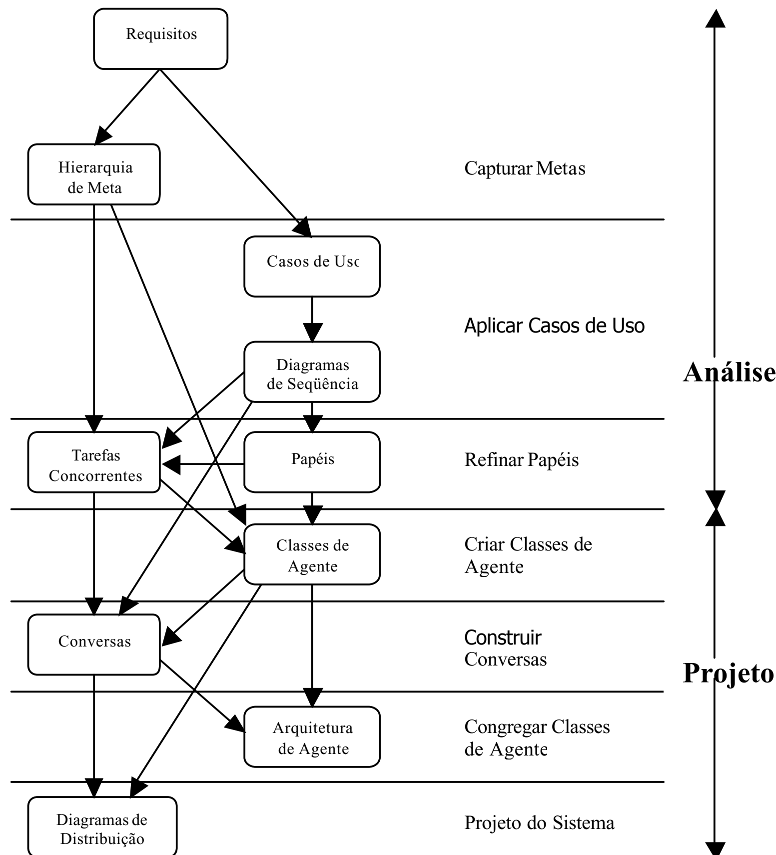


Figura 3.8 - Metodologia MaSE

Na fase de Projeto, os modelos de análise são transformados em elementos úteis para, de fato, implementar o SMA. A fase de Projeto possui quatro fases: Criação de classes de

agente, Construção de conversas, Congregação de classes de agente e Projeto do sistema. No primeiro passo, classes específicas de agente são criadas para cumprir os papéis definidos na fase de Análise. Daí, depois de determinar o número e os tipos de classes de agentes no sistema, as conversações podem ser construídas ou os componentes internos que compreendem as classes de agentes podem ser definidos. O analista pode executar estes passos em paralelo durante os passos de Construção de conversas e Congregação de classes de agente. Uma vez que a estrutura do sistema foi completamente definida, parte-se para a definição de como o sistema será distribuído. Durante este passo, o projetista define o número de agentes individuais, suas posições e outros itens específicos do sistema.

Devido às suposições feitas para simplificar a pesquisa e o projeto da metodologia MaSE, esta apresenta algumas limitações. Primeiro, assume-se que o sistema em desenvolvimento está fechado e que todas as interfaces externas são encapsuladas por um agente participante dos protocolos de comunicação do sistema. Segundo, a metodologia não considera sistemas dinâmicos nos quais agentes podem ser criados, destruídos ou movidos durante a execução. Terceiro, supõe-se que as conversas inter-agentes sejam de um-pra-um, em oposição ao *multicast*. Entretanto, uma série de mensagens ponto-a-ponto pode ser usada para promover o *multicast*. Finalmente, foi suposto que os sistemas projetados com a metodologia MaSE não seriam muito grandes; o alvo seria de até dez classes de agente. Esta não é uma restrição rígida, porém indica simplesmente que nenhuma verificação ou validação de sistemas maiores foi feita e que problemas podem surgir a partir de tais sistemas.

A seguir, descrevemos detalhadamente a metodologia orientada a agentes que será focada nesta dissertação.

3.6 A metodologia *TROPOS*

As técnicas de desenvolvimento de software atuais levam em geral a um software inflexível e não-genérico. Este é o caso devido à eliminação de metas durantes os requisitos finais que congela, no projeto de um sistema de software, uma variedade de afirmações que podem ou não ser verdade no seu ambiente operacional. Dado o aumento sempre crescente por software genérico e componentizado que possam ser baixados e usados em uma variedade de plataformas de computação ao redor do mundo, acredita-se que o uso de conceitos

intencionais durantes as fases finais de desenvolvimento de software se tornará predominante e poderia ser pesquisada mais [Castro01].

Freqüentemente, sistemas de software falham em suportar as organizações das quais eles são parte integrante. Isto acontece devido à ausência de um entendimento apropriado da organização pelos projetistas do sistema de software, bem como a freqüência das mudanças organizacionais que não podem ser acomodadas pelos sistemas de software existentes (ou seus mantenedores). Neste contexto, a engenharia de requisitos vem sendo reconhecida como a fase mais crítica no desenvolvimento de sistemas, porque as considerações técnicas têm de ser balanceadas contra as sociais e organizacionais. O projeto Tropos [Castro02] vem desenvolvendo uma metodologia inspirada em conceitos organizacionais, que reduzem tanto quanto possível este mau casamento de impedância entre o sistema e seu ambiente. Tropos é baseado nos conceitos usados para modelar requisitos iniciais e complementa propostas para plataformas de programação orientadas a agentes. O processo começa com um modelo do ambiente no qual o sistema sendo desenvolvido eventualmente vai operar. O modelo é escrito em termos de atores, suas metas e interdependências. Através de refinamentos incrementais, este modelo é estendido para incluir tanto o sistema a ser desenvolvido quanto os seus subsistemas, que também são representados como atores a quem foram delegadas metas para atingir, planos para executar e recursos para fornecer. A proposta *Tropos* é fundada em um pequeno conjunto de conceitos e provê ferramentas e técnicas para construir modelos baseados nos conceitos oferecidos pelo *i** [Yu95] (que simboliza "intencionalmente distribuído"), um *framework* de modelagem que inclui os conceitos de ator (atores podem ser agentes, posições ou papéis) e suas interdependências intencionais, incluindo dependências de meta, meta-soft, tarefa e recurso. Este *framework* também inclui o *modelo de dependência estratégica* e o *modelo de razão estratégica* que são usados para capturar as intenções dos stakeholders (usuários, proprietários, gerentes, etc.), as responsabilidades do novo sistema com respeito a estes stakeholders, a arquitetura do novo sistema e os detalhes do seu projeto. Estes modelos podem ser usados como parte da documentação de um sistema de software durante a operação e manutenção.

As características chaves do *Tropos* incluem o uso de conceitos de nível de conhecimento [Newell93], tais como agente, meta, plano e outros, durante todas as fases do

desenvolvimento de software. Também é atribuído um papel crucial à análise de requisitos quando o ambiente e o sistema a ser construído são analisados. As fases cobertas por esta metodologia são as seguintes [Castro02]:

- A fase de *Requisitos Iniciais* está preocupada com o entendimento de um problema estudando uma configuração organizacional existente. Durante esta fase, os engenheiros de requisitos modelam os stakeholders como atores e suas intenções como metas. Cada meta é analisada do ponto de vista de seu ator resultando em um conjunto de dependências entre pares de atores. As saídas desta fase são dois modelos:

    1. O modelo de dependência estratégica que captura os atores relevantes, suas metas respectivas e suas interdependências; e

    2. O modelo de razão estratégica que determina através de uma análise meios-fim como as metas podem ser cumpridas através das contribuições de outros atores.

- A fase de *Requisitos Finais* introduz o sistema a ser desenvolvido como um outro ator no modelo de dependência estratégica. O ator que representa o sistema é relacionado aos atores sociais em termos de dependências. Considera-se este ator que representa o sistema e se faz uma análise meio-fim para produzir um novo modelo de razão estratégica. Suas metas são analisadas e irão eventualmente levar a revisar e adicionar novas dependências com um subconjunto de atores sociais (os usuários). Se necessário decompõe-se o sistema que representa o ator em vários sub-atores e se revisa os modelos de razão e dependência estratégica.

- A fase de *Projeto Arquitetural* define a arquitetura global do sistema em termos de sub-sistemas, interconectados através de fluxos de controle de dados. A arquitetura de um sistema constitui um modelo da estrutura relativamente pequeno e intelectualmente gerenciável, que descreve como os componentes do sistema trabalham juntos. Subsistemas são representados como atores e interconexões de dado/controle são representados como dependências de ator (que representa o

sistema). Esta fase consiste de três passos: 1. Selecionar o estilo arquitetural; 2. Refinar os modelos de razão e dependência estratégica e; 3. Aplicar padrões sociais.

Passo 1. Escolher o estilo arquitetural usando como critério as qualidades desejadas que foram identificadas na fase anterior e o framework NFR [Chung00] para conduzir esta análise de qualidade. Incluir novos atores, de acordo com a escolha de um estilo arquitetural específico de agente;

Passo 2. Incluir novos atores e dependênc ias, assim como decompor os atores e as dependências existentes em sub-atores e sub-dependências. Revisar os modelos de razão e dependência estratégica. As capacidades de ator são identificadas da análise das dependências indo e vindo do ator, bem como das metas e planos que o ator irá executar a fim de cumprir requisitos funcionais e não-funcionais.

Passo 3. Definir como as metas associadas a cada ator são cumpridas por agentes com respeito a padrões sociais. Eles são usados para solucionar uma meta específica que foi definida ao nível arquitetural através da identificação de estilos organizacionais e atributos de qualidade relevantes (softgoals). Uma análise detalhada de cada padrão social permite definir um conjunto de capacidades associadas com os agentes envolvidos no padrão. Uma capacidade estabelece que um ator é capaz de agir a fim de atingir uma meta dada. Em particular, para cada capacidade o ator tem um conjunto de planos que podem aplicar em diferentes situações. Um plano descreve a seqüência de ações a executar e as condições sob o qual o plano é aplicável. Capacidades são coletadas num catálogo e associadas ao padrão. Isto permite definir os papéis e capacidade do ator que são adequados para um domínio particular.

- A fase *Projeto Detalhado* visa especificar o nível micro de agente, detalhando capacidades e planos usando o diagrama de atividades, como também protocolos de comunicação e coordenação (modelados, por exemplo, em versões estendidas da UML, tais como AUML [Odell01]). Nesta etapa ocorre:

1. o desenvolvimento de diagramas de atividades para representar as capacidades de cada agente;

2. o desenvolvimento de diagramas de seqüência e colaboração para capturar as interações entre os agentes;

3. a elaboração dos diagramas de atividade baseados em estado para representar cada plano identificado nos diagramas de capacidade.

- A fase de Implementação segue passo a passo, em uma maneira natural, a especificação de projeto detalhado que é transformada em um esqueleto para a implementação. Isto é feito através de um mapeamento entre os conceitos Tropos e os elementos da plataforma de implementação de agente escolhido, tal como JACK [Busetta01]. O código é adicionado ao esqueleto usando a linguagem de programação suportada pela plataforma de programação.

Tropos argumenta em favor de uma metodologia de desenvolvimento de software que seja fundada em conceitos intencionais, tais como aquelas de ator, meta, dependência (de meta, tarefa, recurso, soft-meta), etc. Em particular, defende a solicitação que software de padrão industrial deveria ser organizado da mesma forma que as empresas o são.

Tropos propôs um framework de modelagem que vê software das cinco perspectivas complementares [Castro02]:

Social – quem são os atores relevantes, o que eles querem? Quais são suas obrigações? Quais são suas capacidades?

Intencional – quais são as metas relevantes e como elas se relacionam? Como eles estão sendo conhecidos e quem solicita as dependências?

Comunicativa - como os atores dialogam e como eles podem interagir uns com os outros?

Orientada a processo – quais são os processos de negócio/computador relevantes?Quem é responsável pelo que?

3.7 Considerações Finais

Atualmente, especificar, projetar e implementar sistemas multi-agentes é extremamente difícil. Esta atividade torna-se ainda mais crítica pela imaturidade da tecnologia de agente e pela ausência de notações, ferramentas e metodologias especificas para o desenvolvimento de software OA. Sem isto, torna-se impossível explorar os benefícios oferecidos por este novo nível de abstração. Neste sentido, existem algumas propostas de metodologias voltadas para o desenvolvimento de software orientado a agentes. Tais propostas, como vimos neste capítulo, são na maioria extensões das conhecidas metodologias orientadas a objetos e, conseqüentemente, utilizam extensões da UML visando o seu uso no desenvolvimento orientação a agentes. Outras, no entanto, usam uma abordagem centrada em requisitos e orientada a metas, embora ainda usem extensões da UML para suportar alguma de suas fases, como a metodologia Tropos.

Uma das principais vantagens de se estender UML para agentes é a difusão da tecnologia de agente para uso em larga escala na indústria. Isso é conseguido através do relacionamento desta nova tecnologia com a tecnologia antecedente (isto é, a tecnologia de objetos) e do uso artefatos para suportar o ambiente de desenvolvimento através do completo ciclo de vida do sistema. Assim, torna-se mais fácil para os usuários, que estão familiarizados com desenvolvimento de software orientado a objetos, entenderem o que são os sistemas multi-agentes e os princípios para se conceber um sistema como uma sociedade de agentes, ao invés de uma coleção de objetos distribuídos. Outra vantagem é a redução do tempo gasto para projetar tais sistemas, já que o esforço em construir ferramentas personalizadas ou ambientes de desenvolvimento integrados é drasticamente reduzido. Além disso, usar artefatos ao logo do ciclo de desenvolvimento facilita a comunicação entre a equipe de projeto e ajuda a codificar a melhor prática de desenvolvimento.

Contudo, a tecnologia de agente é nova e ainda está amadurecendo fazendo com que, até o momento, não haja um consenso do que diz respeito à definição de um agente. Isto resulta em um conjunto de diferentes extensões da UML para os diferentes tipos de agentes encontrados na pesquisa da engenharia de software orientada a agentes. Espera-se que à medida que a tecnologia de agente se difunda nos próximos anos, um consenso da indústria

determine qual o conjunto de extensões que finalmente será adotado para o projeto de sistemas multi-agentes.

A cobertura completa do processo de desenvolvimento de software é considerada com sendo essencial para a engenharia de software orientada a agentes. Entretanto, não é só por ir além da fase de requisitos iniciais que uma metodologia orientada a agente pode prover um argumento convincente contra outras metodologias. Em particular, metodologias orientadas a agente são inerentemente intencionais, baseadas em conceitos como agente, meta, plano, etc. Isto mostra claramente a necessidade de se focar no ambiente (organizacional) onde o sistema em desenvolvimento eventualmente vai operar. Entender tal ambiente pode reduzir consideravelmente a incompatibilidade entre o sistema e seu ambiente operacional. Neste contexto, fica clara a relevância do Projeto Tropos no desenvolvimento de software orientado a agentes. Portanto, será o foco desta dissertação. Na Figura 3.9, temos uma comparação entre as metodologias explicadas aqui com respeito à cobertura das fases de desenvolvimento do software que elas suportam.

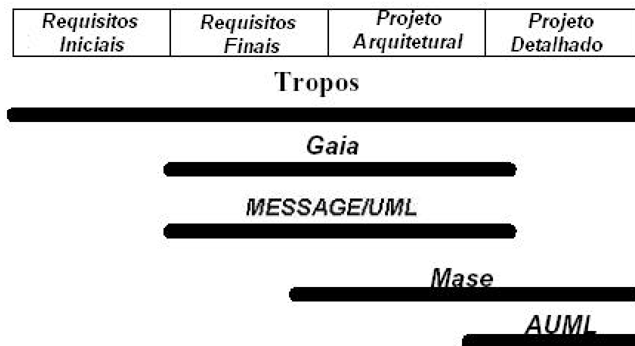


Figura 3.9 - Cobertura das fases de desenvolvimento do software

Em resumo, de uma perspectiva de engenharia de software, a proposta Tropos, embora ainda em constante evolução, ainda apresenta as vantagens de levar a arquiteturas de software mais flexíveis, robustas e abertas e oferecer um *framework* coerente que engloba todas as fases de desenvolvimento de software, dos requisitos inicias até a implementação. Além disso, Tropos é consistente com a próxima geração de paradigma de programação, isto é, a programação orientada a agentes, que está ganhando uma base sólida em áreas de

aplicação chave, tais como telecomunicações, comércio eletrônico e sistemas baseados em *web*.

Entre as várias fases que o Tropos precisa evoluir, encontra-se a etapa de projeto arquitetural. Nesta dissertação nós propomos refinar a fase de projeto arquitetural da metodologia Tropos. Contudo, antes de detalharmos nossa estratégia, no próximo capítulo iremos antes fazer uma breve introdução aos conceitos básicos relacionados à arquitetura de software.

Capítulo 4 - Arquitetura de Software

Este capítulo fornece uma visão geral sobre a importância da arquitetura de software. Inicialmente serão definidos os aspectos e propriedades que caracterizam a arquitetura de um software. Também será descrito o projeto de uma arquitetura, focando na definição de estilos arquiteturais. Finalmente, será apresentada a importância de documentar e descrever adequadamente a arquitetura de um software.

4.1 Introdução

Arquiteturas de software ganharam uma ampla popularidade na última década, tendo sua importância reconhecida na engenharia de software. Considera-se que ela exerça um papel fundamental em lidar com dificuldades inerentes ao desenvolvimento de sistemas de software complexos e de grande porte [Clements96]. Na literatura, dificilmente um consenso é alcançado para a terminologia arquitetura de software, bem como para modelos e abordagens de projeto arquitetural. Partindo da descrição qualitativa dos sistemas organizacionais em uso, a arquitetura de software tem amadurecido para incorporar novas pesquisas sobre notações, ferramentas e abordagens. A arquitetura de software oferece hoje um guia concreto e complexo para projeto e desenvolvimento de software [Shaw00].

A utilização do conceito de arquitetura de sistemas representa o principal suporte para desenvolver e manter sistemas de software de alta durabilidade. A arquitetura de software define, através de um alto nível, o sistema em termos de componentes, a interação entre eles e os atributos e funcionalidades de cada componente [Sommerville01]. Uma definição da arquitetura nos dá uma clara perspectiva de todo o sistema e do controle necessário para seu desenvolvimento.

Neste capítulo apresentaremos os aspectos mais relevantes da arquitetura de software. Inicialmente mostraremos suas definições tradicionais e conceitos básicos. Em seguida teremos uma visão geral sobre o projeto arquitetural. Por fim, algumas considerações sobre a documentação e descrição dos modelos de arquitetura serão abordadas.

4.2 Definições de Arquitetura de Software

Não existe uma definição universal para arquitetura de software. Contudo, entre as mais referenciadas podemos citar:

- "A arquitetura de software de um programa ou sistema de computador é a estrutura ou estruturas do sistema que inclui componentes de software, as propriedades visíveis externamente desses componentes, e as relações entre eles. Por propriedades "visíveis externamente", nos referimos a suposição de que um componente pode fazer uso de outro componente, como seus serviços,

características de desempenho, tolerância a falha, uso de recurso compartilhado, e etc. “ [Bass98].

- “A estrutura dos componentes de um programa/sistema, suas inter-relações, e os princípios e as diretrizes que governam o projeto e evolução no tempo” [Garlan95].

- “Questões estruturais incluem a organização geral e estrutura global de controle; protocolos para comunicação, sincronização e acesso a dados; tarefa de funcionalidade para projetar elementos; distribuição física; composição de elementos de projeto; escala e desempenho; e seleção entre alternativas de projeto” [Shaw96].

Estas definições têm em comum a ênfase na arquitetura como uma descrição de um sistema, que é o resultado da soma de partes menores, e a forma como estas partes se relacionam e cooperam para executar o trabalho do sistema. Portanto, nesta dissertação adotaremos a definição de [Garlan95].

Vendo a arquitetura de software como uma estrutura de alto-nível de um sistema de software, teríamos as seguintes propriedades:

- Alto-nível de abstração suficiente para que o sistema possa ser visto como um todo.

- A estrutura deve suportar a funcionalidade requerida do sistema. Assim, o projeto da arquitetura deve levar em conta o comportamento dinâmico do sistema.

- A arquitetura tem que estar em conformidade com os requisitos de qualidades do sistema (requisitos não-funcionais). Esses requisitos incluem desempenho, segurança e requisito de confiança associado com a funcionalidade atual, como também flexibilidade ou extensibilidade associada com uma funcionalidade futura, adaptável a um custo razoável.

- No nível arquitetural todos os detalhes de implementação são escondidos.

A seguir serão vistos os conceitos básicos usados na definição da arquitetura de um sistema computacional.

4.3 Conceitos Básicos

Focalizando o conjunto central de conceitos da estrutura arquitetônica, a ontologia compartilhada apresenta cinco conceitos básicos, que incluem componentes, conectores, sistemas, propriedades e estilos [Shaw96][Garlan95]:

- *Componentes*: representam os elementos computacionais e os dados armazenados de um sistema. Intuitivamente, eles correspondem às caixas nas descrições gráficas de caixa-e-linha de arquiteturas de software. Exemplos típicos de componentes incluem os clientes, servidores, filtros e bancos de dados;

- *Portas*: cada porta define um ponto de interação entre um componente e seu ambiente. Um componente pode ter várias portas do mesmo tipo. Por exemplo, um servidor pode ter portas múltiplas de HTTP;

- *Conectores*: representam as interações entre componentes e correspondem às linhas em descrições gráficas de caixa-e-linha. Eles provêem o "contato" para projetos arquitetônicos, e então merecem tratamento explícito de modelagem. De uma perspectiva de execução, conectores mediam a comunicação e atividades de coordenação entre componentes. Exemplos incluem formas simples de interação, como *pipes*, chamada de procedimento, e propagação de evento. Conectores também podem representar interações complexas, como um protocolo de cliente-servidor ou uma ligação SQL entre um banco de dados e uma aplicação. Conectores têm interfaces que definem os papéis de cada participante na interação.

- *Sistemas*: representados pela combinação estruturada de componentes e conectores. Em geral, sistemas podem ser hierárquicos - componentes e conectores podem representar subsistema que têm suas próprias arquiteturas internas;

- *Propriedades:* representam informação adicional além de estrutura sobre as partes de uma descrição de arquitetura. Embora as propriedades que podem ser expressas por diferentes ADLs (*Architecture Description Language)* variem consideravelmente, tipicamente eles são usados para representar aspectos extra-funcionais antecipados ou requeridos de um projeto. Por exemplo, algumas ADLs

permitem calcular o processamento de sistema e latência baseado no desempenho estimado dos componentes e conectores. Em geral, é desejável poder associar propriedades com qualquer elemento arquitetural em uma descrição (componentes, conectores, sistemas, e suas interfaces). Por exemplo, uma interface (porta ou papel) pode ter uma propriedade de protocolo de interação associada;

- *Estilo da arquitetura*: representa uma família de sistemas relacionados, e tipicamente define um vocabulário de projeto para componente, conector, porta, papel, ligação, e tipo de propriedade, junto com regras de composição de tipos. Exemplos incluem arquiteturas com fluxo de dados baseados em grafos de *pipes-filter* e arquiteturas em camadas.

As relações entre esses elementos básicos da arquitetura serão vistas na próxima seção, com as definições e considerações relevantes para o projeto da arquitetura.

4.4 Projeto Arquitetural

A arquitetura de software é comumente definida em termos de componentes e conectores. Cada componente é identificado e recebe responsabilidades de componentes clientes, interagindo entre si através de *interfaces* definidas. Interconexões de componentes especificam a comunicação e os mecanismos de controle, e suporte as interações de componentes necessárias para garantir o comportamento do sistema [Garlan00a, Garlan00b].

Na definição da arquitetura do sistema deve-se observar o seguinte:

- A meta-arquitetura: O metamodelo com as visões da arquitetura, o estilo, os princípios, a chave de comunicação e mecanismos controle, bem como os padrões que guiam a equipe de arquitetos na criação de uma arquitetura;

- As visões arquiteturais: A arquitetura de Software, assim como na arquitetura tradicional, será melhor observada em termos do número de visões ou modelos complementares. Em particular, as visões estruturais ajudam a documentar e divulgar a arquitetura em termos de componentes e seus relacionamentos, e são

utilizadas na avaliação de requisitos de qualidade, como extensibilidade. Visões de comportamento são especialmente úteis na avaliação de requisitos de qualidades como desempenho e segurança. Visões de execução ajudam na avaliação de opções de distribuição física e na documentação e divulgação de decisões;

- Os padrões arquiteturais: Padrões estruturais tais como as camadas e cliente/servidor, e os mecanismos tais como conectores e portas;

- Os princípios chaves para projeto de arquitetura: Tais como abstração, separação de problemas, simplificação, adiamento de decisões, e técnicas relacionadas com encapsulamento;

- Os princípios de decomposição de sistemas e um bom projeto de interfaces.

A arquitetura de software provê uma descrição abstrata do sistema expondo certas propriedades, enquanto escondem outras. Idealmente, esta representação é considerada um guia intelectualmente tratável para todo o sistema, permitindo que os projetistas analisem sobre a habilidade do sistema na satisfação de certos requisitos e sugerir um plano de construção e composição do sistema [Garlan00a]. Os projetistas podem usar a estrutura planejada para estimar valores dos fluxos de dados de entrada, computar custos e capacidade de armazenamento, analisar sobre possibilidades de congestionamento dos dados, recursos de hardware software e ainda, cronograma de execução. A arquitetura de software atua como uma ponte entre os requisitos e a implementação.

A arquitetura de software pode assumir um importante papel em pelo menos seis aspectos no desenvolvimento de software:

1. Entendimento: Uma arquitetura simplifica a compreensão de grandes sistemas, por apresentá-lo em um nível de abstração no qual um projeto de sistema pode ser facilmente entendido [Shaw96];

2. Reuso: As descrições de arquitetura suportam reuso em níveis múltiplos. Os trabalhos correntes em reuso geralmente são focalizados em bibliotecas de

componentes. O projeto de arquitetura suporta reuso de grandes componentes e também de *frameworks* no qual os componentes podem ser integrados;

3. Construção: A descrição arquitetural provê um plano parcial para o desenvolvimento pela indicação dos componentes principais e dependências entre eles.

4. Evolução: Uma arquitetura de software pode expor as dimensões da evolução esperada para o sistema. O analista pode entender melhor a ramificação de uma mudança e estimar os custos de cada modificação de forma mais precisa. Além disso, as descrições arquiteturais separam os conceitos sobre a funcionalidade de um componente, da forma como este componente está conectado a outros componentes, fazendo uma distinção clara entre componentes e mecanismos de interação.

5. Análise: As descrições arquiteturais provêm novas oportunidades para analistas, incluindo checagem de consistência de sistema [Allen94], conformidade para restrições impostas pelo estilo de arquitetura, conformidade com os atributos de qualidades [Clements95], análise de dependência [Stafford98], e análise de domínio específico em arquiteturas com estilos específicos [Garlan94];

6. Gerenciamento: A avaliação crítica de uma arquitetura pode levar a um entendimento claro de requisitos, estratégias de implementação e riscos potenciais.

Para a análise da estrutura apropriada ao sistema, a arquitetura de software pode prover algumas abordagens para a representação e decomposição de aspectos arquiteturais, usando os conceitos de Visões e de Estilos.

4.4.1 Visões Arquiteturais

Visões diferentes são usadas para representar aspectos distintos de um sistema, cada visão pode prover um modelo de algum aspecto do sistema [IEEE00][Hilliard00]. Por exemplo, uma visão arquitetural poderia documentar a estrutura de um sistema como uma descrição de camadas na qual o componente representa agrupamentos lógicos de código, enquanto

outra visão poderia documentar a estrutura de um sistema em termos de sua configuração de execução na qual componentes representam processos em comunicação.

Visões diferentes ou modelos são úteis para propósitos diferentes. Decidir quais visões usar é um dos trabalhos principais de um arquiteto de software. Freqüentemente a escolha de visões dependerá fortemente das necessidades da análise de projeto. Por exemplo, um sistema de tempo-real, que tem restrição de programação, poderia usar uma ou mais visões que expõem limites de processos e tarefas, e indicar várias propriedades como periodicidade e requisitos de recursos. Um sistema mais centrado em compartilhamento de dados poderia dedicar visões a descrever a estrutura do espaço de dados e ao modo como componentes diferentes acessam os dados.

Há quatro classes típicas de visões que são requeridas por prover um conjunto razoável de documentação arquitetural [Garlan00a]:

- Visões baseadas em contexto: Estas visões indicam o ambiente na qual o sistema será construído, e freqüentemente identificam os elementos de domínio abstratos que determinam os requisitos globais do sistema e o contexto de negócio;
- Visões baseadas em código: Estas visões descrevem a estrutura do código, indicando como o sistema será construído e seus artefatos de implementação, como módulos, tabelas e classes. Tais visões são particularmente úteis como um guia para implementação e manutenção. Um especial, mas comum, caso de uma visão baseada em código é um diagrama em camada. Dividindo um sistema em camadas, o engenheiro pode melhorar a portabilidade e a modificabilidade do sistema.
- Visões de tempo de execução: Esses descrevem a estrutura do sistema em operação, indicando quais são as principais entidades de execução e como se comunicam. Visões de execução permitem análise sobre as propriedades de comportamento e atributos de qualidade, como consumo de recurso, desempenho, taxa de processamento, latência, confiança, etc.
- Visões baseadas em hardware: Estas visões descrevem a configuração física na qual o sistema será executado, indicando o número e tipos de processadores e ligações de

comunicação. A informação contida nestas visões é combinada com as visões de tempo de execução para derivar as propriedades de desempenho de sistema.

Na próxima seção apresentamos os aspectos e características dos estilos arquiteturais e destacamos alguns estilos existentes atua lmente.

4.5 Estilos Arquiteturais

Estilo de arquitetura é um padrão organizacional para componentes de software [Garlan95]. Estes padrões foram desenvolvidos ao longo dos anos quando os projetistas reconheceram o valor de princípios organizacionais e estruturas específicas para certas classes de software [Shaw96].

Software possui estilos organizacionais, que classicamente estão associados com denominações tais como um "sistema de cliente-servidor", um "sistema de *blackboard*", um "*pipeline*", um "*interpreter*", ou um "sistema de camadas". Alguns destes estilos são comumente associados com métodos de projetos específicos e notações, tais como, orientação a objetos e organização de fluxo de dados.

Uma classe importante de idiomas arquitetônicos constitui o que alguns pesquisadores têm chamado de "estilos de arquitetura". Um estilo de arquitetura caracteriza uma família de sistemas que estão relacionados através de propriedades estruturais e semânticas compartilhadas [Garlan94]. Mais especificamente, a definição de estilos provê quatro características [Monroe97]:

- O Vocabulário de elementos de projeto: Tipos de componentes e conectores como *pipes*, filtros, clientes, servidores, compiladores, bancos de dados;

- Regras de Projeto ou restrições: Que determinam as composições permitidas para os elementos. Por exemplo, as regras poderiam impedir ciclos em um estilo particular de *piper-filter*, especificar que uma organização cliente-servidor deve ser uma relação "n-para-um", ou definir um padrão de composição específico como de compilador *pipeline* de decomposição.

- Interpretação Semântica: Composições de elementos de projeto com significados bem definidos, e adequadamente restringidas pelas regras de projeto.

- Análise: Análise sobre o que pode ser executado nos sistemas construídos dentro do estilo. Exemplos incluem análise de horário de execução para um estilo orientado para processamento em tempo real e detecção de parada completa (*deadlock*) para envio de mensagem de cliente-servidor.

O uso de estilos de arquitetura proporciona benefícios significantes. Primeiro, a adoção de estilo permite a reutilização do projeto: soluções rotineiras com propriedades bem-definidas podem ser reaplicadas a problemas novos com segurança. Segundo, o uso de estilos pode conduzir a um significante reuso de código: Alguns aspectos constantes de um estilo são utilizados no compartilhamento de implementação. Terceiro, é mais fácil para outros entenderem a organização de um sistema se estruturas convencionais forem usadas. Por exemplo, a caracterização de um sistema como uma organização "cliente-servidor" define uma imagem dos tipos de componentes e como eles interagem. Quarto, o uso de estilos padronizados apóia a interoperabilidade. Exemplos incluem CORBA [OMG95] - arquiteturas orientadas a objeto, OSI [ISO84] – protocolo de pilha, e integração de ferramenta baseada em evento. Quinto, um estilo de arquitetura permite análise especializada e específica, por restringir o espaço de projeto ao estilo usado, tipo taxas de transferência, latência e condições de parada. Tais análises poderiam não ser significantes para uma arquitetura arbitrária (*ad hoc*) ou construída em um estilo diferente. Sexto, geralmente é possível (e desejável) prover representações gráficas estilo-específicas de elementos de projeto. Isto torna possível comparar a representação gráfica do projeto com as intuições domínio-específicas dos engenheiros e de como seus projetos serão visualizados.

Na próxima seção, apontaremos algumas técnicas e recomendações para documentar e descrever arquiteturas adequadamente.

4.6 Documentação e Descrição da Arquitetura

Documentação da arquitetura descreve a estrutura de um sistema através de uma ou mais visões. Cada qual identificando uma coleção de componentes de alto-nível e as relações entre esses componentes.

Um componente normalmente é documentado como algum tipo de objeto geométrico e representa uma unidade coerente de funcionalidade. A granularidade dos componentes dependerá do tipo de documentação que é desenvolvida: em algumas situações um componente pode ser tão grande quanto um subsistema principal; em outros pode ser tão pequeno quanto uma única classe de objeto ou agente. Tipicamente, componentes representam estruturas de sistema como módulos, elementos computacionais e processos em execução. As relações entre componentes são documentadas graficamente usando linhas ou adjacência. Estas relações indicam quais os aspectos de um componente que são usados por outros componentes, e como a comunicação inter-componentes é processada.

Nesta seção abordaremos, em particular, as práticas recomendadas para documentação arquitetural (o padrão IEEE 1471 [IEEE99]), o uso de linguagens de descrição de arquitetura (ADLs) e o uso da UML como linguagem de descrição de arquitetura.

4.6.1 Práticas Recomendadas

A documentação arquitetural descreve a estrutura de um sistema através de uma ou mais visões, cada qual identificando uma coleção de componentes de alto-nível e as relações entre esses componentes. Por prover uma descrição abstrata de um sistema, a arquitetura expõe certas propriedades e ajuda a garantir que os requisitos fundamentais sejam satisfeitos em áreas como desempenho, confiança, portabilidade, escalabilidade e interoperabilidade [Garlan00a]. Idealmente esta representação provê uma guia intelectualmente tratável do sistema global, permitindo que desenhistas argumentem sobre a habilidade do sistema para satisfazer certos requisitos, e sugestionando um desenho para construção e composição do sistema.

Embora a arquitetura tenha se tornado um termo popular na comunidade de computação, seu uso é inconsistente, e freqüentemente tem pouca semelhança com as origens do

conceito na engenharia civil. Apesar dessas inconsistências, há uma prática crescente e reconhecida dos conceitos de arquitetura para sistemas de computador. Em 2000, foi aprovado o padrão IEEE 1471 – *Recommended practice for architectural representation* [IEEE00], que documenta o consenso de como os padrões podem abordar questões conceituais e enfatizar as dificuldades associadas com a solução deste tipo de questão dentro de um processo padronizado de desenvolvimento.

O padrão IEEE 1471 define o termo "arquitetura" como "a organização fundamental de um sistema composto por seus componentes, as relações entre eles, a relação com o ambiente, e os princípios que guiam seu projeto e evolução" [Maier01]. Esta definição incorpora a idéia que há uma diferença entre uma descrição arquitetural e uma arquitetura. Uma descrição arquitetural é um artefato concreto, mas uma arquitetura é um conceito de um sistema.

O padrão também coloca restrições normativas nas descrições arquiteturais, que são os documentos que descrevem a arquitetura de um sistema. Esses documentos definem a equivalência entre a representação gráfica e as convenções de simbologia usada. Os elementos mais importantes são [Hilliard00]:

- Um conjunto normativo de definições para termos, que incluem descrição, visão e ponto de vista arquitetural;
- Um *framework* conceitual que estabelece estes termos no contexto de vários usos para descrições arquiteturais na construção, análise e evolução de sistema; e,
- Um conjunto de requisitos da descrição arquitetural de um sistema.

Uma descrição de arquitetura tem que justificar as decisões arquiteturais adotadas. Isto pode levar a forma de apresentações das alternativas que foram consideradas pelos arquitetos, das alternativas excluídas, ou mesmo outras análises que os levaram a seleção da arquitetura que está documentada.

4.6.2 Linguagens para Descrição de Arquitetura

A informalidade de projeto de arquitetura provoca um número considerável de problemas, já que os diagramas informais não podem ser analisados formalmente para consistência,

completude ou exatidão. As restrições arquiteturais assumidas no projeto inicial não podem ser forçadas no desenvolvimento do sistema. Diante destes problemas surgiu a necessidade de uma notação formal para projeto de arquitetura.

As linguagens para descrição de arquitetura (ADL – *Architecture Description Language*), são notações providas de uma definição conceitual e sintaxe concreta para caracterizar uma arquitetura de software [Clementes95][Garlan00a][Medvidovic97][Shaw00].

Apesar de existirem várias linguagens voltadas para o projeto da arquitetura, cada uma provê algumas capacidades específicas e distintas: Aesop [Garlan94] suporta o uso de estilos de arquitetura; C2 [Medvidovic99b] suporta a descrição de interfaces de usuários usando um estilo baseado em eventos; Darwin [Magee95] suporta a análise de troca de mensagens em sistemas distribuídos; Rapide [Luckham95] permite simulação de projeto arquitetural, e têm ferramentas para analisar os resultados de cada simulação; SADL [Moriconi95] prover a base formal para refinamento de arquitetura; UniCon [Shaw95] tem um compilador de alto nível para projeto de arquitetura que suporta uma mistura de componentes heterogêneos e tipos de conectores; Wright [Allen97] suporta a especificação formal e análise de interação entre componentes de arquitetura.

Recentemente a proliferação das capacidades das ADLs tem incentivado a investigação de formas para integrar notação e ferramentas baseadas em arquitetura. Um dos resultados é a linguagem para intercâmbio arquitetural, chamada de ACME [Garlan97], que provê um framework para descrição de estrutura arquitetural e um mecanismo de anotação para adicionar semântica para a estrutura.

4.6.3 Representação Arquitetural em UML

Há um considerável interesse em usar uma notação de propósito geral para modelagem de arquitetura. Recentemente um grande número de propostas tem tentado mostrar como os conceitos encontrados em ADLs podem ser mapeados diretamente em uma notação orientada a objetos, como a UML – *Unified Modeling Language* [Garlan99][Hofmeister99][Medvidovic99a].

Como linguagem genérica de modelagem, a UML oferece uma notação familiar para projetistas, além de permitir uma ligação direta entre implementação orientada a objetos e ferramentas de desenvolvimento. Entretanto, uma linguagem de objetos de propósito geral tem o problema do vocabulário conceitual de objetos, que pode não ser ideal para representar conceitos arquiteturais.

Visões em UML podem capturar aspectos estruturais e comportamentais do desenvolvimento de software. Visões estruturais utilizam classes, pacotes e casos de uso. Por sua vez, as visões comportamentais são representadas por cenários, estados e atividades [Hilliard99].

Adicionalmente, UML suporta a necessidade de refinamento e de expansão de suas especificações através de três mecanismos de extensão [Rumbaugh99]:

1. *Restrições:* colocam restrições semânticas em elementos de projeto de elementos. UML usa OCL (*Object Constraint Language*) [OCL97] para definir as restrições;

2. *Tagged values:* permitem acrescentar atributos novos a elementos particulares do modelo;

3. *Estereótipos:* permitem que grupos de "restrições" e "*tagged values*" recebam nomes descritivos e que sejam aplicados a outros elementos do modelo; o efeito semântico é como se as restrições e valores sejam diretamente aplicados a esses elementos.

Um estereótipo pode introduzir valores adicionais, restrições adicionais e uma nova representação gráfica, mas a maneira na qual esta informação será documentada não está definida pela UML [Hilliard99]. O papel típico da extensão é adaptar a UML para uso em uma área de aplicação em particular, para permitir o uso com uma metodologia ou com um processo em particular, ou para acomodar uma tecnologia nova. Como resultado, estes mecanismos tendem a ser usados de forma *ad hoc*, isto é, de forma não sistematizada.

Na fase de construção do software, uma descrição arquitetural provê um planejamento parcial para a equipe de desenvolvimento, indicando os componentes principais e as

dependências entre eles [Garlan00a]. Por exemplo, uma visão em camada de uma arquitetura documenta os limites de abstração entre as partes da implementação do sistema, identificando claramente as principais conexões internas, e restringindo as partes de serviços confiáveis providos por outras partes componentes.

Organizar os sistemas em camadas é um dos modos mais comuns de lidar com complexidade, e apesar disso, nenhuma das linguagens padrões usadas na indústria provê o conceito de "camada" como um conceito de primeira classe.

Em UML há várias formas de expressar arquitetura em camadas. A primeira forma é usando os mecanismos de extensão para representar arquitetura em camada, ou visões [Hilliard01b][Medvidovic00][Medvidovic99a]. Por exemplo, pode-se criar um pacote de funções entrada com estereótipo <<camada1>>, um pacote de funções acesso a dados com estereótipo <<camada2>>, e assim por diante. A OCL pode ser usada para restringir o modo como cada camada (pacote estereotipado) pode interagir. Portanto, a OCL pode especificar que a Camada1 pode ser dependente da Camada2, enquanto a Camada2 pode ser dependente da Camada3. Tendo especificado como representar as camadas, a ferramenta de projeto de UML pode ser usada para criar diagramas de arquitetura em camadas.

Outra possibilidade de modelagem é utilizar a abordagem de representação arquitetural que usa a UML como uma linguagem de modelagem de arquitetura para sistemas de tempo real, chamada de UML-RT [Selic98][Selic99]. Nessa abordagem há uma forte ênfase no uso de estereótipos e diagramas de colaboração para representar explicitamente a interconexão entre as entidades arquiteturais. A especificação completa da estrutura complexa de sistemas em tempo real é obtida através de combinação de classes e diagramas de colaboração usando a definição de três construtores principais, que são cápsulas, portas e conectores. Esta abordagem também inclui camadas, usando conexões independentes através das quais um componente provê seu serviço.

A UML-RT propõe os seguintes conceitos:

- Cápsulas: Uma cápsula é um estereótipo do conceito de classe da UML com algumas características específicas. Uma cápsula usa suas portas para todas as interações com seu ambiente. A comunicação com outras cápsulas é feita por uma ou mais portas. A interconexão com outras cápsulas é via conectores usando sinais. Uma cápsula é uma classe ativa especializada e é usada para modelar um componente auto-contido de um sistema. Por exemplo, uma cápsula pode ser usada para capturar um subsistema inteiro, ou até mesmo um sistema completo.

- Portas: Uma porta representa um ponto de interação entre uma cápsula e seu ambiente. Elas transportam sinais entre o ambiente e a cápsula. O tipo de sinais e a ordem que eles podem aparecer são definidos pelo protocolo associado à porta. A notação de porta é mostrada como um pequeno símbolo quadrado (Figura 4.3). Se o símbolo da porta é colocado ultrapassando o limite do símbolo do retângulo denota uma visibilidade pública. Se a porta é mostrada dentro do símbolo do retângulo, então a porta está escondida e sua visibilidade é privada. Quando vistas de dentro da cápsula, portas podem ser de dois tipos: portas de repasse e portas finais. Portas de repasse são simplesmente portas por onde passam todos os sinais e portas finais são as últimas fontes e tanques de todos os sinais enviados pelas cápsulas. Estes sinais são gerados pelas máquinas de estado das cápsulas.

- Protocolos: Um protocolo especifica um conjunto de comportamentos válidos (troca de sinais) entre duas ou mais cápsulas colaboradoras. Entretanto, para fazer tal padrão dinâmico reusável, protocolos são desacoplados de um contexto particular de cápsulas colaboradoras e são definidas ao invés em termos de entidades abstratas chamadas de papéis do protocolo (estereótipo de *Classifier Role* em UML) (Figura 4.2).

- Conectores: Um conector é uma abstração de um canal de passagem de mensagem que conecta duas ou mais portas. Cada conector é tipado por um protocolo que define as possíveis interações que podem acontecer através deste conector (Figura 4.1 e Figura 4.3)

Observe que algumas novas capacidades conceituais para modelagem de arquitetura que podem ser usadas para expressar estruturas complexas, incluindo cápsulas (classificadores

compostos), portas e conectores estão incluídos na recente proposta submetida em resposta a UML 2.0 *Superstructure* RPF [OMG].

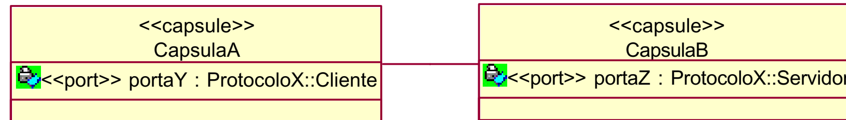


Figura 4.1 - Um diagrama de classes de Cápsula

Uma forma mais compacta para descrever cápsulas é ilustrada na Figura 4.1, onde as portas de uma cápsula são listadas em uma lista especial rotulada. O papel do protocolo (tipo) que uma porta exerce é normalmente identificado por um *pathname*, já que os nomes dos papéis do protocolo (*protocolRoles*) são únicos apenas no escopo de um protocolo dado. Por exemplo, portaY : ProtocoloX :: Cliente, indica que a porta *portaY* exerce ao papel *Cliente* no protocolo *ProtocoloX*.

Um diagrama de classes de Cápsula também pode ser descrito com mais detalhes, explicitando os protocolos entre as cápsulas, bem como as portas que compõem as cápsulas (Figura 4.2).

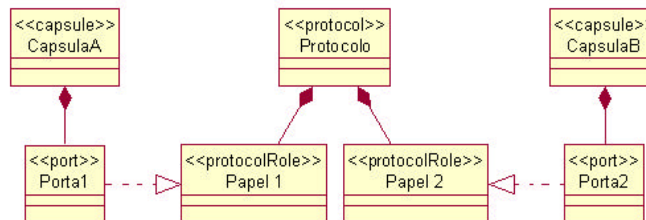


Figura 4.2 - Um diagrama de classes de Cápsula detalhado

Entretanto, portas também são representadas nos diagramas de colaboração (Figura 4.3) que descrevem a decomposição interna de uma cápsula. Nestes diagramas, portas são representadas pelos papéis classificadores apropriados, i.e., os papéis das portas. Para reduzir a confusão visual, os papéis das portas são geralmente mostrados na forma de ícone. Para o caso de protocolos binários, um estereótipo adicional de ícone pode ser usado: a porta exercendo o papel conjugado (o papel de *cliente*) é indicada por um quadrado

Uma afirmação comum é que projeto arquitetural deveria suportar as qualidades de software almejadas, tais como robustez, adaptabilidade, reusabilidade e manutenibilidade [Bass98]. As arquiteturas de software incluem decisões de projeto iniciais e embutem a estrutura global que impacta na qualidade do sistema inteiro. Uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito *não-funcional* [Sommerville01]. A presença de uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais que possam garantir a preservação dessa propriedade durante o desenvolvimento do sistema [Monroe97][Shaw96].

Nesta dissertação, usaremos a UML-RT como linguagem de descrição arquitetural na nossa abordagem, pois ela permite construir modelos arquiteturais mais detalhados, além de ser uma notação industrial que dispõe de um amplo suporte ferramental e tem o potencial de se tornar bastante popular.

O próximo capítulo apresenta vários estilos arquiteturais voltados para sistemas de software complexos e de grande porte, tal como sistemas multi-agentes. São estilos baseados na teoria organizacional que focam em requisitos não-funcionais. Serão enfocadas as deficiências atualmente presentes tanto na descrição destes estilos quanto nos modelos gerados pelo projeto arquitetural a partir destes estilos. Neste sentido será apresentada uma proposta para solucionar de forma adequada tais deficiências.

Capítulo 5 - Estilos Arquiteturais Organizacionais em UML

Este capítulo pretende detalhar a principal contribuição desta dissertação, o uso da UML Real Time como linguagem de descrição arquitetural na metodologia de desenvolvimento orientado a agentes Tropos.

5.1 Introdução

A proposta Tropos [Castro02] para desenvolvimento orientado a agentes consiste de cinco fases (vide seção 3.5). Nesta dissertação, em particular, enfocaremos a sua fase de projeto arquitetural, que atualmente utiliza somente a notação i* [Yu95] para representar seus modelos arquiteturais. Infelizmente, esta notação não é amplamente usada pelos profissionais de software, já que ela está apenas começando a ser reconhecida como sendo adequada para representar requisitos e seu suporte ferramental (*Organizational Modeling Environment* [Liu03]) é limitado. Além disso, ela não é adequada para representar algumas informações detalhadas que algumas vezes são requeridas no projeto arquitetural, tal como o conjunto de sinais que são trocados entre os componentes arquiteturais, além da seqüência válida destes sinais (protocolo) [Silva02][Silva03][Castro03].

Por outro lado, a *Unified Modeling Language* (UML) [Rumbaugh99] vem sendo largamente aceita na indústria como uma linguagem padrão de modelos para software. Como uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de um sistema de software, a UML tem se mostrado de grande valor ao ajudar organizações no gerenciamento da complexidade de seus sistemas. Os poderosos mecanismos de extensão da UML nos permitem representar novos conceitos em UML, além de capturar e representar suficientemente várias visões através do uso do meta-modelo UML. Além disto, a UML vem sendo estendida tanto para dar suporte ao desenvolvimento orientado a agentes (vide capítulo 3), quanto para ser usada como uma linguagem de descrição arquitetural (vide capítulo 4) para representar arquitetura de software. Portanto, podemos obter modelos arquiteturais nos quais descrevemos elementos de alto-nível do projeto do sistema e seus conectores. Além disso, a UML permite suportar diferentes pontos de vista do sistema em construção e é apoiada por uma grande quantidade de provedores de ferramentas.

Nesta dissertação propomos o uso de uma extensão da UML para acomodar os conceitos e características atualmente usadas para representar as arquiteturas organizacionais definidas pelo Tropos. Tal extensão, a UML para Sistemas em Tempo Real (UML-RT)

[Selic98][Selic99][OMG], está sendo usada para modelar arquiteturas de software (vide seção 4.5.3). Nosso objetivo é prover uma representação mais detalhada dos modelos produzidos na fase arquitetural da metodologia Tropos, além de representar os seus estilos arquiteturais organizacionais em uma notação industrial popular, a fim de torná-los conhecidos e usados por outras metodologias de Engenharia de Software [Silva02][Silva03][Castro03]. Neste capítulo nós teremos uma visão mais detalhada da fase de projeto arquitetural da metodologia Tropos, a fim de entendermos seus principais conceitos e estilos arquiteturais. Em seguida, mostraremos como os conceitos usados atualmente para representar arquiteturas organizacionais em Tropos são capturados e representados usando os elementos da UML-RT. Por fim, apresentaremos a nossa proposta de representação dos estilos arquiteturais organizacionais utilizando a UML-RT.

5.2 A Fase de Projeto Arquitetural da Metodologia Tropos

Atualmente, software tem que ser baseado em arquiteturas que possam evoluir e mudar continuamente para acomodar novos componentes e conhecer novos requisitos. Conforme vimos no Capítulo 4, arquiteturas de software descrevem um software num nível macroscópico em termos de um número gerenciável de subsistemas/componentes/módulos inter-relacionados através de dependências de dado e controle. Entretanto, arquitetura é mais do que apenas estrutura, ela também define comportamento. Uma arquitetura flexível com componentes fracamente acoplados é muito mais fácil de acomodar requisitos de novas características do que aquelas que foram altamente otimizadas apenas para seu conjunto inicial de requisitos. Infelizmente, os estilos arquiteturais clássicos [Shaw96] e os estilos para aplicações *e-business* [Conallen00][IBM01] não focam em processos de negócio nem em requisitos não-funcionais (RNFs) da aplicação. Como um resultado, a estrutura organizacional não é descrita e nem a perspectiva conceitual de alto-nível da aplicação.

Tropos definiu os estilos arquiteturais organizacionais [Kolp01a][Kolp01b][Kolp02] para aplicações multi-agente, cooperativas, dinâmicas e distribuídas para guiar o projeto da arquitetura do sistema. Estes estilos arquiteturais *(Pirâmide, União Estratégica, Estrutura em 5, Tomada de Controle, Alavanca, Integração Vertical, Apropriação, Oferta,*

Estrutura Plana, Contratação Hierárquica) são baseados em conceitos e alternativas de projeto vindas da pesquisa na teoria organizacional [Mintzberg92] [Scott98] e alianças estratégicas [Gomes96][Segil96][Yoshino95].

Por exemplo, o estilo arquitetural *União Estratégica* (*Joint-Venture*) envolve acordo entre dois ou mais parceiros principais para obter benefícios derivados da operação em uma larga escala, com investimentos limitados e custos de manutenção mais baixos (Figura 5.1). É feita a delegação de autoridade a um ator específico que será responsável pelo gerenciamento comum coordenando tarefas e operações e gerenciando o compartilhamento de conhecimento e recursos. Observe que os parceiros perseguem objetivos comuns. Cada parceiro principal pode se gerenciar e controlar-se numa dimensão local e interagir diretamente como outros parceiros principais para trocar, prover e receber serviços, dados e conhecimento. Entretanto, em uma dimensão global, a operação e coordenação estratégica de tal sistema e dos atores parceiros no sistema apenas são garantidas pelo gerente comum. Parceiros secundários fornecem serviços e suportam tarefas para o núcleo da organização. Seria bastante interessante sermos capazes de documentar tais estilos organizacionais em notações gráficas tais como UML [Rumbaugh99], além da notação i* [Yu95] (Figura 5.1).

Vimos na seção 3.6 que Tropos adota os conceitos oferecidos pelo framework i* para suportar a modelagem e a análise durante suas fases iniciais. Isto significa que tanto o ambiente do sistema quanto o sistema em si são vistos com organizações de atores, cada um tendo metas a serem cumpridas e dependendo da ajuda de outros atores no cumprimento destas metas.

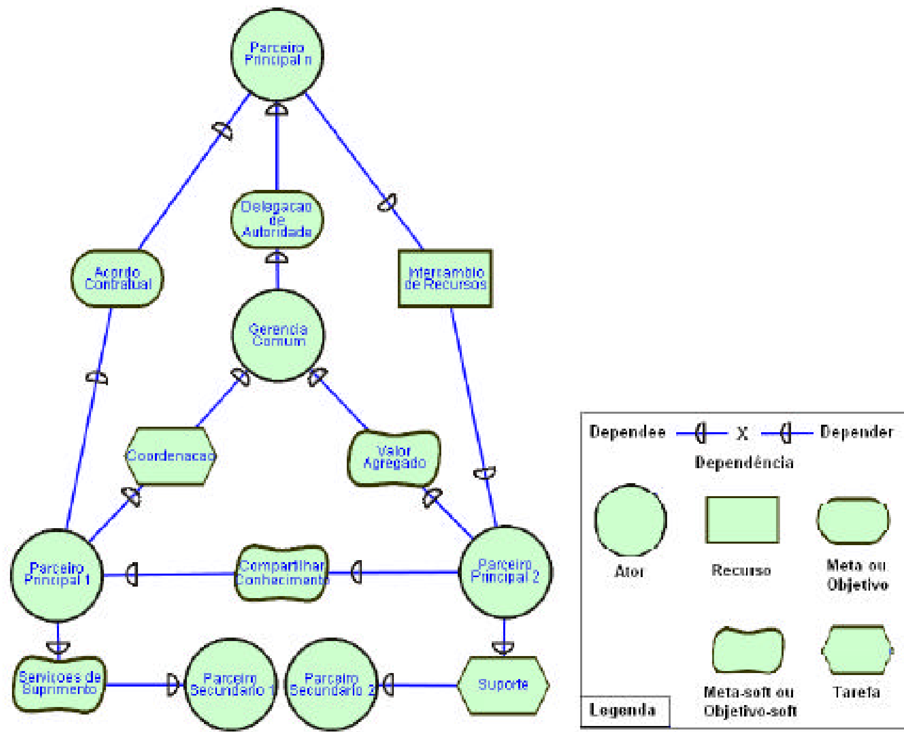


Figura 5.1 - União Estratégica (*Joint-Venture*)

No momento, a metodologia Tropos representa atores como círculos, conforme a Figura 5.1; *dependums* – metas, metas-soft, tarefas e recursos – são respectivamente representados como ovais, nuvens, hexágonos e retângulos; e dependências têm a forma *depender? dependum*? *depende*e. A seguir explicamos os conceitos encontrados no Tropos:

- Ator: Um ator é uma entidade ativa que executa ações para atingir metas exercitando sua base de conhecimento. Por exemplo, na Figura 5.1 encontramos o ator Gerência Comum.

- Dependência: descreve um relacionamento intencional entre dois atores, i.e., um "acordo" (chamado dependum) entre dois atores: o depender e o dependee, onde um ator (depender) depende de um outro ator (dependee) sobre alguma coisa (dependum).

- Depender: O depender é o ator dependente. Por exemplo, na Figura 5.1 encontramos o ator Parceiro Principal 1 sendo o depender da dependência Coordenação.
- Dependee: O dependee é o ator de quem se é dependido. Portanto, o dependee da dependência Coordenação, ilustrada na Figura 5.1, é o ator Gerencia Comum.
- Dependum: O dependum é o tipo da dependência e descreve a natureza do acordo.
- Meta ou Objetivo: Uma meta é uma condição ou situação no mundo que o ator gostaria de atingir. Como a meta vai ser atingida não está especificada, permitindo que alternativas sejam consideradas. Dependências de meta são usadas para representar delegação de responsabilidade para o cumprimento de uma meta. Na Figura 5.1, encontramos como exemplo a meta Acordo Contratual caracterizando a dependência entre os atores Parceiro Principal 1 e Parceiro Principal n.
- Meta-soft ou Objetivo-soft: Uma meta-soft é uma condição ou situação no mundo que o ator gostaria de atingir, mas diferente do conceito de meta, não há um critério claramente definido de como a condição é atingida e depende do julgamento subjetivo e interpretação do projetista para julgar se um estado de casos particular de fato atinge suficientemente a meta-soft estabelecida. Dependências de meta-soft são similares às dependências de meta, mas seu cumprimento não pode ser definido precisamente (por exemplo, o julgamento é subjetivo, ou o cumprimento pode ocorrer apenas em uma dada proporção). Como exemplo, observamos na Figura 5.1 a dependência de meta-soft Compartilhar Conhecimento entre os atores Parceiro Principal 2 e Parceiro Principal 1.
- Recurso: Um recurso é uma entidade (física ou de informação), com a qual a principal preocupação é se ela está disponível. Dependências de recurso requerem que o dependee forneça um recurso para o depender. Por exemplo, a dependência de recurso Intercâmbio de Recursos entre os atores Parceiro Principal n e Parceiro Principal 2 foi ilustrada na Figura 5.1.
- Tarefa: Uma tarefa especifica uma forma particular de fazer algo. Tarefas podem ser vistas como as soluções no sistema alvo, que irão satisfazer as metas-soft

(operacionalizações). Estas soluções provêm operações, processos, representações de dados, estruturação, restrições e agentes no sistema alvo para conhecer, as necessidades estabelecidas nas metas e metas-soft. Dependências de tarefa são usadas em situações onde o dependee é requisitado a executar uma dada atividade. Temos como exemplo a dependência Coordenação entre os atores Parceiro Principal 1 e Gerencia Comum, na Figura 5.1.

Portanto, atualmente Tropos utiliza a notação do i* para representar os seus estilos arquiteturais. O estilo *Estrutura em 5* (*Structure in 5*), que consiste dos típicos componentes estratégicos e logísticos geralmente encontrados em várias organizações (Figura 5.2). No nível de base encontra-se o *Núcleo Operacional* onde as tarefas e operações básicas – os procedimentos de entrada, processamento, saída e apoio direto associados com a execução do sistema – são executados. No topo da organização está o componente *Cúpula* composto de atores executivos estratégicos. Abaixo dele, situam-se os componentes de logística, controle/padronização e gerência respectivamente *Suporte, Coordenação* e *Agência Intermediária.* O componente *Coordenação* executa as tarefas de padronizar o comportamento de outros componentes, em adição a aplicar procedimentos analíticos para ajudar o sistema a adaptar-se a seu ambiente. O componente *Suporte* dá assistência ao *Núcleo Operacional* para serviços não operacionais que estão fora do fluxo básico de tarefas e procedimentos operacionais. Atores que se juntam da *Cúpula* estratégica ao *Núcleo Operacional* caracterizam a *Agência Intermediária*.

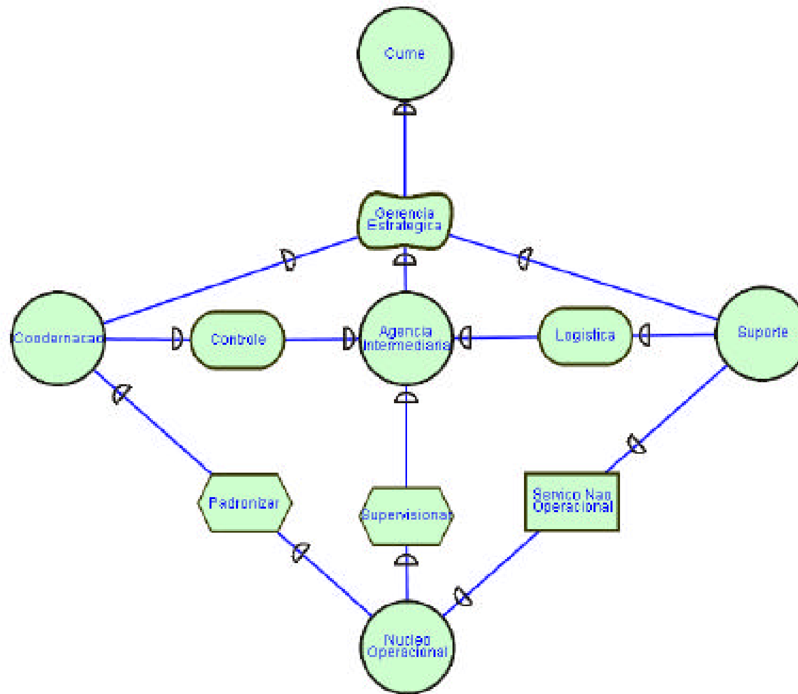


Figura 5.2 - Estrutura em 5 (*Structure in 5*)

O estilo *Integração Vertical (Vertical Integration)* consiste de atores comprometidos em atingir metas ou realizar tarefas relacionadas em estágios diferentes de um processo de produção (Figura 5.3). Um componente *Organizador* mistura e sincroniza interações/dependências entre participantes, que agem como intermediários. Por exemplo, na Figura 5.3 apresentamos um estilo de integração vertical para o domínio de distribuição de bens. Espera-se que o *Provedor* forneça produtos de qualidade, o *Atacadista* seja responsável por garantir o fornecimento de grande quantidade de produto, enquanto que o *Varejista* cuida da entrega direta para os *Consumidores*. Observe que este estilo é adequado para aplicações que solicitam uma série definida de computações independentes a serem executadas em dados ordenados, podendo nesta ótica ser visto como uma especialização do estilo arquitetural clássico *Pipers & Filters* [Shaw96].

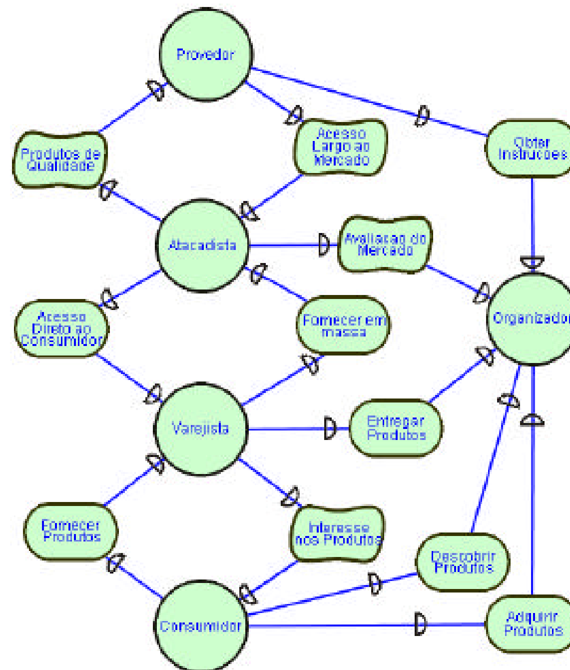


Figura 5.3 - Integração Vertical (*Vertical Integration*)

O estilo *Apropriação (Co-optation)*, como o próprio nome diz, envolve a incorporação de agentes externos na estrutura ou no comportamento tomador de decisão ou conselheiro de uma organização iniciante (Figura 5.4). Ao apropriar-se de agentes de sistemas externos, organizações estão, em efeito, negociado confiança e autoridade sobre recurso, capital de conhecimento e suporte. O sistema contratante tem que decidir com os contratados o que está sendo feito em seu benefício; e cada ator incorporado tem que reconciliar e ajustar suas próprias visões com a política do sistema que ele tem que se comunicar.

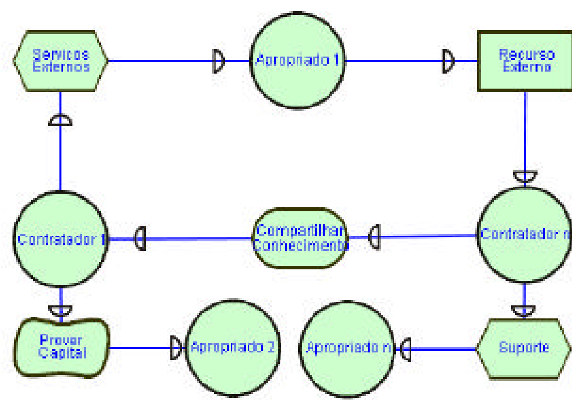


Figura 5.4 - Apropriação (*Co-optation*)

O estilo *Comprimento de Braço (Arm's Length)* implica em acordos entre atores independentes e competitivos, porém parceiros (Figura 5.5). Os parceiros mantêm sua autonomia e independência, mas agem e colocam seus recursos e conhecimentos juntos para cumprir metas precisas comuns. Nenhuma autoridade é delegada ou perdida de um colaborador para outro. Já que este estilo é adequado para aplicações que envolvem uma coleção de computações distintas e largamente independentes cuja execução deveria prosseguir competitivamente, ele pode ser considerado uma derivação do estilo arquitetural clássico processos de comunicação [Shaw96].

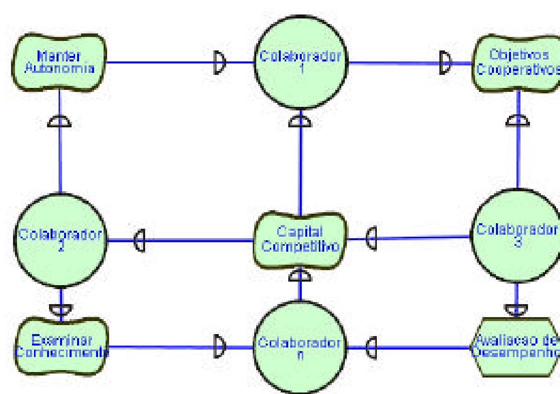


Figura 5.5 - Comprimento de Braço (*Arm's Length*)

O estilo *Tomada de Controle (Takeover)* envolve a delegação total de autoridade e gerenciamento de dois ou mais parceiros para um único ator *Tomador de Controle* coletivo (Figura 5.6). É semelhante em muitas formas ao estilo *União Estratégica* (Figura

5.1). A diferença maior e crucial é que enquanto em uma *União Estratégica* identidade e autonomias das unidades separadas são preservadas, o *Tomador de Controle* monopoliza estas unidades críticas no sentido de que nenhum relacionamento, dependência ou comunicações diretas são tolerados, exceto aquelas envolvendo o *Tomador de Controle.*

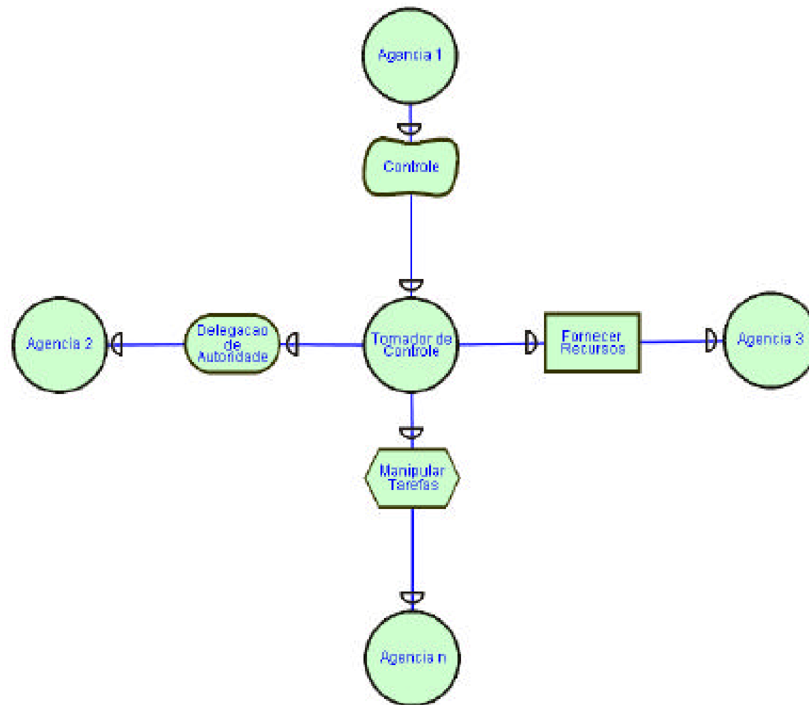


Figura 5.6 - Tomada de Controle (*Takeover*)

O estilo *Pirâmide (Pyramid)* é uma bem conhecida estrutura de autoridade hierárquica exercida com limites organizacionais (Figura 5.7). Os atores nos níveis mais baixos dependem daqueles nos níveis mais altos. O mecanismo crucial é a supervisão direta a partir da *Cúpula*. Gerentes e supervisores são, portanto, apenas atores intermediários direcionando decisões estratégicas e autoridade da *Cúpula* para o nível de operação. Eles podem coordenar comportamentos e tomar decisões por si próprios, mas apenas em um nível local. Este estilo pode ser aplicado quando na distribuição de sistemas multi-agentes simples. Além disso, ele suporta dinamicidade já que mecanismos de coordenação e decisão são imediatamente identificáveis de forma direta e não complexa. A capacidade de evoluir e de se modificar pode então ser implementada em termos deste estilo a custos

baixos. Por outro lado, ele não é adequado para sistemas multi-agentes de grande porte que requerem muitos tipos de agentes. Entretanto, este estilo poderá ser utilizado para gerenciar e resolver situações de crise. Por exemplo, um sistema multi-agente complexo frente a uma intrusão não autorizada de agentes externos e não confiáveis poderia dinamicamente, para um tempo longo ou curto, decidir se migrar para uma organização piramidal para se capacitar a resolver o problema de segurança de uma forma mais eficiente. Quando este estilo for considerado para aplicações em que a computação possa ser apropriadamente definida via uma hierarquia de definições de procedimento, ele também pode ser relacionado ao estilo arquitetural clássico *Programa Principal e Sub-rotinas* [Shaw96].

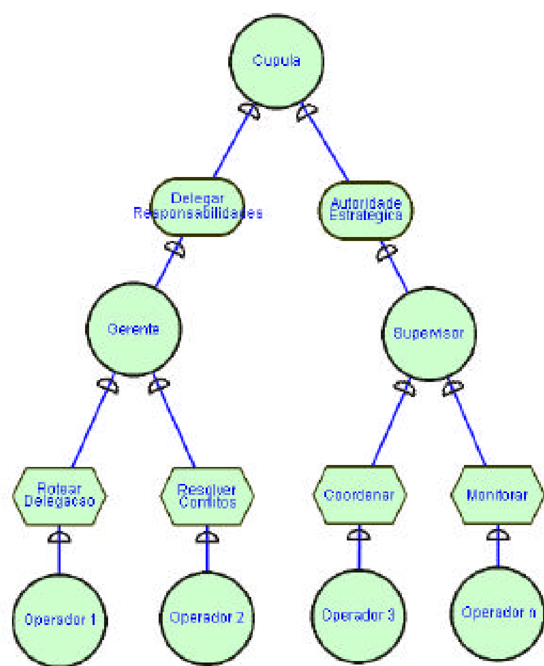


Figura 5.7 - Pirâmide (*Pyramid*)

O estilo *Contratação Hierárquica (Hierarchical Contracting)* identifica mecanismos de coordenação que combinam características do acordo *Comprimento de Braço* com aspectos de autoridade do estilo *Pirâmide* (Figura 5.8). Os mecanismos de coordenação desenvolvidos para caracterizar o *Comprimento de Braço* (independente) envolvem uma variedade de negociadores, mediadores e observadores em diferentes níveis manipulando

cláusulas condicionais para monitorar e gerenciar possíveis contingências, negociar e resolver conflitos e finalmente deliberar e tomar decisões. Relacionamentos hierárquicos, da Cúpula Executiva para os contratadores, restringem autonomia e destacam um risco cooperativo entre as partes contratantes. Tal arranjo contratual duplo (e admitidamente complexo) pode ser usado para gerenciar condições de complexidade e incerteza distribuída em aplicações de alto-custo-alto-benefício (alto-risco). Já que este estilo é adequado para aplicações que envolvem distintas classes de serviços em camada, que podem ser arranjados hierarquicamente, ele pode ser considerado uma especialização do estilo clássico de arquitetura em camadas [Shaw96].

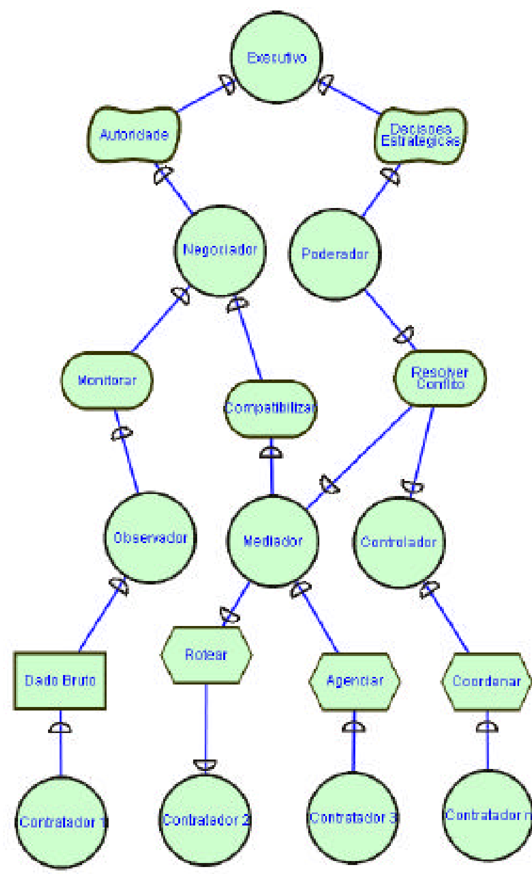


Figura 5.8 - Contratação Hierárquica (*Hierarchical Contracting*)

O estilo *Oferta (Bidding)* envolve mecanismos de competitividade e os atores se comportam com se eles estivessem tomando parte em um leilão (Figura 5.9). O ator

preenchido de branco (versus um preenchido de preto). Neste caso, o nome do protocolo e o sufixo til são suficientes para identificar o papel do protocolo como o papel conjugado; o nome do papel do protocolo é redundante e poderia ser omitido. De forma similar, o uso do nome do protocolo sozinho em um quadrado preto indica o papel base (papel de *servidor*) do protocolo. Na Figura 4.3, nós podemos ver os detalhes de (dentro) da cápsula e a distinção entre a porta de repassagem (portas Porta1 e Porta2) e a porta final é indicada graficamente como um quadrado (portas Porta1 e Porta2) e um quadrado associado a uma máquina de estado (portas Porta3 e Porta4), respectivamente.

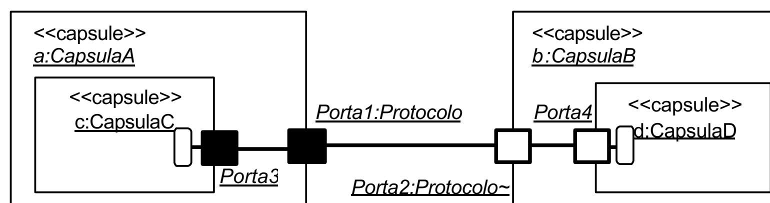


Figura 4.3 - Um diagrama de colaboração de Cápsulas

A UML-RT permite incorporar mais detalhes ao projeto arquitetural, tanto na estrutura da arquitetura (através do refinamento de cápsulas), como também no seu comportamento (através dos protocolos de interação entre as cápsulas).

4.7 Considerações Finais

A arquitetura de software é um dos campos em considerável crescimento através dos últimos anos, e promete um contínuo crescimento para a próxima década. A maturidade do projeto arquitetural dentro da disciplina de engenharia de software já é universalmente reconhecida e praticada. Há ainda um grande número de novas tendências e desafios significantes que precisam ser explorados e pesquisados [Shaw00]. Muitas das soluções para esses desafios surgem como uma conseqüência natural da disseminação e maturação das práticas arquiteturais e da tecnologia disponível. Outros desafios surgem por conta das mudanças nos processos computacionais e nas necessidades de software, que requerem novas e significantes inovações [Garlan00a].

Leiloeiro faz a mostra, anuncia o leilão exposto pelo *Expositor* do leilão, recebe ofertas dos atores *Comprador* e garante a comunicação e o *feedback* com o *Expositor* do leilão. O Leiloeiro deve ser um ator do sistema que meramente organiza e opera o leilão e seus mecanismos. Ele também pode ser um dos compradores (por exemplo, vendendo um item que todos os outros compradores estão interessados). O *Expositor* do leilão é responsável por expor a oferta. Este estilo implica em rápido tempo de resposta e adaptabilidade para o sistema.

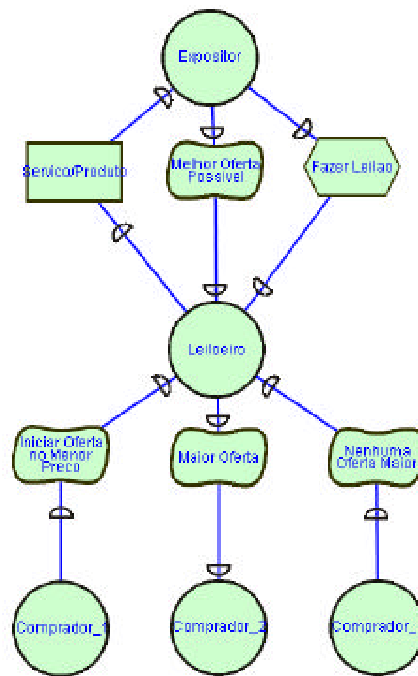


Figura 5.9 - Oferta (*Bidding*)

O estilo *Estrutura Plana (Flat Structure)* não possui estrutura fixa e assume que nenhum ator tenha controle sobre outro (Figura 5.10). A principal vantagem desta arquitetura é que ela suporta autonomia, distribuição e evolução contínua da arquitetura de um ator. Entretanto, a desvantagem chave é que esta arquitetura requer uma grande quantidade de raciocínio e comunicação por cada ator participante.

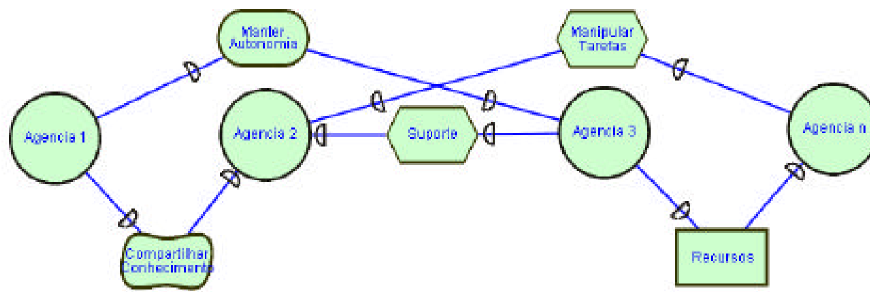


Figura 5.10 - Estrutura Plana (*Flat Structure*)

Os estilos organizacionais são estruturas genéricas definidas em um meta-nível que podem ser instanciados para projetar a arquitetura de uma aplicação específica. Devido à natureza organizacional e intencional dos sistemas multi-agentes, os estilos organizacionais têm sido avaliados e comparados usando atributos de qualidade de software identificados por arquiteturas envolvendo componentes coordenados autônomos tais como *previsibilidade*, *segurança*, *adaptabilidade*, *habilidade de coordenar*, *habilidade de cooperar*, *disponibilidade*, *integridade*, *habilidade de se dividir em módulos* ou *habilidade de agregar-se* [Kolp01a][Kolp01b][Kolp02]. Portanto, as arquiteturas organizacionais integram requisitos não-funcionais com o projeto arquitetural. Diferentemente dos requisitos funcionais que definem o que se espera que um software faça, requisitos não-funcionais especificam restrições globais em como o software opera ou como a funcionalidade é exibida. Requisitos não-funcionais são tão importantes quanto requisitos funcionais. Eles não são simplesmente propriedades de qualidades desejadas, mas aspectos críticos de sistemas dinâmicos sem os quais as aplicações não podem trabalhar e evoluir apropriadamente. A necessidade de tratar propriedades não-funcionais explicitamente é uma questão crítica quando a arquitetura de software é construída. Na próxima seção apresentaremos nossa proposta para capturar e representar os conceitos e características atualmente usadas para representar os estilos arquiteturais organizacionais utilizando os elementos da UML-RT.

5.3 Mapeando i* para UML-RT

Conforme vimos nos capítulos anteriores, em Tropos, atores são entidades ativas que executam algumas ações para atingir metas exercitando sua base de conhecimento. Na seção 4.5.3, nós mostramos que na UML Real-Time, cápsulas são classes ativas especializadas usadas para modelar componentes autocontidos de um sistema. Dessa forma podemos mapear naturalmente um ator em Tropos para uma cápsula em UML-RT. Por exemplo, na Figura 5.11, o ator *AtorA* foi mapeado para a cápsula *CapsulaA*, assim como o ator AtorB foi mapeado para a cápsula CapsulaB. Note que portas são partes físicas da implementação de uma cápsula que mediam a interação da cápsula com o mundo externo.

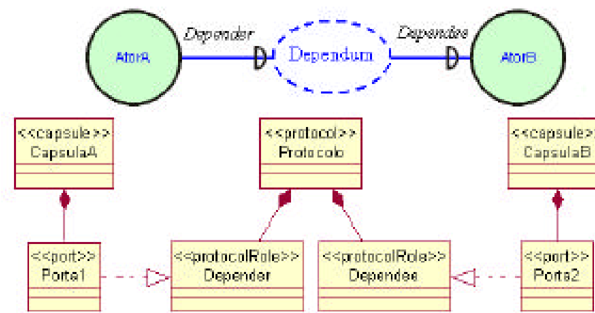


Figura 5.11 - Mapeando uma dependência entre atores para UML

Em Tropos uma dependência descreve um "acordo" (chamado dependum) entre dois atores exercendo os papéis de depender e dependee, respectivamente. O depender é o ator dependente e o dependee é o ator de quem se é dependido. Dependências tem a forma *depender? dependum*? *dependee*. Em UML-RT, um protocolo é uma especificação explícita do acordo contratual entre seus participantes, os quais exercem papéis específicos no protocolo. Em outras palavras, um protocolo captura as obrigações contratuais que existem entre cápsulas. Assim, um dependum é mapeado para um protocolo e os papéis de depender e dependee são mapeados para os papéis (estereótipo protocolRoles) que compreendem o protocolo. Observe na Figura 5.11, que o ator AtorA, que atua como depender numa dependência, agora será representado por uma cápsula que possui uma porta realizando o papel de depender no protocolo. Analogamente, o ator

AtorB, que atua como dependee numa dependência, agora será representado por uma cápsula que possui uma porta realizando o papel de dependee no protocolo.

O tipo de dependência entre dois atores (chamada dependum) descreve a natureza do acordo. Tropos define quatro tipos de dependums: metas, metas-soft, tarefas e recursos. Cada tipo de dependum definirá diferentes características no protocolo e conseqüentemente nas portas que realizam seus papéis.

Como notado anteriormente, protocolos são definidos em termos de entidades chamadas de papéis do protocolo. Já que papéis do protocolo são classes abstratas e portas exercem um papel específico em algum protocolo, um papel do protocolo define o tipo de uma porta. Isto pode significar simplesmente que a porta implementa o comportamento especificado pelo papel do protocolo. Definimos anteriormente que cápsulas são objetos arquiteturais complexos, físicos, possivelmente distribuídos que interagem com o ambiente que os cerca através de portas. Assim, note que uma porta tanto é uma parte componente da estrutura da cápsula como também uma restrição sobre seu comportamento. Conseqüentemente, cada um dos tipos de dependum restringirá o comportamento da cápsula de forma diferente.

Por exemplo, o dependum do tipo "meta" será mapeado para um atributo presente na porta que realiza o papel *dependee* do protocolo (Figura 5.12). Em particular, este atributo será do tipo booleano para sinalizar a satisfação (*true*) ou não (*false*) desta meta. Ele representa uma meta que uma cápsula é responsável por cumprir através da troca de sinais definidos no papel *dependee* do protocolo. Observe que os sinais de entrada (<<incoming>>) definidos em um papel base do protocolo, estarão presentes no papel conjugado do protocolo como sinais de saída (<<outgoing>>) e vice-versa. Por exemplo, o sinal de entrada sinal1(), presente no papel Dependee, é definido no papel Depender como um sinal de saída também chamado de sinal1().

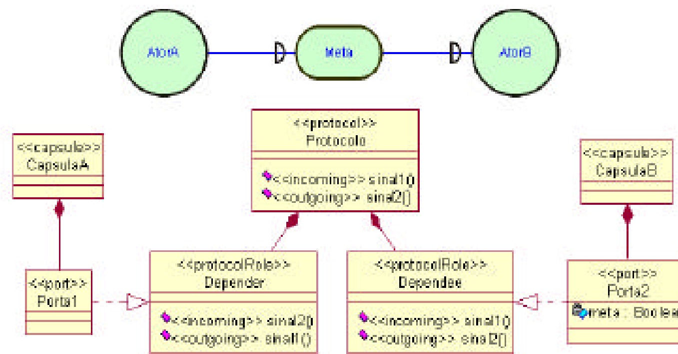


Figura 5.12 - Mapeando uma dependência de meta para UML

O dependum do tipo "meta-soft" é mapeado para um atributo presente na porta que realiza o papel *dependee* do protocolo (Figura 5.13). Em particular, este atributo será do tipo enumerado para sinalizar os níveis de satisfação atingidos para esta meta-soft. Ele representa uma meta de qualidade, que uma cápsula é responsável por cumprir em uma dada proporção através da troca de sinais definidos no papel *dependee* do protocolo.

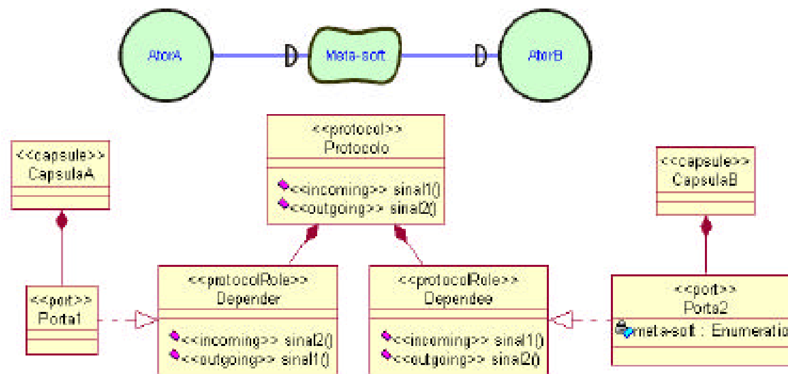


Figura 5.13 - Mapeando uma dependência de meta-soft para UML

O tipo recurso é mapeado para o tipo de retorno de um método abstrato (representado como um sinal de entrada) localizado no papel *dependee* do protocolo, que irá ser realizado por uma porta de uma cápsula (Figura 5.14). Este tipo de retorno representa o tipo do produto resultante da execução de uma operação relacionada a algum serviço de responsabilidade da cápsula. Este produto resultante da operação representará o recurso a ser fornecido pela cápsula.

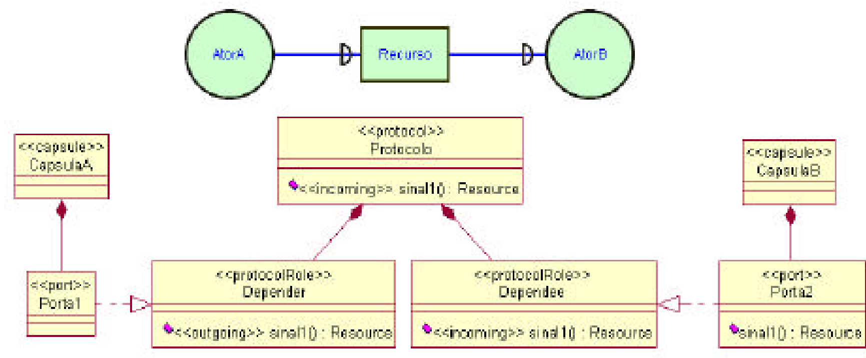


Figura 5.14 - Mapeando uma dependência de recurso para UML

O tipo tarefa é mapeado para um método abstrato localizado no papel *dependee* do protocolo, que irá ser realizado por uma porta de uma cápsula (Figura 5.15). Este método sinalizará uma operação relacionada a algum serviço de responsabilidade da cápsula e é representado como um sinal de entrada (<<incoming>>) no papel *dependee* do protocolo.

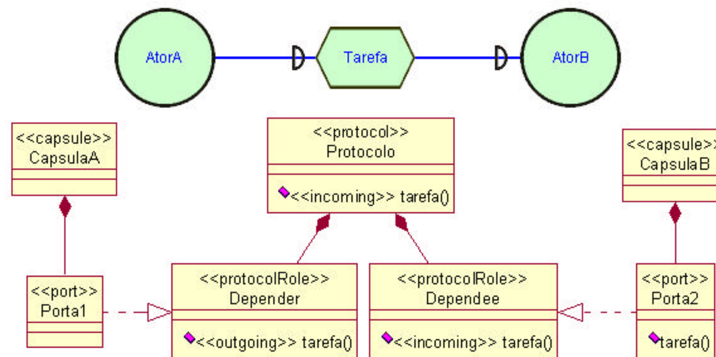


Figura 5.15 - Mapeando uma dependência de tarefa para UML

Na UML-RT, cada conector é tipado por um protocolo que especifica o comportamento desejado que pode acontecer através deste conector. Uma característica chave de conectores é que eles só podem interconectar portas que exercem papéis complementares no protocolo associado com o conector. Em um diagrama de classes, um conector é modelado por uma associação enquanto que em um diagrama de colaboração de cápsulas ele é declarado através dos papéis na associação. Assim, uma dependência (*depender? dependum? dependee*) em Tropos é mapeada para um conector em UML-

RT. Por exemplo, na Figura 5.16 da próxima seção, encontramos as dependências entre os atores ilustrados inicialmente na figura 5.1, representadas através de conectores entre as cápsulas correspondentes a estes atores.

Na próxima seção nós mostramos como os estilos arquiteturais organizacionais definidos na metodologia Tropos são modelados usando UML-RT.

5.4 Estilos Arquiteturais Organizacionais em UML

A notação UML de cápsulas, portas, protocolos e conectores agora será usada para modelar os atores arquiteturais e suas dependências. Na Figura 5.16, cada cápsula está representando um ator da arquitetura *União Estratégica*.

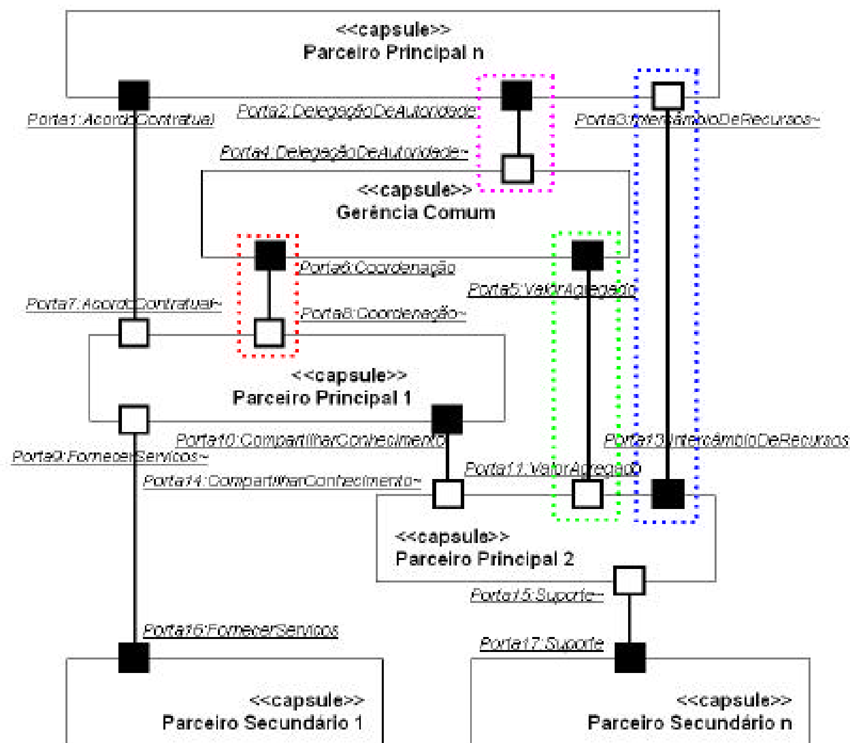


Figura 5.16 - Estilo União Estratégica em UML-RT

Quando um ator é um *dependee* de alguma dependência, sua cápsula correspondente tem uma porta de implementação (porta final) para cada dependência que exerce o papel base

do protocolo associado (ex. Port1). Esta porta é usada para prover serviços para outras cápsulas e é indicada por um quadrado preto. Quando um ator é um *depender* de algumas dependências, sua cápsula correspondente tem uma porta de implementação (porta de repasse) que exerce o papel conjugado do protocolo associado (ex. Port3). Esta porta é usada para trocar mensagens e é indicada por um quadrado branco.

Cada ator presente na Figura 5.1 é mapeado para uma cápsula na Figura 5.16. Por exemplo, o ator Gerência Comum ilustrado na Figura 5.1 é representado como a cápsula Gerência Comum na Figura 5.16. Cada *dependum*, i.e., o "acordo" entre estes dois atores é mapeado para o protocolo na Figura 5.17. Um protocolo é uma especificação explícita do acordo contratual entre os participantes no protocolo. Em nosso estudo estes participantes são os dois atores previamente mapeados para cápsulas. Cada dependência é mapeada para um conector na Figura 5.16. Cada conector é tipado pelo protocolo que representa o *dependum* da sua correspondente dependência. O tipo da dependência descreve a natureza do acordo, i.e., o tipo do conector descreve a natureza do protocolo. Os quatro tipos de *dependums* (Meta, Meta-soft, Tarefa e Recurso) são mapeados para quatro tipos de protocolos (Figuras 5.17, 5.18, 5.19, 5.20). A partir da Figura 5.1 podemos recordar a dependência de meta *Delegação de Autoridade* entre os atores *Parceiro Principal n* e *Gerência Comum*. O tipo desta dependência será representado como o protocolo *Delegação de Autoridade* (Figura 5.17), que garante que a meta associada será cumprida pelo uso dos sinais descridos no papel *dependee* deste protocolo. A meta associada a esta dependência será mapeada para um atributo booleano presente na porta que implementa o papel *dependee* do protocolo. Este atributo será verdadeiro se a meta foi cumprida e falso caso contrário. Assim, na dependência entre as cápsulas *Parceiro Principal n* e *Gerência Comum* representadas na segunda área pontilhada da Figura 5.16, a dependência de meta será mapeada para um atributo booleano localizado na porta que compõe a cápsula *Parceiro Principal n* e implementa o papel *dependee* do protocolo que garante o cumprimento desta meta (Figura 5.17).

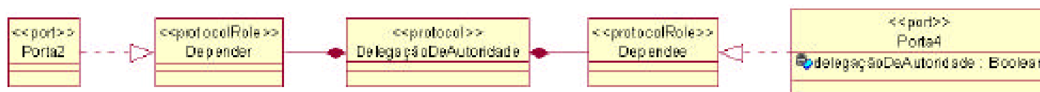


Figura 5.17 - Protocolos e portas que representam a dependência de meta Delegação de Autoridade do estilo União Estratégica

Agora examine a dependência de meta-soft *Valor Agregado* entre os atores *Parceiro Principal 2* e *Gerência Comum* representados na Figura 5.1. Neste caso, o protocolo *Valor Agregado* (Figura 5.18) garante que esta meta-soft será satisfeita em alguma proporção pelo uso dos sinais descritos no papel *dependee* do protocolo A meta-soft será mapeada para um atributo enumerado presente na porta que implementa o papel *dependee* do protocolo. Este atributo representará os diferentes graus de cumprimento da meta-soft. Assim, na dependência entre as cápsulas *Parceiro Principal 2* e *Gerência Comum* representadas na terceira área pontilhada da Figura 5.16, a dependência de meta-soft será mapeada para um atributo enumerado localizado na porta que compõe a cápsula *Gerência Comum* e implementa o papel *dependee* do protocolo que garante algum grau de cumprimento desta meta-soft (Figura 5.18).

Figura 5.18 - Protocolos e portas que representam a dependência de meta-soft Valor Agregado do estilo União Estratégica

Na seqüência, observe a dependência de tarefa *Coordenação* entre os atores *Parceiro Principal 1* e *Gerência Comum* representados na Figura 5.1. Aqui, o protocolo *Coordenação* (Figura 5.19) garante que esta tarefa será realizada pelo uso dos sinais descritos no papel *dependee* do protocolo. A tarefa em si será mapeada para um método (representado como um sinal <<*incoming*>>) no papel *dependee* do protocolo e a porta que implementa este papel estará comprometida a realizar este método. Assim, na dependência entre as cápsulas *Parceiro Principal 2* e *Gerência Comum* representadas na primeira área pontilhada da Figura 5.16, a dependência de tarefa será mapeada para um sinal com o estereotipo <<*incoming*>> localizado no papel *dependee* do protocolo que garante a execução desta tarefa. A cápsula *Gerência Comum* é composta por uma porta que implementa o papel *dependee* deste protocolo (Figura 5.19).

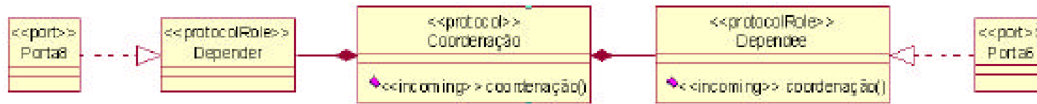


Figura 5.19 - Protocolos e portas que representam a dependência de tarefa Coordenação do estilo União Estratégica

Finalmente nós temos a dependência de recurso *Intercâmbio de Recursos* entre os atores *Parceiro Principal 2* e *Parceiro Principal n* descritos na Figura 5.16. Novamente, o protocolo *Intercâmbio de Recursos* (Figura 5.20) garante que este recurso será fornecido pelo uso dos sinais descritos como sinais <<*incoming*>> no papel *dependee* do protocolo. O recurso será mapeado para o tipo de retorno de um método (representado com um sinal <<incoming>>) presente no papel *dependee* do protocolo e a porta que implementa este papel estará comprometida a realizar este método. Assim, na dependência entre as cápsulas *Parceiro Principal 2* e *Parceiro Principal n* representadas na quarta área pontilhada da Figura 5.16, a dependência de recurso será mapeada para um sinal <<*incoming*>> que retorna o recurso e está localizada no papel *dependee* do protocolo que garante o fornecimento deste recurso. A cápsula *Parceiro Principal 2* é composta por uma porta que implementa o papel *dependee* deste *protocolo* (Figura 5.20).

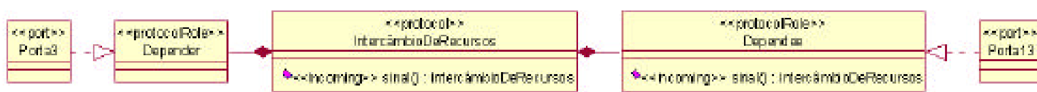


Figura 5.20 - Protocolos e portas que representam a dependência de recurso Intercâmbio de Recursos do estilo União Estratégica

Embora nós tenhamos apenas detalhado o mapeamento de quatro dependências no estilo *União Estratégica* para sua representação em UML-RT, as dependências restantes são mapeadas de forma análoga, de acordo com seus tipos como vemos na Figura 5.21.

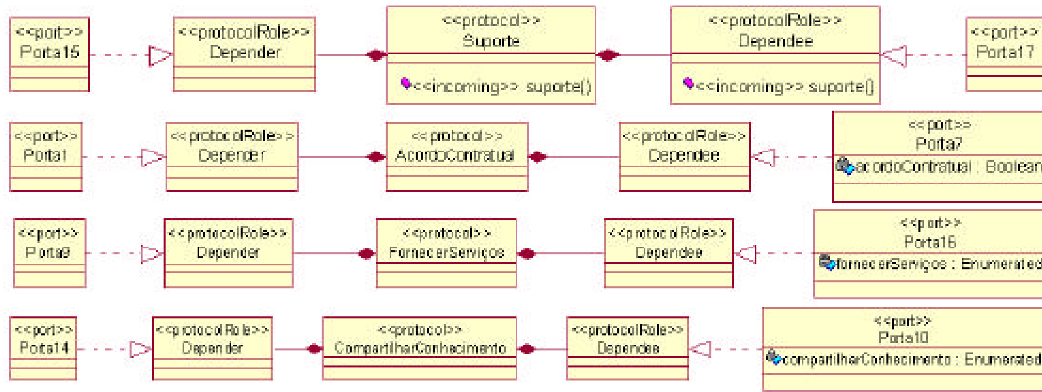


Figura 5.21 - Protocolos e portas que representam as demais dependências do estilo União Estratégica

No Apêndice A, apresentamos a representação dos demais estilos arquiteturais organizacionais definidos pelo Tropos, usando a linguagem de descrição arquitetural UML-RT.

5.5 Considerações Finais

Interessado em ajustar o espaço semântico entre a arquitetura do software e o modelo de requisitos do qual ela é derivada, *Tropos* definiu estilos arquiteturais organizacionais que focam tanto em processos de negócio quanto em requisitos não-funcionais e são direcionados a aplicações multi-agente, cooperativas, dinâmicas e distribuídas. Contudo, Tropos ainda utiliza a notação do framework i* [Yu95] para representar seus modelos arquiteturais, o que não é adequado, pois ela não representa informações detalhadas como os protocolos de comunicação entre os componentes arquiteturais do sistema. Além disso, esta notação está apenas começando a ser reconhecida como sendo adequada para criar modelos de requisitos e o seu suporte ferramental é limitado. Neste sentido, nosso objetivo foi o de buscar uma linguagem de descrição arquitetural adequada para representar os modelos arquiteturais produzidos pela técnica Tropos. No entanto, como foi discutido na seção 4.5.3, a UML, além de estar sendo estendida para dar suporte ao desenvolvimento orientado a agentes, também vem sendo usada para representar a arquitetura de sistemas simples e complexos. Nesse contexto, nossa proposta utiliza os elementos da UML-RT para capturar e representar os conceitos usados atualmente para

representar arquiteturas organizacionais em Tropos, além de ilustrar a interação entre os componentes arquiteturais através do diagrama de seqüência.

A notação UML estendida através de conceitos como cápsulas, portas, protocolos e conectores foi usada para modelar os atores arquiteturais e suas interdependências intencionais. Neste sentido, um ator em Tropos foi naturalmente mapeado para uma cápsula em UML-RT. Uma dependência (*depender? dependum*? *depende*e) em Tropos é mapeada para um conector em UML-RT. O tipo de uma dependência, chamada de um dependum, foi mapeado para um protocolo e os papéis que compõem a dependência, isto é, o depender e o dependee, mapeados para os papéis que compreendem o protocolo. Visto que Tropos define quatro tipos de dependums (metas, metas-soft, tarefas e recursos), cada um deles definirá diferentes características no protocolo e conseqüentemente nas portas que realizam seus papéis. Assim, o dependum do tipo meta será mapeado para um atributo com tipo booleano presente na porta que realiza o papel *dependee* do protocolo. O dependum do tipo meta-soft é mapeado para um atributo com o tipo enumerado presente na porta que realiza o papel *dependee* do protocolo. O tipo recurso é mapeado para o tipo de retorno de um método abstrato localizado no papel *dependee* do protocolo que irá ser realizado por uma porta de uma cápsula. E por fim, o tipo tarefa é mapeado para um método abstrato localizado no papel *dependee* do protocolo que irá ser realizado por uma porta de uma cápsula.

Usar UML-RT na fase de projeto arquitetural do Tropos permite construir modelos arquiteturais mais detalhados, além de representar os seus estilos arquiteturais organizacionais em uma notação industrial popular e ampliar o suporte ferramental da metodologia. As arquiteturas organizacionais representadas em UML podem se tornar conhecidas e utilizadas por outras metodologias de Engenharia de Software. O próximo capítulo fornece um estudo de caso que utiliza o framework Tropos para desenvolver um sistema de software multi-agentes para uma aplicação de comercio eletrônico. Assim poderá ser será avaliada a adequação da UML-RT como notação para a fase de projeto arquitetural do Tropos.

Capítulo 6 - Estudo de Caso

Este capítulo fornece um estudo de caso para validar o uso da UML Real Time como uma linguagem de descrição arquitetural apropriada no desenvolvimento de software orientado a agentes. Em particular, foi desenvolvido um sistema multi-agentes de comércio eletrônico, conforme a metodologia Tropos.

6.1 Introdução

Este capítulo apresenta um estudo de caso desenvolvido com o uso do Framework Tropos com o propósito de aplicar a nossa proposta de modelagem arquitetural no desenvolvimento de sistemas multi-agentes. O sistema escolhido é uma aplicação de comércio eletrônico entre empresas e clientes. Na seção 6.2 aplicamos nossa proposta ao estudo de caso de um comércio eletrônico para uma loja de mídia. Os modelos de dependência estratégica e de razoes estratégicas que utilizamos nas primeiras fases do desenvolvimento do sistema são extraídos de [Castro02].

6.2 O Sistema de Comércio Eletrônico *Medi@*

A Loja de Mídia é uma loja que vende e entrega diferentes tipos de itens de mídia tais como livros, jornais, revistas, CDs de áudio, fitas de vídeo e assim por diante. Os clientes da *Loja de Mídia* (locais ou remotos) podem definir seu pedido através de um catálogo periodicamente atualizado que descreve os itens de mídia disponíveis na *Loja de Mídia*. Ela é abastecida com os mais recentes lançamentos pelo *Produtor de Mídia* e com os itens já cadastrados pelo *Fornecedor de Mídia.* Para aumentar a fatia de mercado, a *Loja de Mídia* decidiu abrir uma frente de vendas na internet. Com a nova configuração, um cliente pode pedir itens à *Loja de Mídia* pessoalmente, por telefone ou através da internet. O sistema foi chamado *Medi@* e está disponível na *rede* mundial de computadores usando as facilidades de comunicação providas pela *Cia de Telecomunicações*. O sistema também usa serviços de finanças providos pela *Cia Bancária*, especializada em transações *on-line*. O propósito básico relativo ao novo sistema é o de permitir que um cliente possa examinar os itens disponíveis no catálogo do *Medi@* e fazer pedidos via internet.

Não há restrições acerca do registro ou sobre procedimentos de identificação para usuários do *Medi@*. Clientes potenciais podem localizar itens na loja *on-line* navegando no catálogo ou buscando-os no banco de dados de itens. O catálogo agrupa itens de mídia do mesmo tipo em (sub) hierarquias e gêneros (ex, CDs de áudio são classificados como pop, rock, jazz, opera, mundo, música clássica, trilha sonora,etc.). Portanto, estes clientes podem navegar só na (sub) categoria de seu interesse. Um motor de busca *on-line* permite que os clientes com itens particulares em mente possam pesquisar por título, autor/artista e campos

de descrição através da busca por palavras-chaves ou texto completo. Se o item não está disponível no catálogo, o cliente tem a opção de pedir que a *Loja de Mídia* solicite-o, desde que o cliente tenha as referências da editora/gravadora (ex. ISBN< ISSN) e se identifique (em termos de nome e número do cartão de crédito). Os detalhes sobre itens de mídia incluem título, categoria de mídia (ex. livro) e gênero (ex. ficção científica), autor/artista, descrição concisa, referências e informações internacionais de editora/gravadora, data, custo e às vezes figuras (quando houver disponível). De acordo com a proposta Tropos, nesta dissertação o sistema Medi@ será desenvolvido ao longo das fases de Requisitos Iniciais, Requisitos Finais e Projeto Arquitetural, conforme apresentaremos nas seções a seguir.

6.2.1 Requisitos Iniciais

Analisando o ambiente organizacional descrito acima conseguimos elementos suficientes para produzir um primeiro modelo i* de dependências estratégicas (Figura 6.1) para a *Loja de Mídia* [Castro02]. Os principais atores deste ambiente são *Cliente*, *Loja de Mídia*, *Fornecedor de Mídia* e *Produtor de Mídia*. *Cliente* depende de *Loja de Mídia* para completar sua meta: *Comprar Itens de Mídia*. Inversamente, *Loja de Mídia* depende dos *Clientes* para *Aumentar Fatia de Mercado* e *Satisfazer Clientes*. Já que o dependum *Satisfazer Clientes* não pode ser definido precisamente, é representado como uma meta-*soft*. O Cliente também depende da *Loja de Mídia* para *Consultar Catálogo* (dependência de tarefa). Além disso, *Loja de Mídia* depende do *Fornecedor de Mídia* para o suprimento de itens de mídia de forma contínua e para o fornecimento de *Itens de Mídia* (dependência de recurso). Espera-se que os itens sejam de boa qualidade porque, caso contrário, a dependência *Negócio Contínuo* não seria completada. Finalmente, espera-se que o *Produtor de Mídia* supra o *Fornecedor de Mídia* com *Pacotes de Qualidade*.

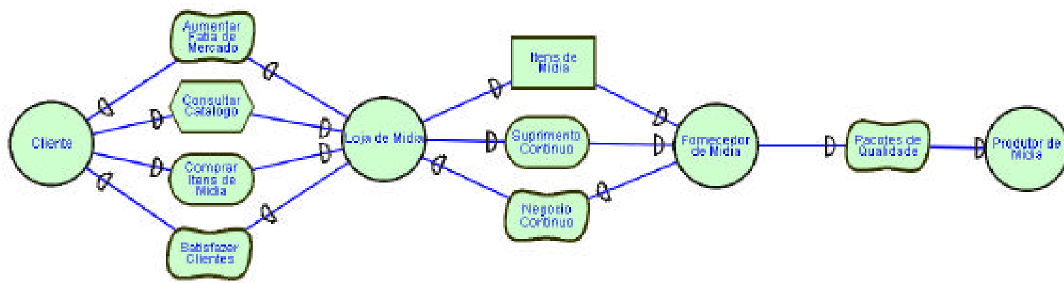


Figura 6.1 - Modelo i* de Dependência Estratégica da *Loja de Mídia*

A Figura 6.2 destaca uma das dependências de meta-*soft* identificadas pela *Loja de Mídia*, chamada *Aumentar a Fatia de Mercado*. Para atingir esta meta-*soft*, a análise determina uma meta Loja Funcionando que pode ser cumprida através de uma tarefa chamada *Operar Loja*. Tarefas são parcialmente seqüências ordenadas de passos que pretendem realizar alguma meta-*soft*. Tarefas também podem ser decompostas em metas e/ou subtarefas, cujo cumprimento coletivo completa a tarefa.

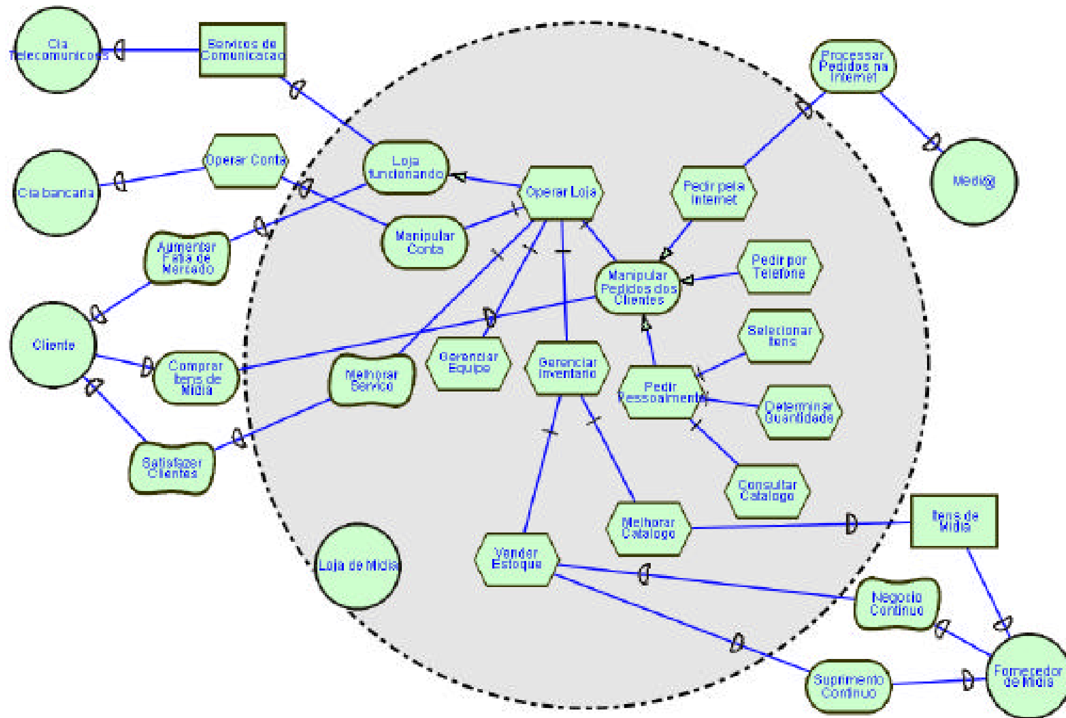


Figura 6.2 - Modelo i* de Razão Estratégica da *Loja de Mídia*

Na figura 6.2, a tarefa *Operar Loja* é decomposta nas metas *Manipular Conta* e *Manipular Pedidos dos Clientes*, nas tarefas *Gerenciar Equipe* e *Gerenciar Inventário* e na meta-*soft Melhorar Serviço* que juntos realizam a tarefa de mais alto nível. Submetas e subtarefas podem ser especificadas mais precisamente através de refinamento. Por exemplo, a meta *Manipular Pedidos dos Clientes* é cumprida através das tarefas *Pedir por Telefone* ou *Pedir Pessoalmente* ou *Pedir pela Internet*, enquanto que a tarefa *Gerenciar Pessoal* seria coletivamente realizada pelas tarefas *Vender Estoque* e *Melhorar Catalogo.* Estas decomposições eventualmente permitem-nos identificar atores que podem realizar uma meta, executar uma tarefa ou entregar algum recurso necessário para a *Loja de Mídia*. Tais dependências na Figura 6.2 são, entre outras, as dependências de meta e recurso de *Fornecedor de Mídia* para fornecer, de maneira contínua, itens de mídia para realçar o catálogo e vender produtos, as dependências de meta-*soft* de *Cliente* para *Aumentar a Fatia de Mercado* (rodando a loja virtual) e *Satisfazer os Clientes* (melhorando o serviço), e a dependência de tarefa *Operar Conta* da *Cia Bancária* para controle de transações de negócio. Observe que na Figura 6.2 introduzimos um ator para lidar com a responsabilidade de tratar dos pedidos feitos pela internet. Este ator representa o sistema a ser desenvolvido, chamado de *Medi@*, que será o foco da segunda fase do Tropos, como veremos a seguir.

6.2.2 Requisitos Finais

Para nosso exemplo, o sistema *Medi@* é introduzido como um ator no modelo de dependência estratégica descrito na Figura 6.1.

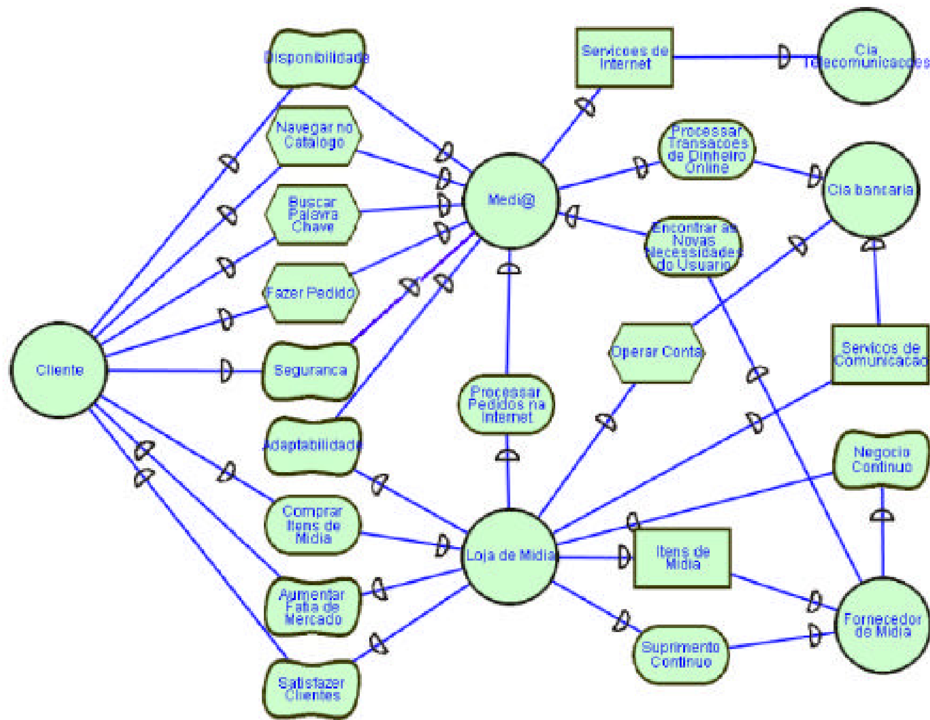


Figura 6.3 - Modelo i* de Dependência Estratégica do sistema *Medi@*

Embora um modelo de dependência estratégico forneça dicas sobre porque processos são estruturados de uma certa forma, ele não apóia suficientemente o processo de sugestão, exploração e avaliação de soluções alternativas. À medida que a análise de requisitos finais segue, é dado a *Medi@* responsabilidades adicionais, e finaliza como um dependente de várias dependências. Além disso, o sistema é decomposto em vários sub-atores que tomam algumas destas responsabilidades. Esta decomposição e atribuição de responsabilidade são realizadas usando o mesmo tipo de análise meios-fins junto com a análise de razão estratégica ilustrada na Figura 6.4. Esta figura se foca no sistema em si, ao invés de focar um *stakeholder* externo.

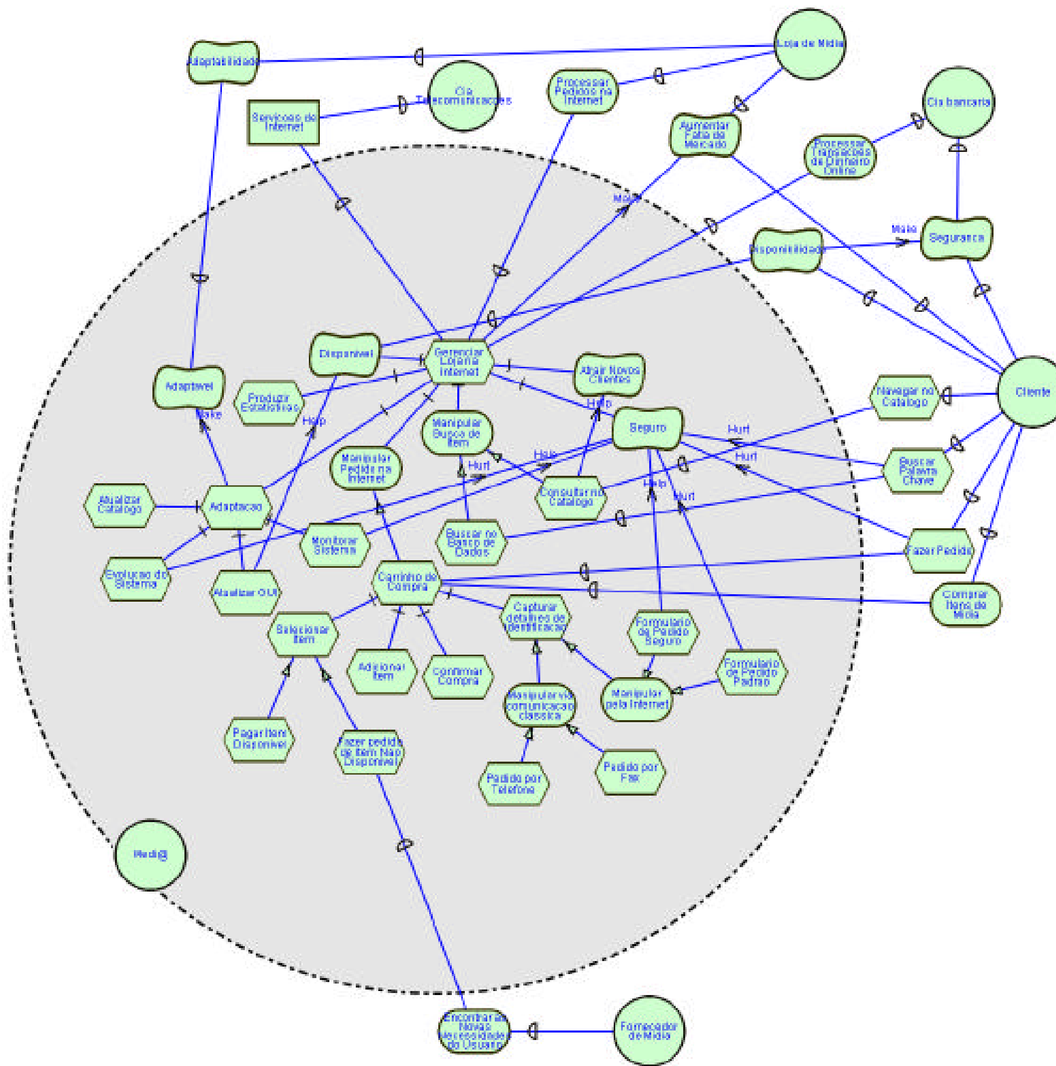


Figura 6.4 - Modelo i* de Razão Estratégica do sistema Medi@

A figura determina uma tarefa raiz *Gerenciar Loja na Internet* que provê apoio suficiente (*make*) [Chung00] a meta-*soft Aumentar Fatia de Mercado*. Esta tarefa é inicialmente refinada nas metas *Manipular Pedidos na Internet* e *Manipular Busca de Item*, metas-*soft Atrair Novos Clientes*, *Seguro e Disponível*, bem como nas tarefas *Produzir Estatísticas* e *Adaptação*. Para gerenciar pedidos de internet, *a meta Manipular Pedidos na Internet* é atingida através da tarefa *Carro de Compra* que é decomposta nas sub-tarefas *Selecionar Item*, *Adicionar Item*, *Confirmar Compra* e *Capturar Detalhes de Identificação*. Estas são

as principais atividades de processo requisitadas para projetar um carro de compra operacional on-line [Conallen00]. A última tarefa é atingida ou através da sub-meta *Manipular Comunicação clássica* lidando com pedidos por telefone e fax ou *Manipular pela Internet* gerenciando pedidos padrão. Para permitir o pedido de novos itens não listados no catálogo, *Selecionar Item* também é mais refinado em duas sub-tarefas alternativas, uma dedicada a selecionar itens de catálogo, e outra para pré-pedidos de produtos não disponíveis. Para prover apoio suficiente (*make*) para a meta-*soft Adaptável, Adaptação* é refinada em quatro sub-tarefas lidando com as atualizações de catálogo, evolução do sistema, atualizações de interface e monitoração do sistema. A meta *Manipular Busca de Item* deve ser alternativamente cumprida através das tarefas *Buscar no Banco de dados* ou *Consultar Catálogo* com respeito ao desejo de navegação de clientes, ou seja, uma busca por itens particulares em mente usando funções de busca ou simplesmente navegando no catálogo de produtos.

Em adição, como já mostrado, a Figura 6.4 introduz as contribuições de meta-*soft* para apoiar suficiente/parcial positivamente (respectivamente *make* e *help*) ou negativamente (respectivamente *break* e *hurt*) o modelo das meta-*softs Seguro, Disponível, Adaptável, Atrair Novos Clientes e Aumentar Fatia de Mercado*. O resultado desta análise meios-fins é um conjunto de atores (sistema e humano) possuem algumas das dependências que foram postuladas.

No exemplo, foram deixadas três metas-*soft* (*Disponibilidade, Segurança, Adaptabilidade*) no modelo de requisitos finais, que serão utilizadas na próxima etapa da construção do software. Portanto, após a conclusão dos requisitos finais, o desenvolvimento deverá partir para a fase de projeto da arquitetura do sistema, conforme veremos a seguir.

6.2.3 Projeto Arquitetural

A primeira atividade durante o projeto arquitetural é a de selecionar o estilo arquitetural adequado usando como critério qualidades desejadas que foram identificadas na fase anterior (fase de requisitos). Elas guiarão o processo de seleção do estilo arquitetural apropriado. Tropos utiliza o *framework* NFR [Chung00] para conduzir tal análise de qualidade.

A Tabela 6.1 resume o catálogo de correlação entre os estilos organizacionais e os atributos de qualidade [Kolp01a][Kolp01b][Kolp02]. As seguintes notações usadas pelo *framework* NFR [Chung00], *help, make, hurt, break*, respectivamente modelam contribuições parcial/positiva, suficiente/positiva, parcial/negativa e suficiente/negativa.

Catálogo de Correlação	Previsibilidade	Segurança	Adaptabilidade	Coodenabilidade	Cooperatividade	Disponibilidade	Integridade	Modularidade	Agregabilidade
Estrutura Plana	*Break*	*Break*	*Hurt*			*Help*	*Help*	*Make*	*Hurt*
Estrutura em cinco	*Help*	*Help*		*Help*	*Hurt*	*Help*	*Make*	*Make*	*Make*
Pirâmide	*Make*	*Make*	*Help*	*Make*	*Hurt*	*Help*	*Break*	*Hurt*	
União Estratégica	*Help*	*Help*	*Make*	*Help*	*Hurt*	*Make*		*Help*	*Make*
Leilão	*Break*	*Break*	*Make*	*Hurt*	*Make*	*Hurt*	*Break*	*Make*	
Tomada de controle	*Make*	*Make*	*Hurt*	*Make*	*Break*	*Help*		*Help*	*Help*
Comprimento de braço	*Hurt*	*Break*	*Help*	*Hurt*	*Make*	*Break*	*Make*	*Help*	
Contratação Hierárquica			*Help*	*Help*	*Help*	*Help*		*Help*	*Help*
Integração Vertical	*Help*	*Help*	*Hurt*	*Help*	*Hurt*	*Help*	*Break*	*Break*	*Break*
Apropriação	*Hurt*	*Hurt*	*Make*	*Make*	*Help*	*Break*	*Hurt*	*Break*	

Tabela 6.1 - Catálogo de Correlação

Lembrar que os atributos de qualidade de software *Segurança, Disponibilidade* e *Adaptabilidade* foram apresentados no modelo de requisitos finais (Seção 6.2.2). Nesta fase eles serão críticos, pois guiarão o processo de seleção do estilo arquitetural apropriado (Figura 6.5). A análise envolve refinar estas qualidades, representadas como metas-*soft*, para sub-metas que são mais precisas e mais específicas e então avaliar os estilos arquiteturais alternativos contra elas. Os estilos são representados como metas-*soft* operacionalizadas. As razões de projeto são representadas por metas-*soft* requeridas desenhadas como nuvens tracejadas. Estas podem representar informação de conteúdo (tais como prioridades) a serem consideradas e apropriadamente refletidas no processo de tomada de decisão. Marcas de exclamação (! (prioritário) e !! muito (prioritário)) são usadas para marcar prioridade de metas-*soft*. Uma marca de *check* "√" indica uma meta-*soft* cumprida, enquanto que uma cruz "X" rotula uma meta-*soft* que não pode ser cumprida.

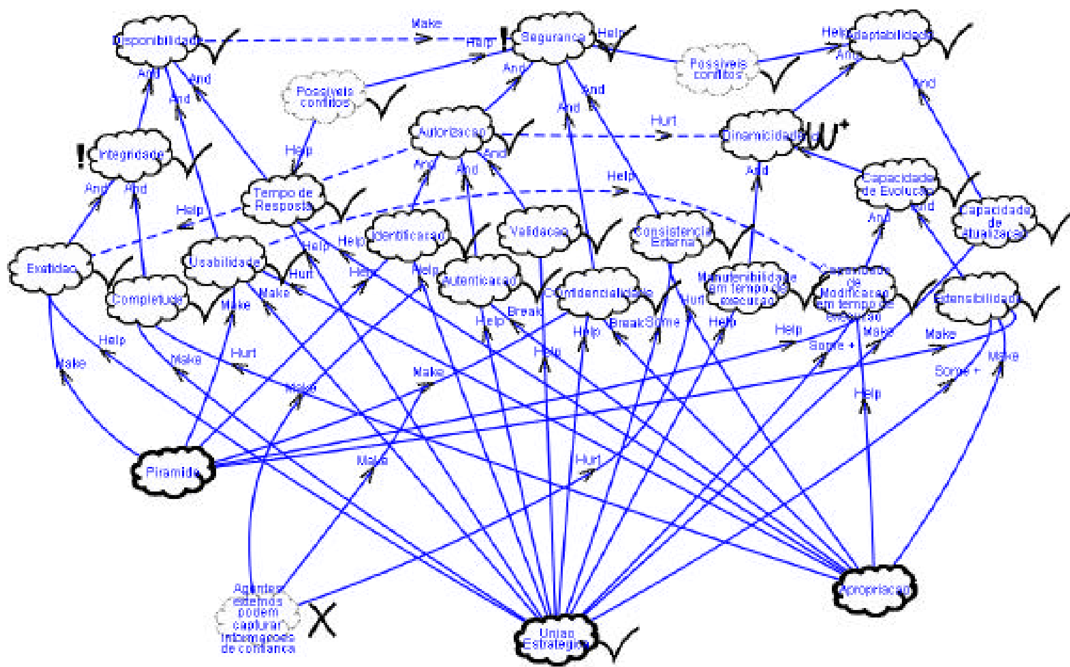


Figura 6.5 - Grafo de Interdependência de Metas-*soft*

Eventualmente, a análise mostrada na Figura 6.5 permite escolher o estilo arquitetural União Estratégica para o exemplo que está sendo apresentado (o atributo operacionalizado é marcado com um "√"). Em adição, atributos mais específicos foram identificados durante o processo de decomposição, tais como *Integridade* (exatidão, completude), *Usabilidade*, *Tempo de Resposta*, Manutenibilidade, Adaptabilidade, *Confidencialidade*, *Autorização* (*Identificação*, *Autenticação*, *Validação*), *Consistência* e precisam ser considerados na arquitetura do sistema. Mais detalhes sobre a seleção e processo de decomposição de requisitos não funcionais pode ser encontrado em [Kolp01a][Kolp01b].

A figura 6.6 sugere uma possível atribuição de responsabilidades do sistema, baseado no estilo arquitetural *União Estratégica*. O Sistema é decomposto em três parceiros principais (*Frente de Loja, Processador de Fatura e Retaguarda de Loja*) representados como cápsulas que se controlam numa dimensão local e trocam, fornecem e recebem serviços, dados e recursos. Cada uma delas delega autoridade e é controlada e coordenada por uma quarta cápsula, *Gerente Comum*, responsável por gerenciar o sistema em uma dimensão global. As responsabilidades de cada cápsula estão representadas na Figuras 6.6 como

protocolos de interação realizados pelas portas que compõem as cápsulas interativas. Os detalhes de cada um destes protocolos podem ser vistos no Apêndice B.

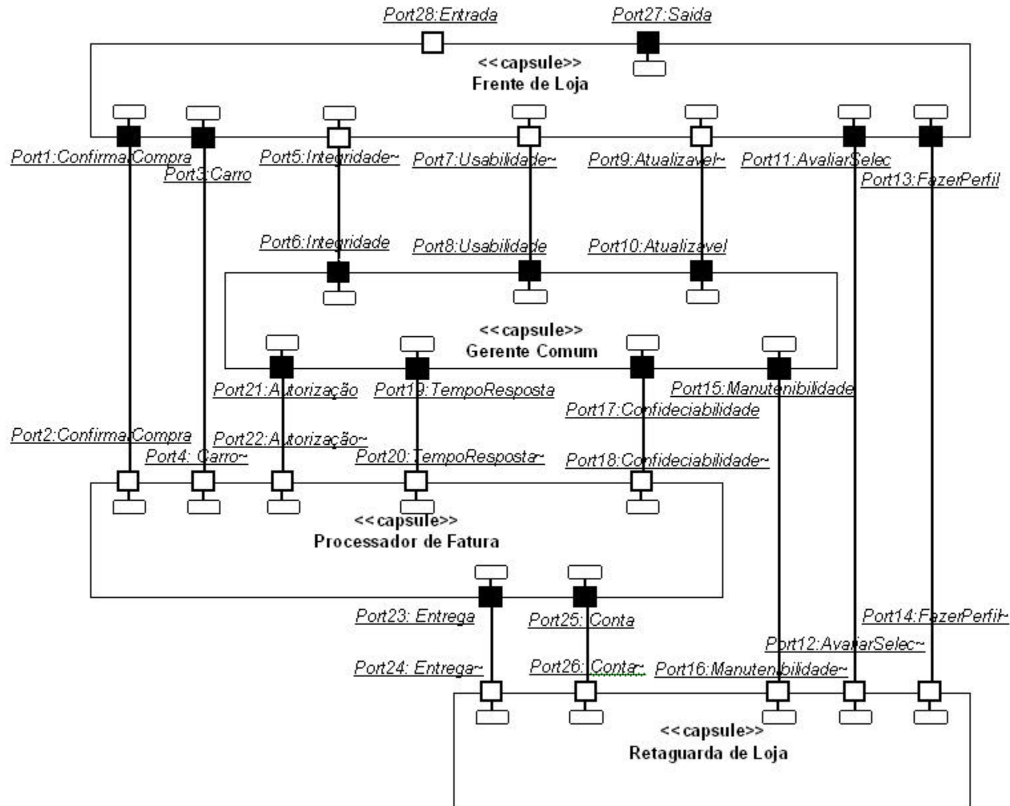


Figura 6.6 - A arquitetura *União Estratégica* do sistema *Media@*

A *Frente de Loja* interage principalmente com o *Cliente* e o supre com uma aplicação *web front-end* usável. É responsável pela navegação no catálogo e busca de itens no banco de dados, além de fornecer aos clientes on-line informações sobre itens de mídia. A *Retaguarda de Loja* mantêm o registro de todas as informações *web* sobre os clientes, produtos, vendas, contas e outros dados de importância estratégica para a *Loja de Mídia*. O *Processador de Fatura* é responsável pelo gerenciamento seguro de pedidos e contas e outros dados financeiros, como também pelas interações com a *Cia Bancária*. O *Gerente Comum* gerencia todas elas, controlando as possíveis aberturas na segurança, gargalos na disponibilidade e questões de adaptabilidade, de forma a garantir os requisitos não-funcionais do software. Todas os quatro cápsulas precisam se comunicar e colaborar entre

si no sistema em execução. Por exemplo, a *Frente de Loja* se comunica com o *Processador de Fatura* para fornecer informações relevantes do cliente para processar a fatura. A *Retaguarda de Loja* organiza, armazena e mantém todas as informações vindas da *Frente de Loja* e do *Processador de Fatura* a fim de produzir análise estatística, mapas históricos e dados de marketing.

Diante dessa decomposição, o cumprimento das obrigações do sistema pode ser realizado através de delegação de suas responsabilidade para cápsulas componentes. A introdução de outras cápsulas demanda uma forma de delegação no sentido que a cápsula *Frente de Loja* retém suas obrigações, mas delega sub-tarefas, sub-metas, etc. a outras cápsulas – um projeto arquitetural alternativo teria a cápsula *Frente de Loja* delegando alguma dessas suas responsabilidades a algumas outras cápsulas, de forma que *Frente de Loja* se remove do caminho crítico de cumprimento da obrigação. Em adição, como mostrado na Figura 6.7, a cápsula *Frente de Loja* – e as demais cápsulas do sistema (Figuras 6.8, 6.9 e 6.10) – é também refinada em uma agregação de cápsulas que, por projeto, trabalham juntas para cumprir as obrigações de *Frente de Loja*.

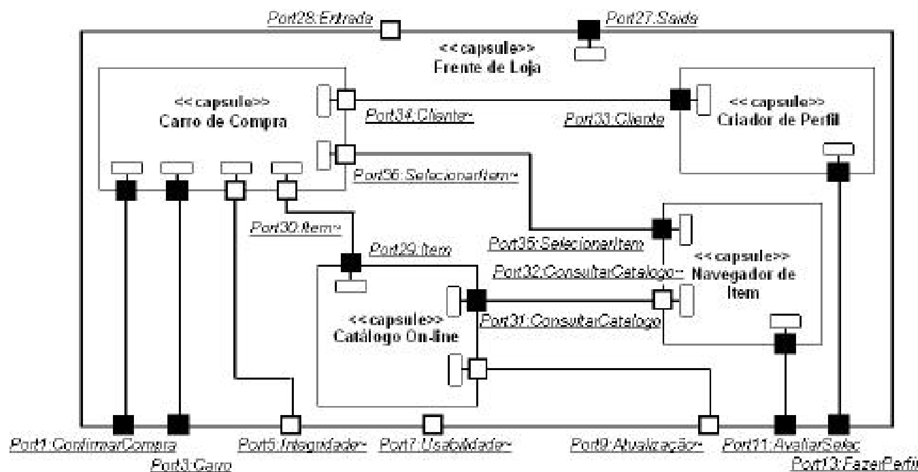


Figura 6.7 - Refinamento da cápsula Frente de Loja

Neste sentido, para acomodar as responsabilidades da *Frente de Loja*, nós introduzimos as sub-cápsulas *Navegador de Item* para gerenciar a navegação do catálogo, *Carro de Compra* para selecionar e personalizar itens, *Criador de Perfil* dos clientes para rastrear os dados do

cliente e produzir os perfis dos clientes e *Catálogo On-line* para lidar com obrigações da livraria digital (Figura 6.7).

Além disso, para lidar com os atributos de qualidade de software identificados (*Segurança. Disponibilidade e Adaptabilidade*), a cápsula *Gerente Co*mum é refinada em quatro novas sub-cápsulas (Figura 6.8): *Controle de Segurança, Gerente de Disponibilidade* e *Gerente de Adaptabilidade*, cada uma assumindo uma ou mais metas-*soft* (e sua sub-metas mais específicas) e observadas por um *Monitor*.

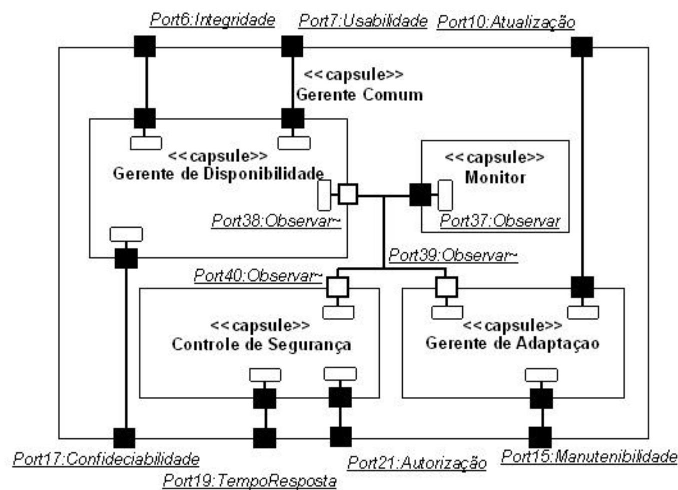


Figura 6.8 - Refinamento da Cápsula Gerente Comum

A cápsula *Gerente de Disponibilidade* (Figura 6.8) é responsável por permitir que o sistema decida automaticamente, em tempo de execução, as características dos seus componentes que se adaptem melhor às necessidades dos clientes ou às especificações de navegador/plataforma. Ela preocupa-se com a capacidade do sistema fazer o que precisa ser feito, tão rápido e eficientemente quanto possível numa situação de perda do servidor. Em particular a habilidade de o sistema responder a tempo às solicitações dos clientes relativas aos seus serviços, além de prover ao cliente uma aplicação usável, isto é, compreensível à primeira vista, intuitiva e ergonômica. Outras preocupações se referem à usabilidade e a portabilidade da aplicação através das implementações do navegador e da qualidade da interface. Por exemplo, que tipo de engenho de busca, refletindo diferentes técnicas de busca, o *Navegador de Item* deve usar (booleano, palavra-chave, thesaurus, léxico, texto

completo, lista indexada, navegação simples, navegação de hipertexto, buscas SQL, etc.), ou que tipo de *Carro de Compra* é mais apropriado às configurações de *plug-ins*, plataforma ou navegador do cliente (carro de compra ao modo Java, carro de compra de navegador simples, carro de compra baseado em frame, carro de compra CGI, carro de compra CGI melhorado, carro de compra baseado em *shockwave*, etc.). A cápsula *Controle de Segurança* leva em consideração as configurações de ambiente, especificações do navegador e os protocolos de redes utilizados. Por exemplo, os navegadores da *web* e servidores de aplicação podem descarregar e submeter conteúdo e programas que poderiam tornar o sistema do cliente vulneráveis para crackers e agentes automatizados mal intencionados espalhados por toda a rede. Já a cápsula *Gerente de Adaptabilidade* lida com a forma com que o sistema pode ser projetado usando mecanismos genéricos permitindo que páginas *web,* interfaces do usuário, esquema de banco de dados e o modelo arquitetural sejam facilmente mudados ou ampliados de forma dinâmica para integrar novas e futuras tecnologias relacionadas a w*eb*. Dessa forma ela permite que o conteúdo da informação e o layout possam ser freqüentemente atualizados para dar a informação correta para os clientes ou simplesmente serem modelados por razões de mercado.

A cápsula *Processador de Fatura* (Figura 6.9) é decomposta em *Processador de Pedido* para conversar com *Carro de Compra* e lidar com detalhes de conta, *Processador de Conta* para interagir com *Companhia Bancária* (não representada na Figura 6.6) e *Processador de Recibo* para lidar com entregas e informações de recibo.

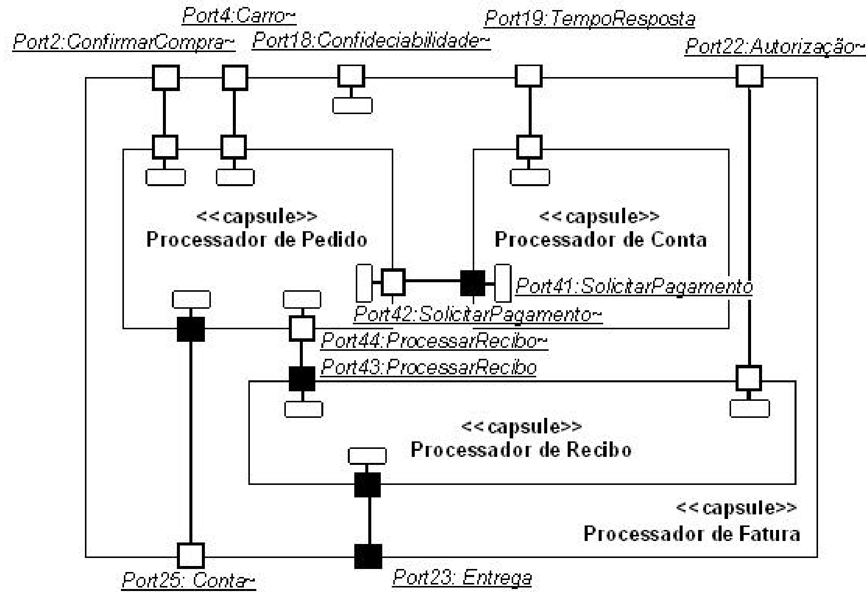


Figura 6.9 - Refinamento da Cápsula Processador de Fatura

Finalmente, a cápsula *Retaguarda de Loja* (Figura 6.10) é refinada em *Processador de Estatísticas* que produz mapas, relatórios, exames, promoções, previsão de retorno e *Processador de Entrega* para interagir com sistemas de informação de companhias de entrega.

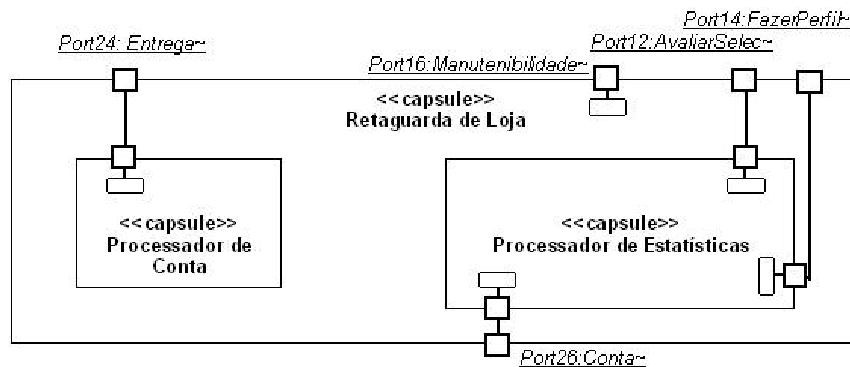


Figura 6.10 - Refinamento da Cápsula Retaguarda de Loja

A Figura 6.11 mostra a arquitetura completa do sistema Medi@, incluindo o refinamento das cápsulas de alto nível que compõem o sistema e os protocolos que regem as interações entre elas.

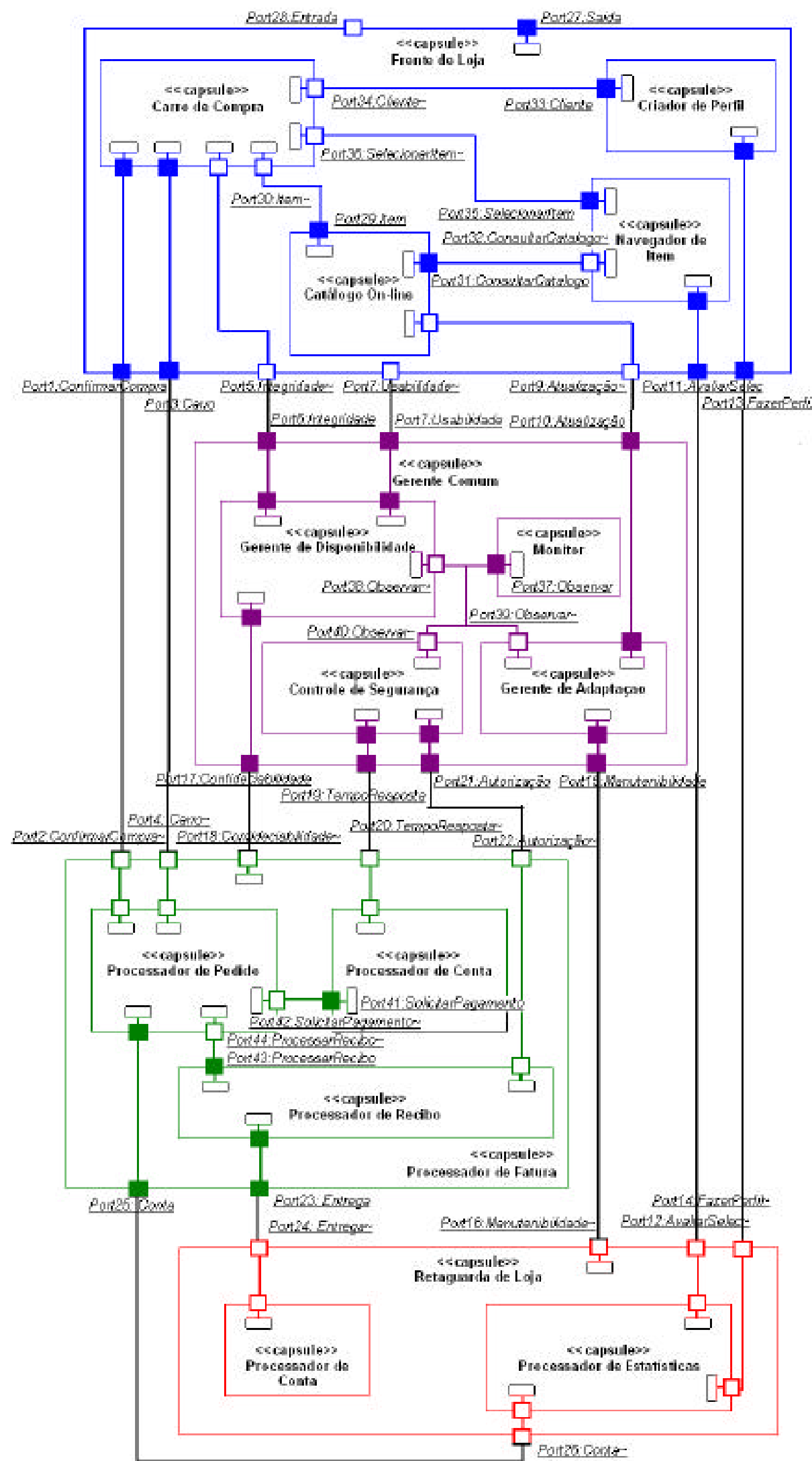


Figura 6.11 - O refinamento da arquitetura do sistema Medi@

Através de diagramas de seqüência, nós ilustraremos a interação existente entre as cápsulas que compõem o sistema. Em particular concentramos na realização de dois cenários: a solicitação para *Fazer um Pedido* e a solicitação para *Pesquisar um Item* no sistema *Medi@*. A troca de mensagens entre as cápsulas acontece no contexto definido pelos protocolos implementados pelas portas que compõem cada cápsula envolvida na interação. Tais protocolos podem ser encontrados em detalhe no Apêndice B.

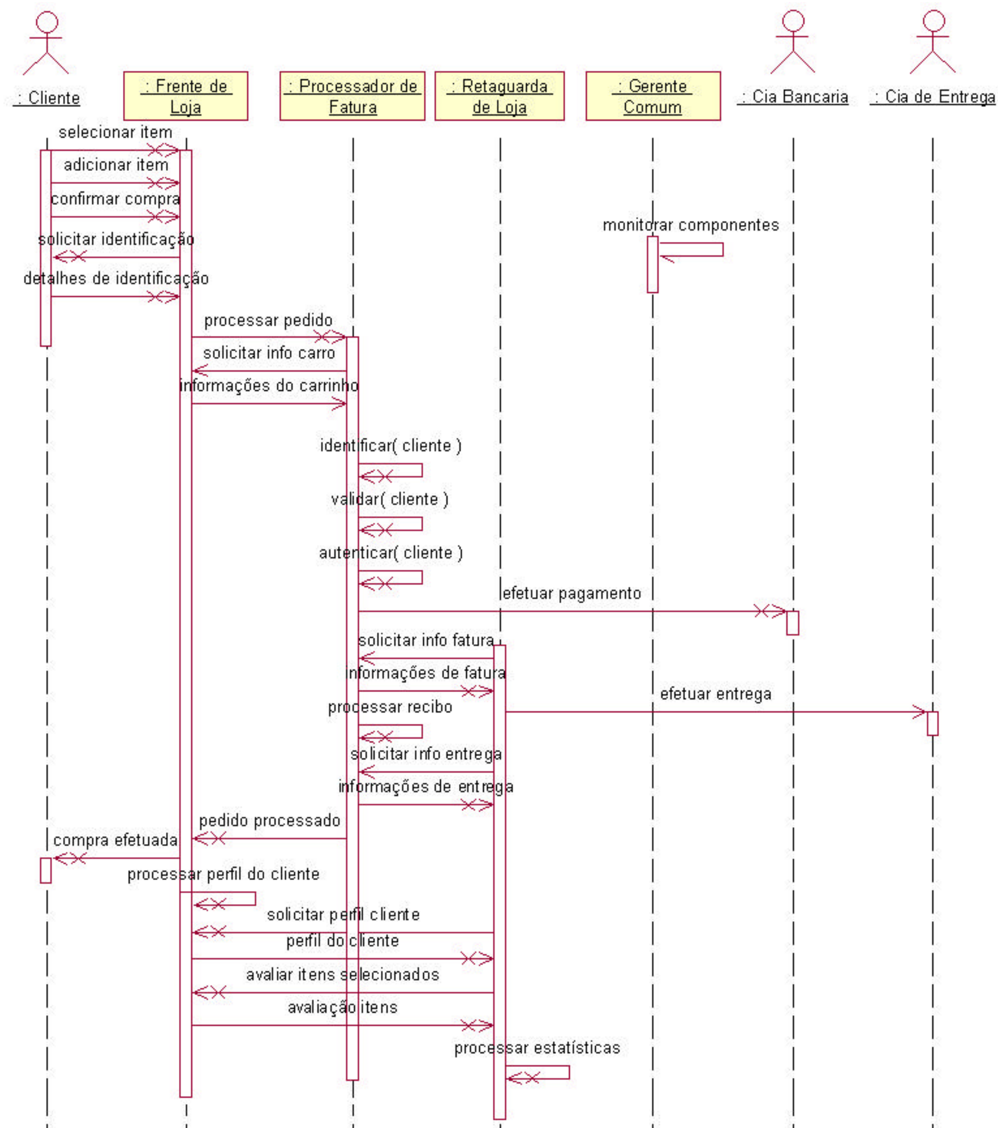


Figura 6.12 - Diagrama de seqüência para o contexto Fazer Pedido

Por exemplo, a Figura 6.12 introduz o contexto de interação para fazer um pedido de compra, que é disparado pela ação de comunicação (AC) *confirmar compra* dos itens selecionados e adicionados ao carrinho de compras e termina com uma informação de falha ou sucesso (*compra efetuada*) retornada. Quando o *Cliente* pressiona o botão confirmar compra, a *Frente de Loja* pede que ele submeta seus dados de identificação. Após receber os *detalhes de identificação* do cliente, a *Frente de Loja* pede ao *Processador de Fatura* que processe o pedido. Este, por sua vez, autentica o cliente, processa a fatura, efetua o pagamento desta e envia uma ação de comunicação de *informações de entrega* e uma ação de comunicação de *informações de fatura* para a *Retaguarda de Loja* processar as estatísticas e efetuar a entrega do pedido.

A Figura 6.13 foca no contexto em que o cliente solicita a compra de um item não disponível. Neste caso, o pedido é processado normalmente, mas a *Retaguarda de Loja* atualiza uma listagem de itens não-disponíveis para que o *Fornecedor de Mídia* possa tomar conhecimento, através do sistema *Medi@,* dos itens que precisam ser repostos no estoque da *Loja de Mídia*.

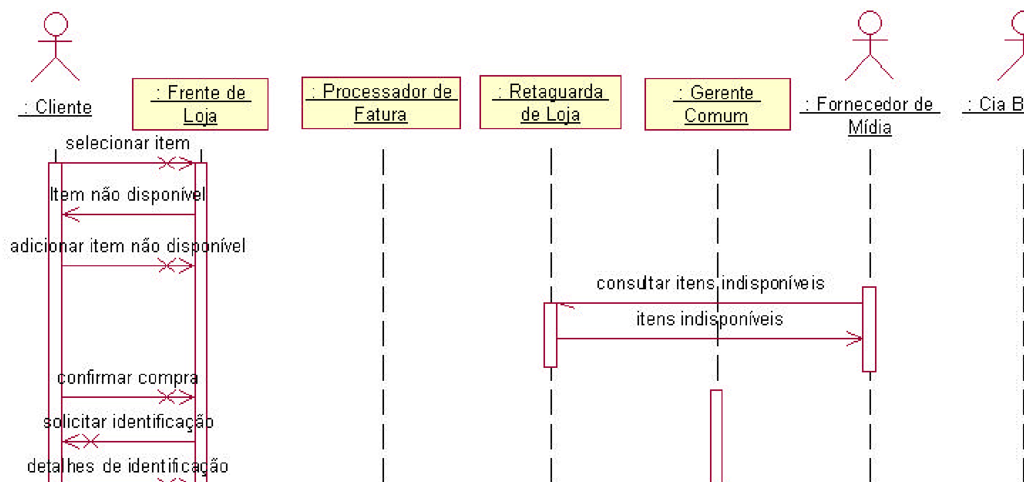


Figura 6.13 - Diagrama de seqüência para o contexto Fazer Pedido de Item Não Disponível

O contexto de interação para pesquisar um item é mostrado da Figura 6.14, o *Cliente* possui duas alternativas: Buscar item por palavra-chave (Figura 6.14a) ou Navegar no catálogo da loja de mídia (Figura 6.14b). Na Figura 6.14a encontramos o *Cliente* fazendo uma busca de

itens por palavra-chave e a *Frente de Loja* retornando o resultado desta busca (falha ou sucesso). A Figura 6.14b ilustra o *Cliente* navegando no catálogo de itens e solicitando a *Frente de Loja* os detalhes de algum item específico.

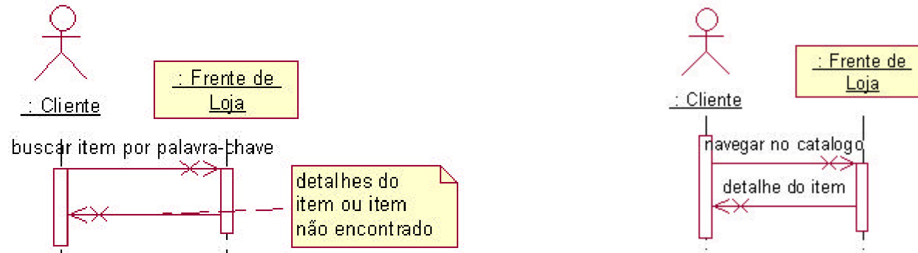


Figura 6.14 - Diagrama de seqüência para o contexto Pesquisar Item. (a) Buscar item por palavra-chave. (b) Navegar no catálogo da *Loja de Mídia*

Embora tenhamos descrito a interação dos componentes arquiteturais de mais alto nível, nas próximas figuras descreveremos a troca de mensagem entre os sub-componentes que fazem parte do refinamento da arquitetura, conforme descrito na figura 6.11. O foco será nas responsabilidades que lhes foram delegadas.

Observamos na Figura 6.15 que a cápsula *Frente de Loja* foi decomposta em quatro sub-cápsulas: *Navegador de Item*, *Carro de Compra*, *Catálogo On-line* e *Criador de Perfil*. À estas cápsulas, foram delegadas as responsabilidades da *Frente de Loja* e elas dependem umas das outras para realizar tais responsabilidades. O *Navegador* de Item é responsável pela navegação no catálogo, seleção de itens e por efetuar a avaliação dos itens previamente selecionados. Já o *Carro de Compra* lida com a adição dos itens selecionados e com o envio uma ação de comunicação de processamento de pedido ao *Processador de Fatura*, ativado pela ação de comunicação de confirmação da compra enviada pelo *Cliente*. O *Catálogo On-line* fornece ao *Carro de Compra* os detalhes dos itens selecionados e o *Criador de Perfil* trata dos detalhes de identificação do *Cliente*, fornecendo estas informações ao *Carro de Compra* e processando o perfil do *Cliente*.

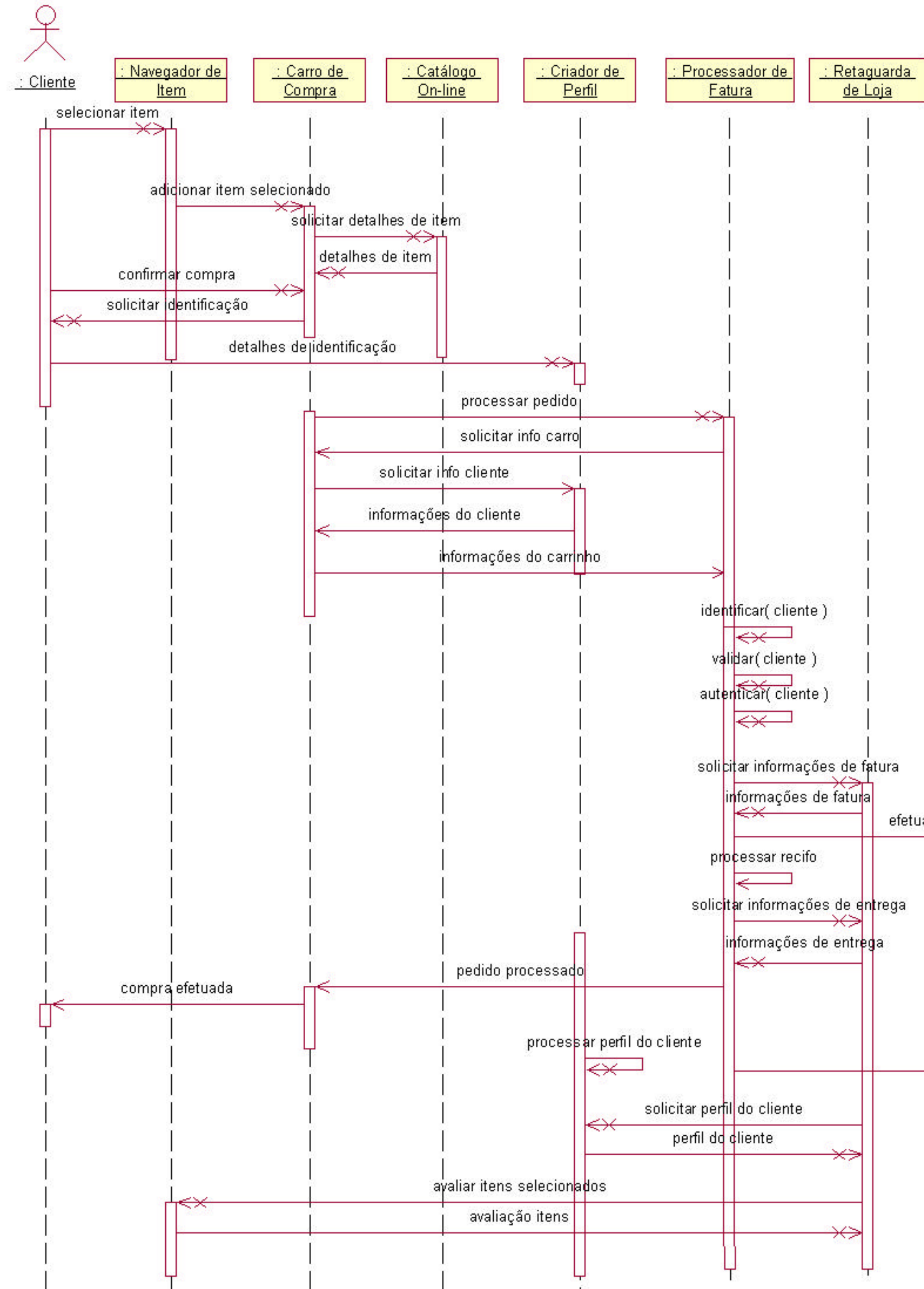


Figura 6.15 - Diagrama de seqüência refinando a *Frente de Loja* para o contexto Fazer Pedido

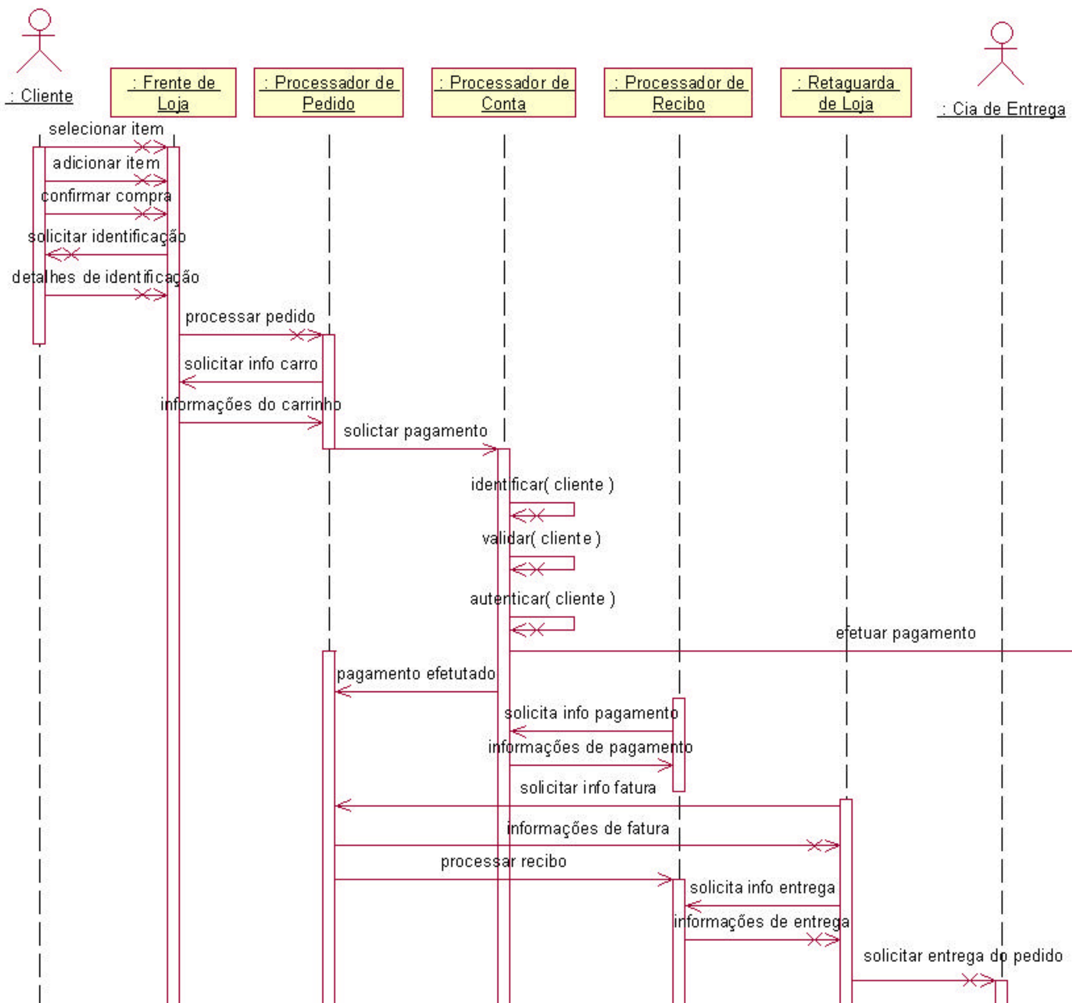


Figura 6.16 - Diagrama de seqüência refinando o *Processador de Fatura* para o contexto Fazer Pedido

Na Figura 6.16 encontramos o comportamento da decomposição da cápsula *Processador de Fatura* nas sub-cápsulas *Processador de Pedido*, *Processador de Conta* e *Processador de Recibo*, cada uma lidando com diferentes responsabilidades do *Processador de Fatura*. Em particular, o *Processador de Pedido* é disparado pelo envio da CA processar pedido a partir da *Frente de Loja* para, em seguida, solicitar ao *Processador de Conta* a efetivação do pagamento do pedido. O *Processador de Conta* faz a verificação das informações do cliente antes de efetuar o pagamento do pedido e, em seguida, trata das informações de fatura.

Estas informações posteriormente serão fornecidas a *Retaguarda de Loja* para o processamento das estatísticas. O *Processador de Recibo*, por sua vez, processará o recibo e enviará uma CA de informações de entrega para a *Retaguarda de Loja* solicitar a entrega do pedido.

Um passo adicional no projeto arquitetural consiste no refinamento das cápsulas de mais baixo nível em agentes de software conforme padrões sociais e definir como as responsabilidades atribuídas a cada uma destas cápsulas serão cumpridas pelos agentes. Entretanto esta etapa do projeto arquitetural está fora do escopo desta dissertação.

6.3 Considerações Finais

Apresentamos neste capítulo a aplicação prática da nossa proposta de detalhamento da arquitetura organizacional através de sua representação em UML *Real Time* (UML-RT). Um estudo de caso referente a um sistema de comércio eletrônico foi tratado neste capítulo. Como resultado, conseguimos descrever os modelos de projeto arquitetural com mais precisão e riqueza de detalhes, a representação destes modelos numa notação popular e bem-estabelecida, como também suportado por ferramentas CASE.

O estudo de caso auxiliou na observação de que o uso de arquiteturas organizacionais pode levar a modelos arquiteturais mais próximos da configuração ambiental onde o sistema multi-agente eventualmente irá operar, diminuindo a abismo semântico entre a fase de requisitos e a fase arquitetural do desenvolvimento de software orientado a agentes. Além disso, estas arquiteturas permitem construir aplicações multi-agentes, cooperativas, dinâmicas e distribuídas, que demandam por arquiteturas abertas e flexíveis e precisam considerar atributos de qualidade no seu projeto.

Verificamos também, que um conjunto de informações detalhadas no modelo de razão estratégica i* (SR) podem ser vistas como responsabilidades que serão atribuídas aos componentes do projeto arquitetural do sistema. No entanto, o sucesso da transição da fase de requisitos para a fase arquitetural depende da experiência do engenheiro de software com relação às duas técnicas empregadas, i* e UML, já que no momento há poucas diretrizes sistemáticas que suportem a construção de modelos arquiteturais em UML a

partir de modelos de requisitos em i* (Vide [Bastos03]). Além disso, é necessário que o engenheiro domine o mapeamento entre os conceitos destas duas técnicas. Contudo, desde que heurísticas adequadas tenham sido encontradas para guiar tal processo, acreditamos que seja possível automatizá-lo através de uma ferramenta computacional apropriada.

O uso de cápsulas definidas pela UML-RT nos permitiu refinar a arquitetura do sistema em componentes de mais baixo nível (sub-cápsulas) que dependem uns dos outros para realizar as responsabilidades do sistema como um todo. Os diagramas de seqüência ilustram as interações entre as cápsulas envolvidas na realização de um determinado cenário de uso do sistema. Em particular, estes diagramas incorporam detalhe no comportamento arquitetural, já que mostram os sinais trocados nas interações e a seqüência desses sinais (protocolo de comunicação entre cápsulas). Contudo, o diagrama de seqüência só fornece a especificação básica para um protocolo intra-cápsulas. Em particular, o diagrama não considera o procedimento interno de cada cápsula ao receber ou para produzir uma ação de comunicação. Uma alternativa seria usar o diagrama de estados para detalhar a máquina de estado de cada cápsula ilustrando o seu comportamento ao receber ou produzir uma ação de comunicação. No próximo capítulo apresentamos nossas conclusões e apontamos para trabalhos futuros.

Capítulo 7 - Conclusões e Trabalhos Futuros

Este capítulo apresenta os resultados alcançados e as lições aprendidas com esta dissertação, como também aponta para os trabalhos futuros.

7.1 Contribuições do Trabalho

Apresentamos nesta dissertação um conjunto de extensões baseadas na UML-RT para representar os estilos arquiteturais organizacionais definidos pelo Tropos. Mostramos que modelar arquiteturas organizacionais em UML-RT possibilita incorporar detalhe ao projeto arquitetural de um sistema no que diz respeito tanto ao seu comportamento quanto à sua estrutura. Podemos ilustrar graficamente não apenas os sinais trocados entre os componentes de software a fim de realizarem as suas responsabilidades, mas também a ordenação na qual essa troca de sinais ocorre. Os protocolos de interação e os diagramas de seqüência auxiliam neste sentido. O refinamento dos componentes arquiteturais de nível mais alto em componentes de nível mais baixo a fim de delegar suas responsabilidades pode ser também facilmente alcançado. Dessa forma, se torna possível analisar a atribuição de responsabilidades do sistema aos componentes arquiteturais, bem como a dependência entre tais componentes no que diz respeito à realização de suas responsabilidades. Aplicamos nossa proposta ao desenvolvimento de um sistema de software orientado a agentes para uma aplicação de comércio eletrônico.

Através do estudo de caso observamos que a instanciação dos meta-modelos arquiteturais definidos pelo Tropos ocorre de um ponto de vista organizacional. Neste caso, cápsulas dependem de outras cápsulas para atingir metas, realizar tarefas, fornecer recursos ou satisfazer metas-soft através da troca de sinais entres elas. A caracterização de cada protocolo de interação irá depender da natureza da dependência que existe entre as cápsulas.

Outras vantagens alcançadas ao se utilizar nossa proposta incluem:

- Obtenção de um modelo arquitetural mais próximo do modelo do ambiente organizacional onde o sistema vai operar, diminuindo o abismo semântico existente entre o sistema de software e seu ambiente de implantação.

- Modelagem de arquiteturas mais detalhadas tanto no aspecto estrutural quanto no comportamental.

- Construção de uma arquitetura flexível, com componentes fracamente acoplados, que podem evoluir e mudar continuamente para acomodar novos requisitos e suportam requisitos não-funcionais.
- Projeto de arquiteturas com o auxílio da ferramenta CASE que atualmente suporta a UML-RT.
- Popularização das arquiteturas organizacionais, inclusive permitindo o seu uso em outras metodologias orientadas a agentes voltadas para sistemas abertos, cooperativos, dinâmicos e distribuídos.

No entanto, baseando-se no estudo de caso realizado neste trabalho observamos algumas deficiências relevantes na proposta. Existem poucas heurísticas que auxiliem os engenheiros de software na construção de modelos arquiteturais a partir do modelo de requisitos representado em i*. Contudo, alguns trabalhos neste sentido estão sendo produzidos como podemos ver em [Bastos03]. Além disso, a utilização da nossa proposta demanda de um conhecimento completo dos conceitos suportados tanto pelo i* quanto pela UML-RT. Dessa forma, a qualidade e a consistência do projeto arquitetural possui uma forte dependência da experiência do engenheiro de software, bem como do nível de detalhamento associado às informações descritas nos modelos de requisitos em i*, para construir modelos arquiteturais completos e precisos. Contudo, é possível desenvolver uma ferramenta para suportar este processo.

Outro ponto levantado pelo estudo de caso diz respeito à complexidade dos modelos produzidos no projeto arquitetural devido à riqueza de detalhes incorporada por nossa proposta. Estes modelos arquiteturais podem se tornar bastante complicados, dependendo do domínio da aplicação sendo tratada. Para gerenciar esta complexidade, mecanismos de estruturação precisam ser aplicados de forma a tornar os modelos arquiteturais mais simples e fáceis de entender.

Embora haja ferramentas CASE comerciais específicas para a modelagem usando UML-RT, elas são voltadas apenas para as fases de projeto e implementação de sistemas em tempo-real e não consideram o uso da UML-RT como uma linguagem de descrição

arquitetural. No entanto, é possível utilizá-las com a UML-RT atuando como uma linguagem de descrição arquitetural.

7.2 Trabalhos Futuros

Como trabalhos futuros temos como prioridade propor diretrizes sistemáticas de um ponto de vista organizacional que permitam instanciar os meta-modelos arquiteturais a partir de modelos de requisitos representados em i*, já que hoje este processo é feito de uma forma *ad hoc* e baseada na experiência do engenheiro de software.

Também pretendemos realizar mais estudos de caso com projetos reais utilizando o *framework* Tropos. Dessa forma, poderemos encontrar mecanismos de estruturação adequados que permitam gerenciar a complexidade dos modelos arquiteturais produzidos por nossa proposta quando aplicada a sistemas de grande porte.

Para completar nossa proposta precisamos modelar o comportamento interno de cada cápsula arquitetural através, por exemplo, do diagrama de estado da UML. Além disso, visamos propor extensões da UML para representar os padrões sociais envolvendo agentes, bem como os aspectos e as características estruturais e comportamentais que definem tais agentes de software no contexto da abordagem Tropos.

Dessa forma, esperamos oferecer uma metodologia confiável e completa, onde modelos, técnicas e ferramentas CASE suportarão as várias fases do ciclo de desenvolvimento de software orientado a agentes. Assim, os projetistas de software orientado a agentes poderão realizar plenamente os benefícios prometidos por esta nova tecnologia.

Capítulo 8 - Referências Bibliográficas

[Allen94] Allen, R., Garlan, D.: **Formalizing architectural connection**. In proceeding of the 16th International Conference on Software Engineering (71-80), Sorrento, Italy, May (1994)

[Allen97] Allen, D., Garlan, D.: **A formal basis for architecture connection**. ACM Transaction on Software Engineering and Methodology, July (1997)

[Agentlink03] The Agentcities-Agentlink: **Agents for Commercial Applications.** Agent Technology Conference. Barcelona, Spain (2003)

[Bachmann00] Bachmann, F.: **Software Architecture Documentation in Pratice: Documenting Architectural Layers**. Special Report. CMU/SEI-2000-SR-004. March (2000)

[Bass98] Bass, L., Clements, P., Kazman, R.: **Software Architecture in Practice**. Reading, MA: Addison-Wesley Longman (1998)

[Bastos03] Bastos, L. R. D., Castro, J. F. B.: **A Proposal for Integrating Organizational Requirements and Socio-Intentional Architectural Styles in Tropos**. STRAW03 - Second International Workshop From SofTware Requirements to Architectures, 2003, Portland, Oregon, USA.

[Bauer01a] Bauer, B.: **UML Class Diagrams and Agent-Based Systems**. AGENTS'01, Montréal, Quebec, Canadá, May-June (2001)

[Bauer01b] Bauer, B., Müller, J. P., Odell, J.: **Agent UML:A Formalism for Specifying Multiagent Interaction** Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds. (91-103), Springer, Berlin (2001)

[Bergenti00] Bergenti, F., Poggi, A.: **Exploiting UML in the Design of Multi-Agent Systems.** In Engineering Societies in the Agent World, First International Workshop - ESAW2000 (106-113), Berlin, Germany, August (2000)

[Bueno90] Bueno, F. S.: Didicionário da Língua Portuguesa. 3ª. Edição Atualizada. Editora Lisa Ltda (1990)

[Busetta01] Busetta, P., Ronnquist, R., Hodgson, A, Lucas, A.: **Jack intelligent agents—components for intelligent agents in java**. User Manual 1, Agent Oriented Software, Melbourne, Australia, March (2001)

[Caire01] Caire, G., Leal, F., Chainho, P., Evans, R., Jorge, F.G., Juan Pavon, G., Kearney, P., Stark, J., Massonet, P.: **Agent Oriented Analysis using MESSAGE/UML**. In Proceedings of the 2nd International Workshop on Agent-oriented Software Engineering (AOSE 2001), Montreal, May (2001)

[Castro01] Castro, J., Kolp, M., Mylopoulos, J.: **A requirements-driven development methodology**. In Proc. of the 13th Int. Conf. on Advanced

Information Systems Engineering, CAiSE'01 (108-123), Interlaken, Switzerland, June (2001)

[Castro02] Castro, J., Kolp, M., Mylopoulos, J.: **Towards Requirements-Driven Information Systems Engineering: The Tropos Project**. In Information Systems Journal, Vol. 27:365-389, Elsevier, Amsterdam, The Netherlands (2002)

[Castro03] Castro, J., Silva, C. T. L. L., Mylopoulos, J.: **Modeling Organizational Architectural Styles in UML.** 15th Int. Conf. on Advanced Information Systems Engineering - CAiSE'03. Velden, Austria (2003) (To appear)

[Chung00] Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: **Non-Functional Requirements in Software Engineering**. Kluwer Publishing (2000)

[Clements95] Clements, P. C., Bass, L., Kazman, R., Abowd, G.: **Predicting software quality by architectural-level evaluation**. In proceeding of the Fifth International Conference on Software Quality, Austin, Texas, October (1995)

[Clements96] Clements, P. C., Northrop, L. M.: **Software Architecture: An Executive Overview.** Technical report CMU/SEI-96-TR-003 (1996)

[Conallen00] Conallen, J.: **Building Web Applications with UML**. Addison-Wesley (2000)

[DeLoach01] DeLoach, S.: **Analysis and Design using MaSE and agentTool**. Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference - MAICS (2001)

[Flake01] Flake, S., Geiger, C., Küster, J. M.: **Towards UML-based Analysis and Design of Multi-Agent Systems**. International NAISO Symposium on Information Science Innovations in Engineering of Natural and Artificial Intelligent Systems (ENAIS'2001), Dubai, March (2001)

[Ferber98] Ferber, J., Gutknecht, O.: **A meta-model for the analysis and design of organizations in multi-agent systems**. In Proceedings of Third International Conference on Multi-Agent Systems (128-135). ICMAS'98, IEEE Computer Society (1998)

[Garlan94] Garlan D., Allen, R., Ockerbloom, J.: **Exploiting style in architectural design environments**. In proceeding of SIGSOFT'94: The second ACM SIGSOFT Symposium on the Foundation of Software Engineering (170-185). ACM Press, December (1994)

[Garlan95] Garlan, D., Perry, D.: **Introduction to the Special Issue on Software Architecture**. IEEE Transactions on Software Engineering 21, April (1995).

[Garlan97] Garlan D., Monroe R.T., Wile D.: **Acme: An architecture description interchange language**. In proceeding of CASCON'97 (169-183), Ontário, Canada, November (1997)

[Garlan99] Garlan D., Kompanek A.J., Pinto, P.: **Reconciling the needs of architectural description with object-modeling notations**. Technical report, Carnegie Mellon University, December (1999)

[Garlan00a] Garlan, D., Sousa, J.P.: **Documenting Software Architectures: Recommendations for Industrial Practice**. CMU-CS-00-169. October (2000)

[Garlan00b] Garlan, D.: **Software Architecture: a Roadmap**. In proceedings of

the 22nd International Conference on Software Engineering, ICSE'00. IEEE Computer Society Press. Limerick, Ireland. June (2000)

[Gomes96] Gomes-Casseres, B.: **The alliance revolution: the new shape of business rivalry.** Harvard University Press (1996)

[Hilliard99] Hilliard, R.: **Building Blocks for Extensibility in the UML: Response to UML 2.0 Request for Information**. Integrated Systems and Internet Solutions Inc, Concord, Massachusetts. December (1999)

[Hilliard00] Hilliard, R.: **Impact Assessment of IEEE 1471 on the Open Group Architecture Framework**. Integrated Systems and Internet Solutions, Inc. Massachusetts, USA. March (2000)

[Hilliard01b] Hilliard, R.: **Viewpoint Modeling**. In Proc. To 1st ICSE Workshop on Describing Software Architecture in UML. Toronto, Canada. May (2001)

[Hofmeister99] Hofmeister, C., Nord R.L., Soni, D.: **Describing software architecture with UML**. In proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, February (1999)

[IBM01] IBM: **Patterns for e-business**. At http://www.ibm.com/developerworks/patterns (2001)

[IEEE99] IEEE Architecture Working Group. **Recommended Practice for Architectural Description.** IEEE P1471/D5.2 Draft December (1999)

[IEEE00] IEEE Std 1471-2000.: **Recommended Practice for Architectural Description of Software Intensive Systems**. (2000)

[ISO84] ISO 7498: **The OSI Model**. ISO/IEC (1984)

[ISO96] ISO 9126: **Information Technology - Software quality characteristics and metrics**. ISO/IEC (1996)

[Jennings00] Jennings, N. R.: **On Agent-Based Software Engineering.** Artificial Intelligence, 117 (2) 277-296 (2000)

[Jennings01] Jennings, N. R.: **An agent-based approach for building complex software systems.** Comms. Of the ACM, 44 (4) 35-41 (2001)

[Jennings03a] Jennings, N. R..: **Applying Agent Technology**. Proceedings of the 5th European Agent Systems Spring School. EASSS 2003. Barcelona, Spain, February (2003)

[Jennings03b] Jennings, N. R., Bussmann, S.: **Agent-based control systems**. IEEE Control Systems Magazine (2003) (to appear).

[Li00] Li, Y., Benwell, G., Whigham, P.A., Mulgan, N.: **An Agent-oriented software engineering paradigm and the design of a new generation of spatial information system.** The 12th Annual Colloquium of the Spatial Information Research Centre. P.A. Whigham ed. (165-179) Dunedin, New Zealand (2000)

[Liu03] Liu, L.: **Organization Modelling Environment**. http://www.cs.toronto.edu/km/ome/ (Fevereiro de 2003)

[Luck03] Luck, M., McBurney, P., Preist, C.: **Agent Technology: Enabling Next Generation Computing.** AgentLink (2003)

[Luckham95] Luckham, D.C., Augustin, L.M., Kenny, L., Veera, J., Bryan, D., Mann, W.: **Specification and analysis of system architecture**

using Rapide. IEEE Tr on software engineering vol 21 (4):336-355, April (1995)

[Kolp01a] Kolp, M., Castro, J., Mylopoulos, J.: **A social organization perspective on software architectures**. In Proc. of the 1st Int. Workshop From Software Requirements to Architectures (5–12). STRAW'01, Toronto, Canada (2001)

[Kolp01b] Kolp, M., Giorgini, P., Mylopoulos, J.: **A goal-based organizational perspective on multi-agents architectures**. In Proc. of the 8th Int. Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages. ATAL'01, Seattle, USA (2001)

[Kolp02] Kolp, M., Giorgini, P., Mylopoulos, J.: **Information Systems Development through Social Structures**. 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ishia, Italy, July (2002)

[Kotonya97] Kotonya, G., Sommerville, I.: **Requirements engineering – Processes and Techniques**. Chichester, John Willy & Sons (1997)

[Magee95] Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: **Specifying distributed software architecture**. Proc 5th European software engineering conference, ESEC '95, September (1995)

[Maier01] Maier, M.W.: **Software Architecture: Introducing IEEE Standard 1471**. IEEE Computer (107-109), April (2001)

[Medvidovic97] Medvidovic N., Taylor R.N.: **Architecture description languages**. In Software Engineering ESEC/FSE'97, Lecture Notes in Computer Science, Vol. 1301, Zurich, Switzerland, September (1997)

[Medvidovic99a] Medvidovic, N., Egyed A.: **Extending Architectural Representation in UML with View Integration**. Proceedings of the 2nd International Conference on the Unified Modeling Language (UML) (2-16), Fort Collins, CO, October (1999)

[Medvidovic99b] Medvidovic, N., Oreizy, P., Robbins, J.E., Taylor, R.N.: **Using object-riented typing to support architectural design in the C2 style**. Proc 1st Working IFIP Conference on Software Architecture, WICSAI, February (1999)

[Medvidovic00] Medvidovic, N., Rosenblum, D.S., Robbins, J.E., Redmiles D.F.: **Modeling Software Architectures in the Unified Modeling Language**. Computer Science Department, University of Southern California, Los Angeles (2000)

[Mintzberg92] Mintzberg, H.: **Structure in fives: designing effective organizations**. Prentice-Hall (1992)

[Moriconi95] Moriconi, M., Qian, X., Riemenschneider, R.A.: **Correct architecture refinement**. IEEE Trans. Softw. Eng., 21(4):356-372, April (1995)

[Monroe97] Monroe, R. T., Kompanek, A., Melton R., Garlan D.: **Architectural Styles, Design Patterns, and Objects**. IEEE Software (43-52) January (1997)

[Müller96] Müller, J.: **The Design of Intelligent Agents, A Layered Approach**, Springer (1996)

[Newell93] Newell, A.: **Reflections on the Knowledge Level**. Artificial Intelligence, Vol. 59:31-38 (1993)

[OCL97] **Object Constraint Language Specification**, version 1.1, OMG document ad970808 (1997)

[Odell99] Odell, J.: **Objects and Agents: How do they differ?**, working paper v2.2, September (1999)

[Odell00a] Odell, J.: **Agent Technology**, OMG, green paper produced by the OMG Agent Working Group (2000)

[Odell00b] Odell, J., Parunak, H. V. D., Bauer, B.: **Extending UML for Agents**. Proc. of the Agent-Oriented Information Systems Workshop (AOIS'00) at the 17th National conference on Artificial Intelligence (AIII'00) (3-17), Gerd Wagner, Yves Lesperance, and Eric Yu eds., Austin, TX (2000)

[Odell01] Odell, J., Parunak, H.V. D., Bauer, B.: **Representing Agent Interaction Protocols in UML**. Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds. (121-140), Springer-Verlag, Berlin (Held at the 22nd International Conference on Software Engineering (ISCE)) (2001)

[OMG95] OMG: **The Common Object Request Broker: Architecture and Specification (2.0).** Object Management Group (1995)

[OMG] OMG: **Unified Modeling Language 2.0**. Initial submission to OMG RFP ad/00-09-01 (UML 2.0 Infrastructure RFP) and ad/00-09-02 (UML 2.0 Superstructure RFP).: Proposal version 0.63 (draft). http://www.omg.org/. (2002)

[Papasimeon01] Papasimeon, M., Heinze, C.: **Extending the UML for Designing Jack Agents**. In proceedings of ASWEC 2001, Canberra, Australia, August (2001)

[Parunak01] Parunak, H.V. D., Odell., J.: **Representing Social Structures in UML**. Proc. of the Agent-Oriented Software Engineering Workshop, Agents 2001, Montreal, Canada (2001)

[Rumbaugh99] Rumbaugh, J., Jacobson, I., Booch, G.: **The Unified Modeling Language – Reference Manual**. Addison Wesley (1999)

[Scott98] Scott, W. R.: **Organizations: rational, natural, and open systems.** Prentice Hall (1998)

[Segil96] Segil, L.: **Intelligent business alliances: how to profit using today's most important strategic tool.** Times Business (1996)

[Selic98] Selic, B., Rumbaugh, J.: **Using UML for Modeling Complex Real-Time Systems**. Rational Whitepaper, www.rational.com, March (1998)

[Selic99] Selic, B.: **Using UML to Model Complex Real-Time Architectures – Extended Abstract**. ObjectTime Limited, Ontario. (1999)

[Shaw96] Shaw, M., and Garlan, D.: **Software Architecture: Perspectives on an Emerging Discipline**, New Jersey (1996)

[Shaw95] Shaw, M.: **Some Patterns for Software Architecture**. Second Annual Conference on Pattern Languages of Programming, September (1995)

[Shaw00] Shaw, M.: **The Coming-of-Age of Software Architecture Research**. Proceedings of the 23rd International Conference on Software Engineering - ICSE 2001. Keynote address. Toronto, Canada. May (2001)

[Silva02] Silva, C. T. L. L., Castro, J. F. B.: **Modeling Organizational Architectural Styles in UML: The Tropos Case** WER02 - V Workshop on Requirements Engineering, 2002, Valencia, Espanha.

[Silva03] Silva, C. T. L. L., Castro, J. F. B.: **Detailing Architectural Design in the Tropos Methodology.** 2nd Int. Workshop From Software Requirements to Architectures - STRAW'03. Portland, Oregon, USA (2003) (To appear)

[Sommerville01] Sommerville, I.: **Software Engineering** – Ed.6. Addison Wesley (2001)

[Stafford98] Stafford, J.A., Richardson D.J., Wolf A.L.: **Aladdin: A Tool for Architecture-Level Dependence Analysis of Software**. University of Colorado at Boulder, Technical Report CUCS-858-98, April (1998)

[Tahara99] Tahara, Y., Ohsuga, A., Honiden, S.: **Agent System Development Method Based on Agent Patterns**. The 21st International Conference on Software Engineering - ICSE 1999 (356-367), Los Angeles, CA, USA (1999)

[Tveit01] Tveit, A.: **A survey of Agent-Oriented Software Engineering**, First NTNU CSGSC, May (2001)

[Wooldridge95a] Wooldridge, M., Jennings, N. R.: **Intelligent agents: theory and practice** The Knowledge Engineering Review, vol.10 (2):115-152. (1995)

[Wooldridge95b] Wooldridge, M.: **Conceptualising and Developing Agents**. Proceedings of The Agent Software Seminar, UNICOM Seminars, London, April (1995)

[Wooldridge98] Wooldridge, M.: **Agents and software engineering**. AI*IA Notizie XI, 3, September (1998)

[Wooldridge99] Wooldridge, M., Jennings, N.R.: **Software Engineering with agents: Pitfalls and Pratfalls**. Internet Computing 3 (3) 20-27 (1999)

[Wooldridge00] Wooldridge, M., Jennings, N.R., Kinny, D.: **The Gaia Methodology for Agent-Oriented Analysis and Design**, Journal of Autonomous Agents and Multi-Agent Systems 3 (3):285-312 (2000)

[Wooldridge01a] Wooldridge, M., Ciancarini, P.**: Agent-Oriented Software Engineering: The State of the Art.** In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*. Springer-Verlag Lecture Notes in AI Volume 1957, January (2001)

[Wooldridge01b] Wooldridge, M., Weiss, G., Ciancarini, P. editors: **Agent-Oriented Software Engineering II**. Springer-Verlag Lecture Notes in Computer Science Volume 2222, February (2001)

[Wooldridge02] Wooldridge, M.: **Introduction to Multiagent Systems**. John Wiley and Sons, New York (2002)

[Yoshino95] Yoshino, M. Y., Rangan, U. S.: **Strategic Alliances: An Entrepreneurial Approach to Globalization**. Harvard Business School Press (1995)

[Yu95] Yu, E.: **Modelling Strategic Relationships for Business Process Reengineering**, Ph.D. thesis. Dept. of Computer Science, University of Toronto (1995)

[Yu01] Yu, E.: **Agent Orientation as a Modelling Paradigm**, Wirtschaftsinformatik 43 (2): 123-132 (2001)

[Zambonelli00] Zambonelli, F., Jennings, N., Ornicini, A., Wooldridge, M.: **Agent Oriented Software Engineering for Internet Applications** , Published as Chapter 13 in the book: Coordination of Internet Agents: Models, Technologies and Applications, F. Zambonelli, M. Klusch, R. Tolksdorf (Eds.), Springer (2000)

Apêndice A

A seguir, encontra-se um catálogo contendo todos os demais estilos arquiteturais organizacionais representados em UML-RT, incluindo os detalhes dos protocolos definidos por cada um deles.

O estilo *Oferta (Bidding)* envolve mecanismos de competitividade e os componentes se comportam com se eles estivessem tomando parte em um leilão.

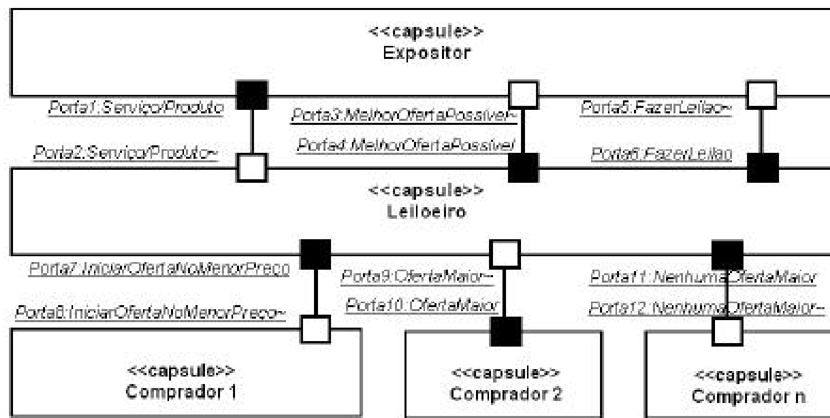


Figura A.1 - Estilo Arquitetural Organizacional Leilão representado em UML-RT

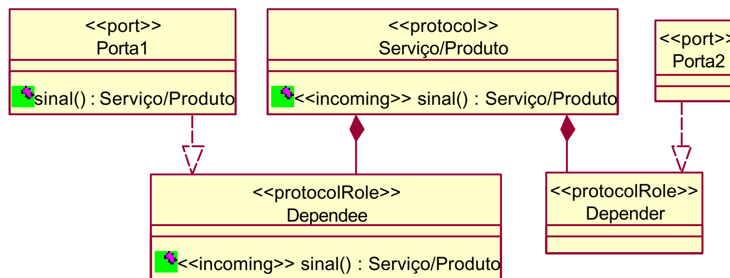


Figura A.2 – Protocolo Serviço/Produto representando uma dependência de recurso entre as cápsulas Leiloeiro e Expositor

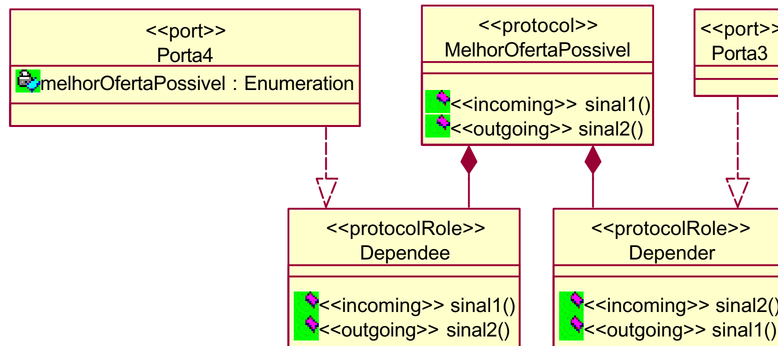


Figura A.3 - Protocolo MelhorOfertaPossivel representando uma dependência de meta-soft entre as cápsulas Expositor e Leiloeiro

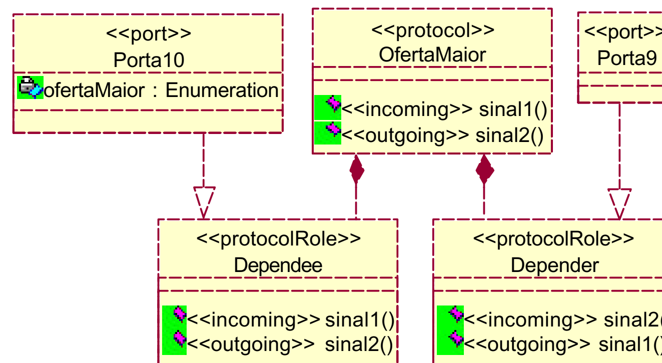


Figura A.4 - Protocolo OfertaMaior representando uma dependência de meta-soft entre as cápsulas Leiloeiro e Comprador 2

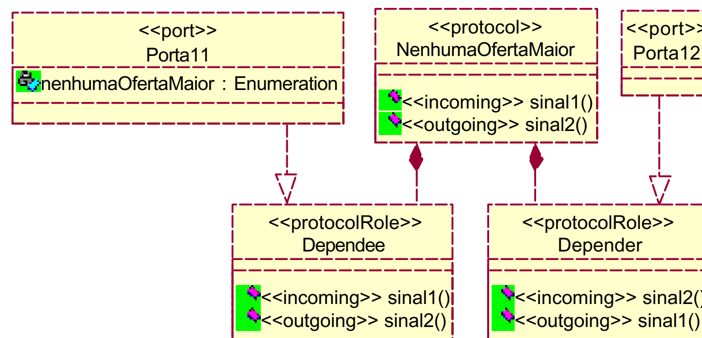


Figura A.5 - Protocolo NenhumaOfertaMaior representando uma dependência de meta-soft entre as cápsulas Comprador n e Leiloeiro

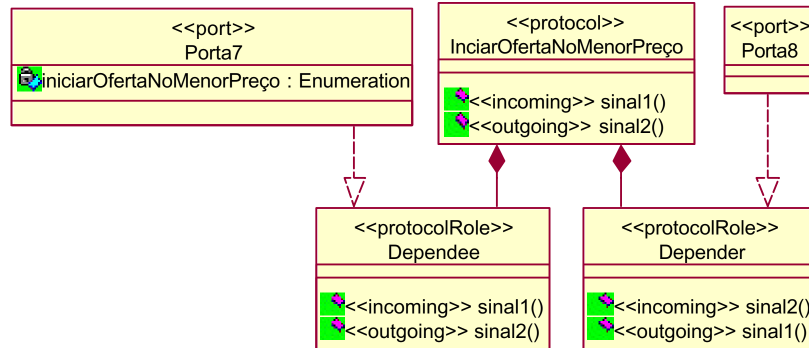


Figura A.6 - Protocolo IniciarOfertaNoMenorPreço representando uma dependência de meta-soft entre as cápsulas Comprador 1 e Leiloeiro

O estilo *Apropriação (Co-optation)*, como próprio nome diz, envolve a incorporação de agentes externos na estrutura ou no comportamento tomador de decisão ou conselheiro de uma organização iniciante.

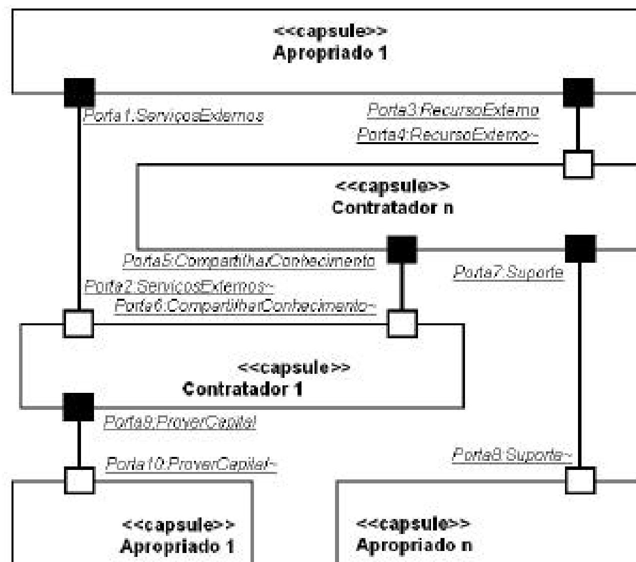


Figura A.7 - Estilo Arquitetural Organizacional Apropriação representado em UML-RT

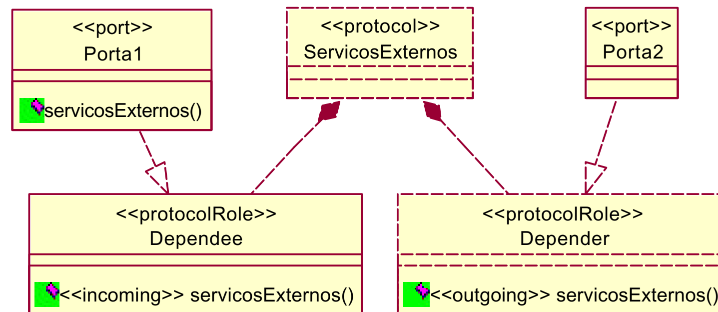


Figura A.8 - Protocolo ServicosExternos representando uma dependência de tarefa entre as cápsulas Contratador 1 e Apropriado 1

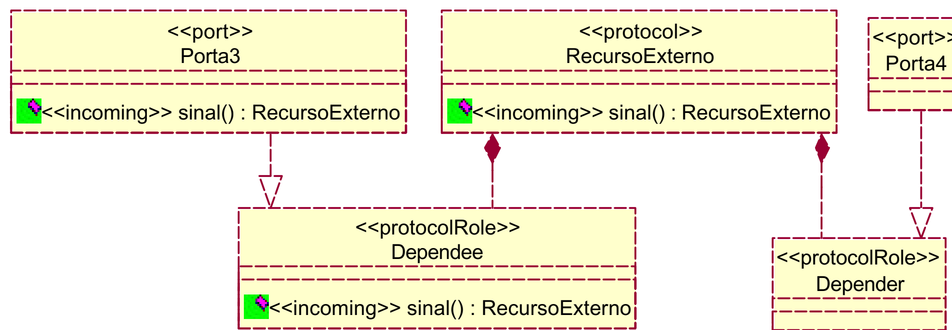


Figura A.9 - Protocolo RecursoExterno representando uma dependência de recurso entre as cápsulas Contratador n e Apropriado 1

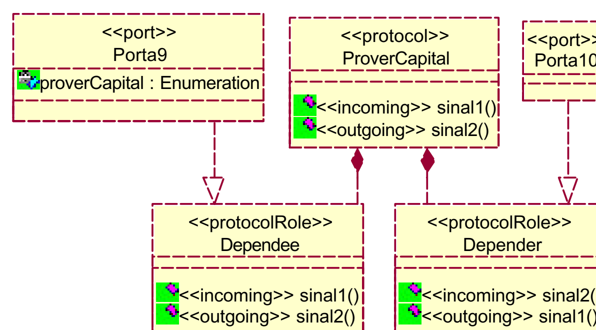


Figura A.10 - Protocolo ProverCapital representando uma dependência de meta-soft entre as cápsulas Apropriado 2 e Contratador 1

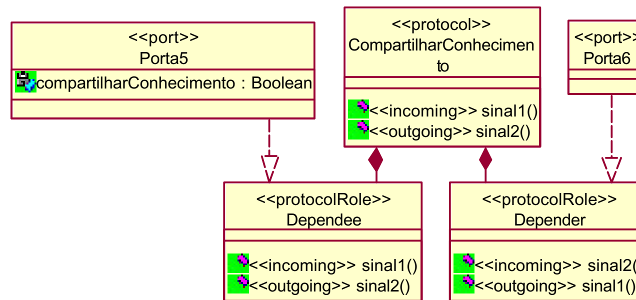


Figura A.11 - Protocolo CompartilharConhecimento representando uma dependência de meta entre as cápsulas Contratador 1 e Contratador n

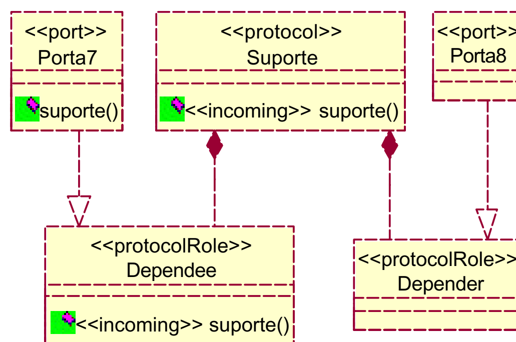


Figura A.12 - Protoc olo Suporte representando uma dependência de tarefa entre as cápsulas Apropriado n e Contratador n

O estilo *Integração Vertical (Vertical Integration)* consiste de componentes comprometidos em atingir metas ou realizar tarefas relacionadas em estágios diferentes de um processo de produção.

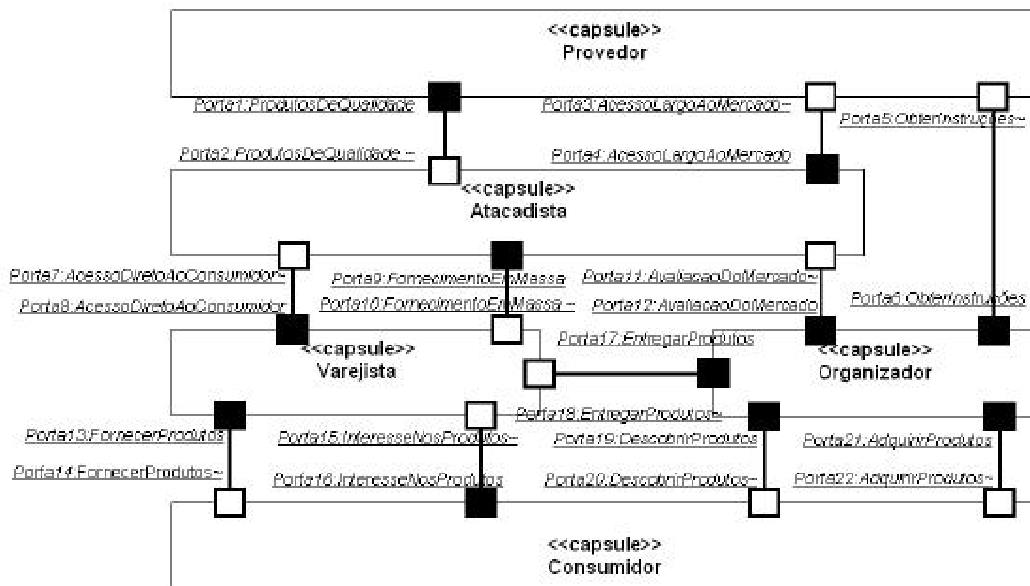


Figura A.13 - Estilo Arquitetural Organizacional Integração Vertical representado em UML-RT

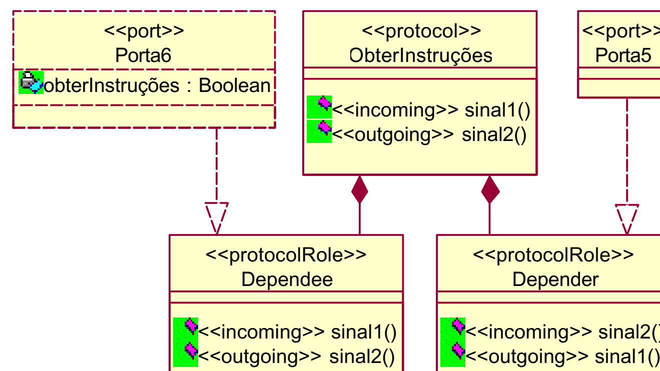


Figura A.14 - Protocolo ObterInstruções representando uma dependência de meta entre as cápsulas Provedor e Organizador

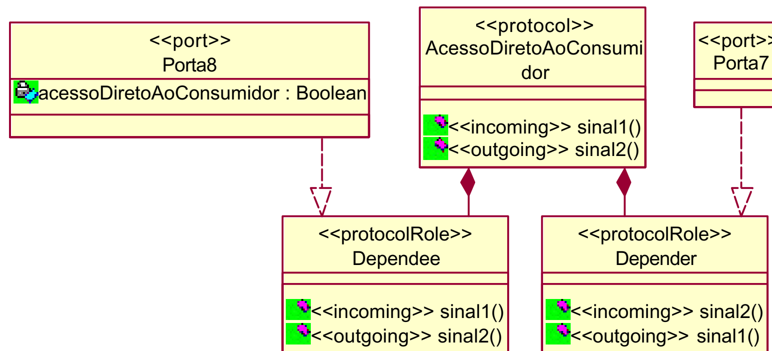


Figura A.15 - Protocolo AcessoDiretoAoConsumidor representando uma dependência de meta entre as cápsulas Atacadista e Varejista

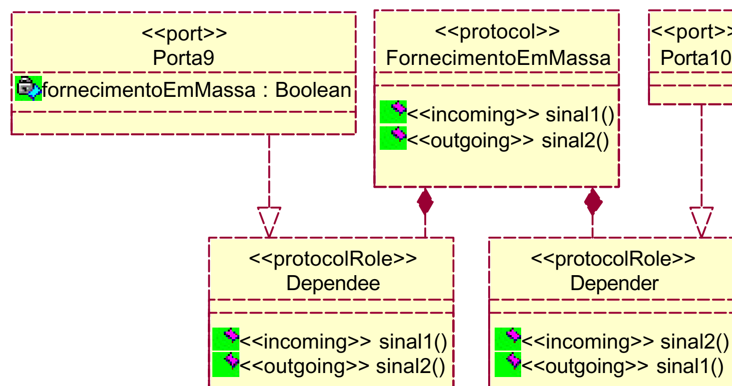


Figura A.16 - Protocolo FornecimentoEmMassa representando uma dependência de meta entre as cápsulas Varejista e Atacadista

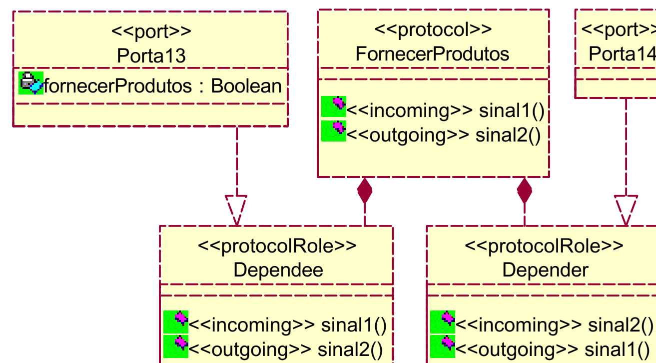


Figura A.17 - Protocolo FornecerProdutos representando uma dependência de meta entre as cápsulas Consumidor e Varejista

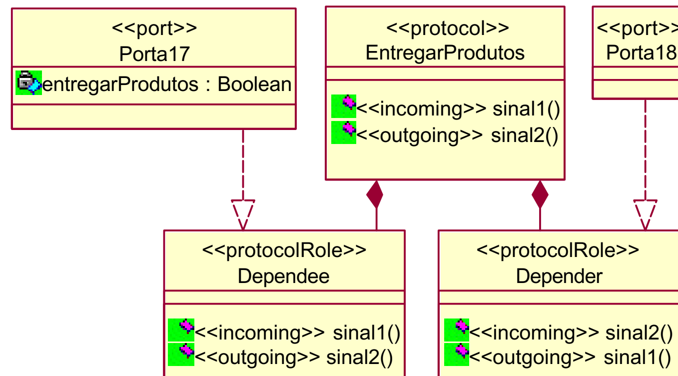


Figura A.18 - Protocolo EntregarProdutos representando uma dependência de meta entre as cápsulas Varejista e Organizador

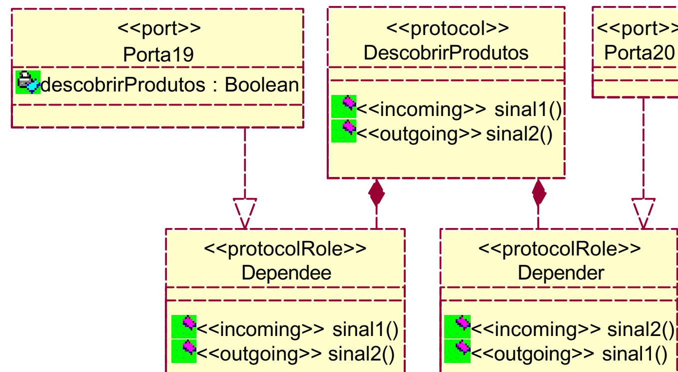


Figura A.19 - Protocolo DescobrirProdutos representando uma dependência de meta entre as cápsulas Consumidor e Organizador

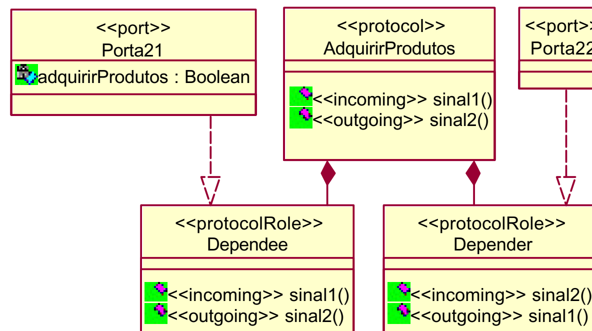


Figura A.20 - Protocolo AdquirirProdutos representando uma dependência de meta entre as cápsulas Consumidor e Organizador

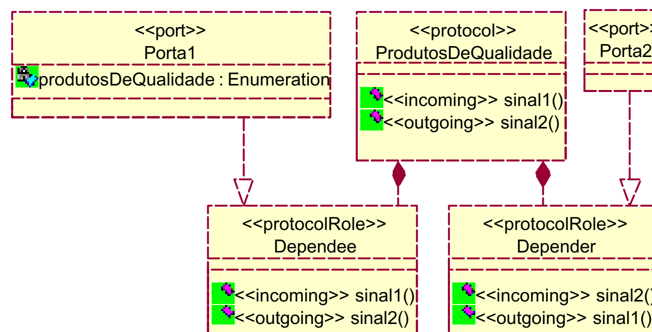
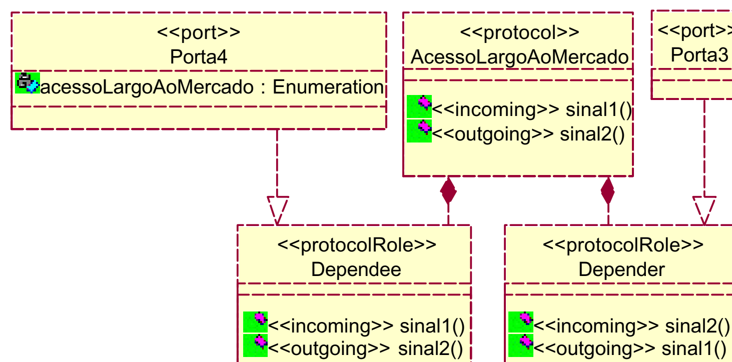


Figura A.21 - Protocolo ProdutosDeQualidade representando uma dependência de meta-soft entre as cápsulas Atacadista e Provedor



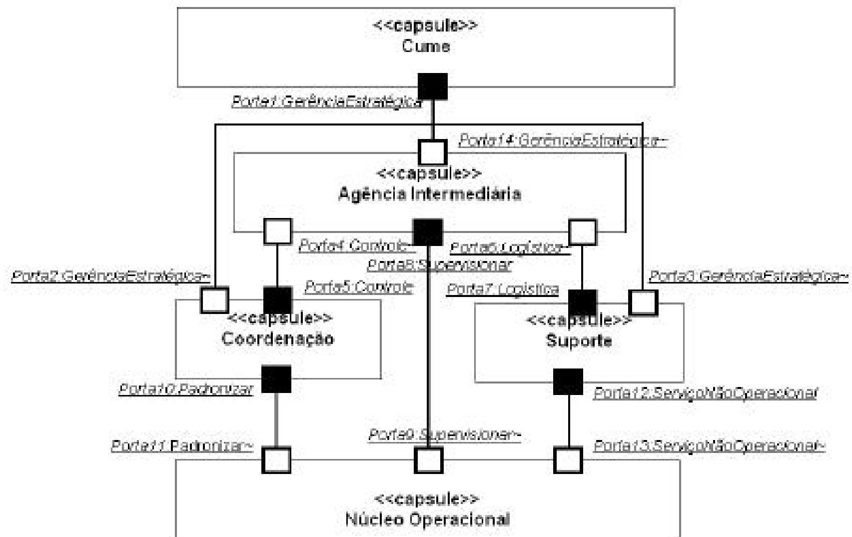


Figura A.25 - Estilo Arquitetural Organizacional Estrutura em Cinco representado em UML-RT.

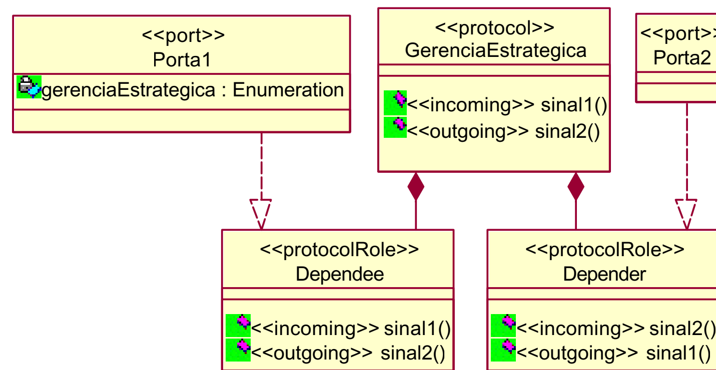


Figura A.26 - Protocolo GerênciaEstratégica representando uma dependência de meta-soft entre as cápsulas Agência Intermediária e Cúpula.

Figura A.22 - Protocolo AcessoLargoAoMercado representando uma dependência de meta-soft entre as cápsulas Provedor e Atacadista

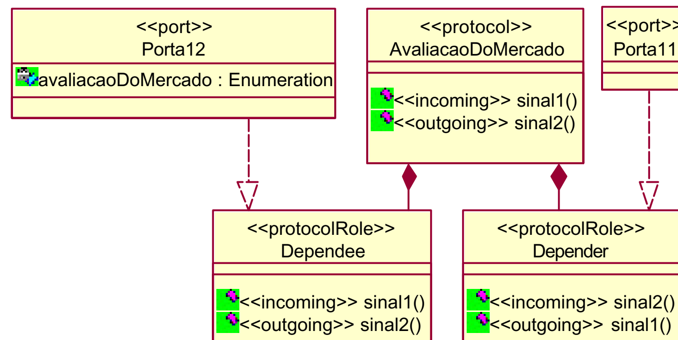


Figura A.23 - Protocolo AvaliaçaoDoMercado representando uma dependência de meta-soft entre as cápsulas Atacadista e Organizador

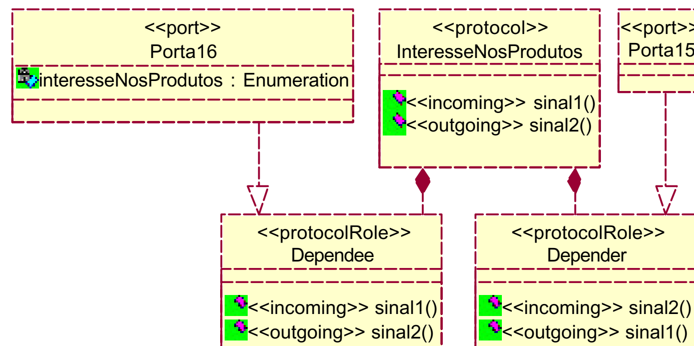


Figura A.24 - Protocolo InteresseNosProdutos representando uma dependência de meta-soft entre as cápsulas Varejista e Consumidor

O estilo Estrutura em 5 (Structure in 5), que consiste dos típicos componentes estratégicos e logísticos geralmente encontrados em várias organizações.

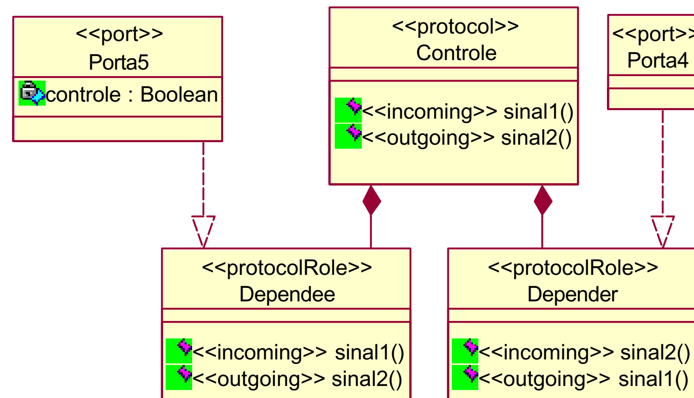


Figura A.27 - Protocolo Controle representando uma dependência de meta entre as cápsulas Agência Intermediária e Coordenação.

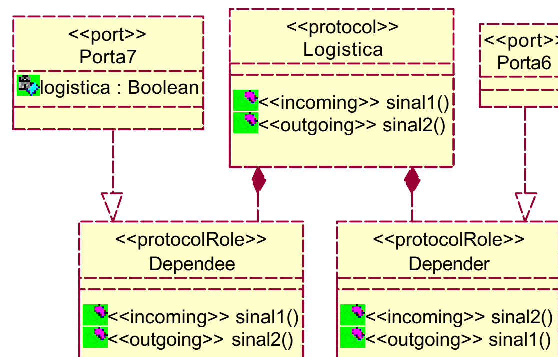


Figura A.28 - Protocolo Logística representando uma dependência de meta entre as cápsulas Agência Intermediária e Coordenação.

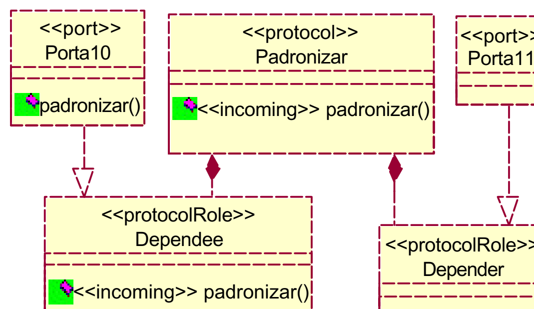


Figura A.29 - Protocolo Padronizar representando uma dependência de tarefa entre as cápsulas Núcleo Operacional e Coordenação.

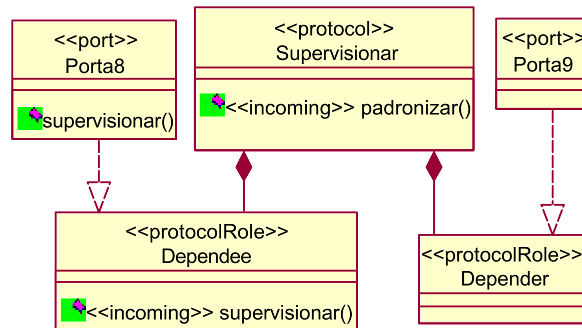


Figura A.30 - Protocolo Supervisionar representando uma dependência de tarefa entre as cápsulas Agência Intermediária e Coordenação.

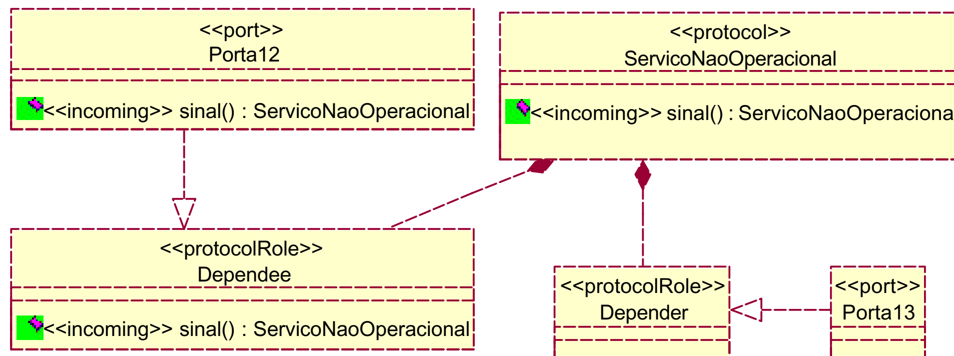


Figura A.31 - Protocolo ServiçoNaoOperacional representando uma dependência de meta entre as cápsulas Núcleo Operacional e Suporte.

O estilo *Contratação Hierárquica (Hierarchical Contracting)* identifica mecanismos de coordenação que combinam características do acordo C*omprimento de Braço* com aspectos de autoridade do estilo *Pirâmide.*

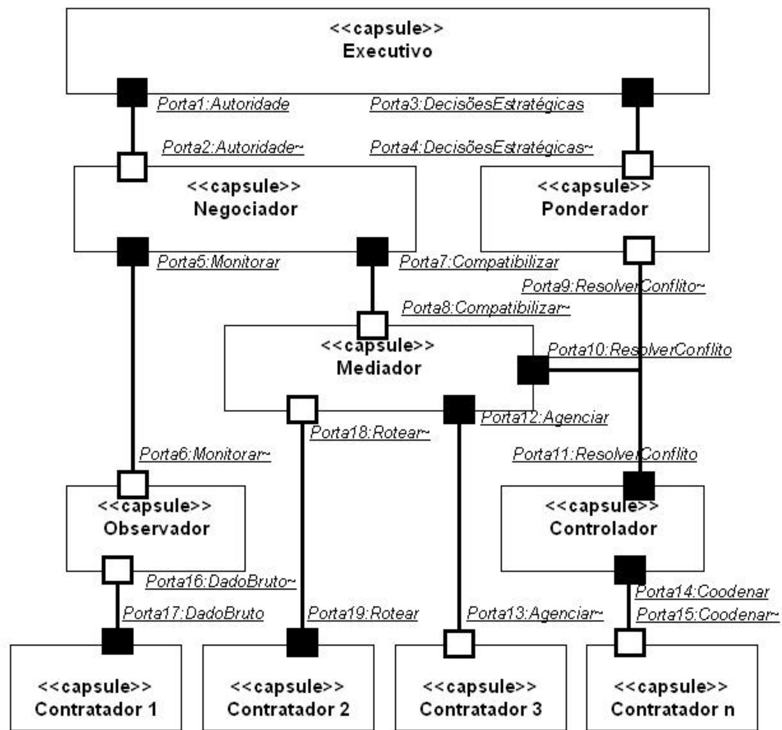


Figura A.32 - Estilo Arquitetural Organizacional Contratação Hierárquica representado em UML-RT

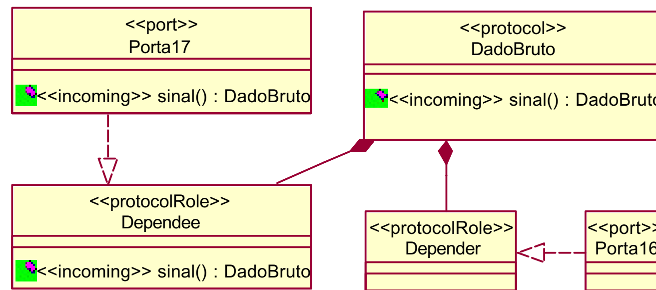


Figura A.33 - Protocolo DadoBruto representando uma dependência de recurso entre as cápsulas Observador e Contratador 1

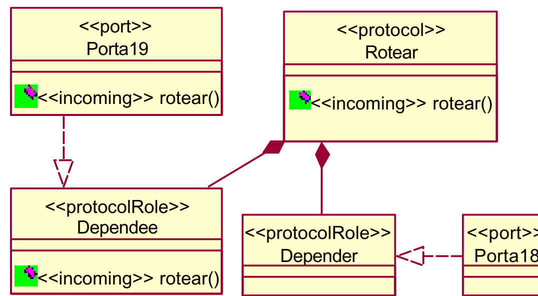


Figura A.34 - Protocolo Rotear representando uma dependência de tarefa entre as cápsulas Mediador e Contratador 2

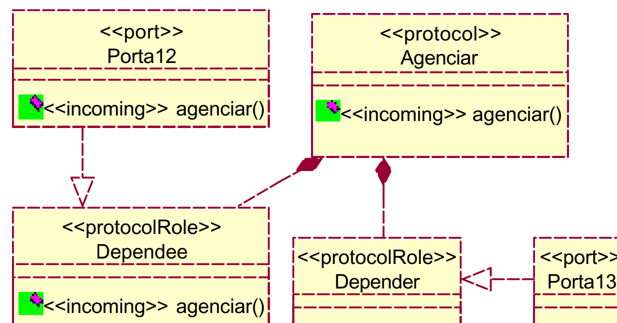


Figura A.35 - Protocolo DadoBruto representando uma dependência de tarefa entre as cápsulas Contratador 3 e Mediador

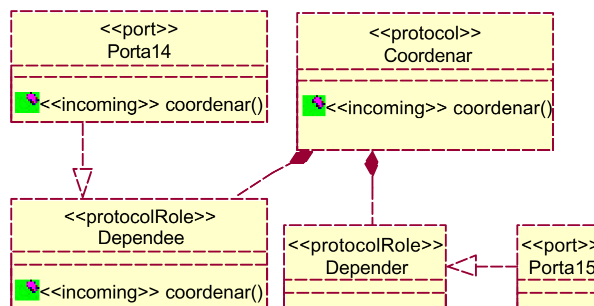


Figura A.36 - Protocolo Coordenar representando uma dependência de tarefa entre as cápsulas Contratador n e Controlador

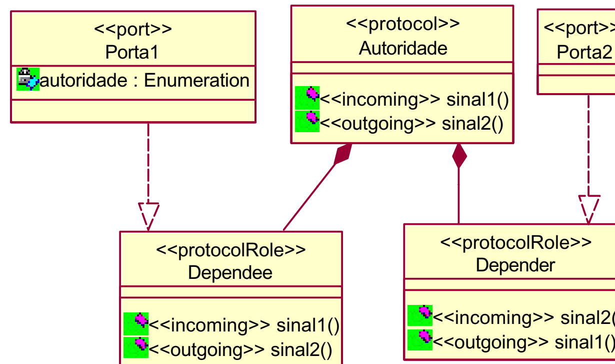


Figura A.37 - Protocolo Autoridade representando uma dependência de meta-soft entre as cápsulas Negociador e Executivo

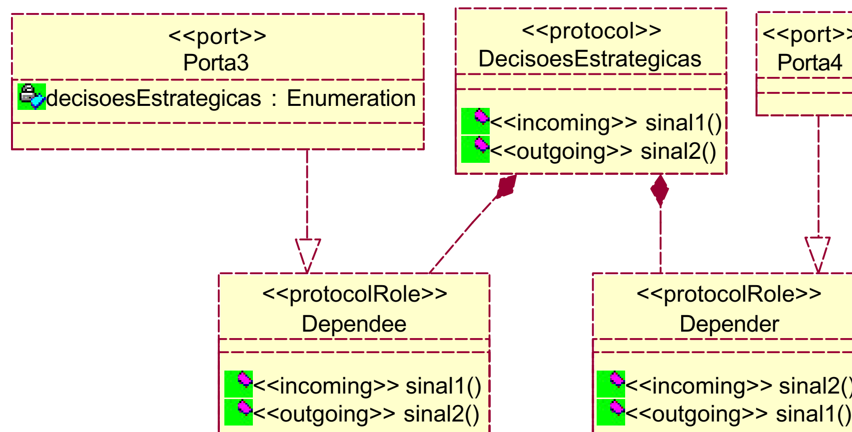


Figura A.38 - Protocolo DecisoesEstrategicas representando uma dependência de meta-soft entre as cápsulas Ponderador e Executivo

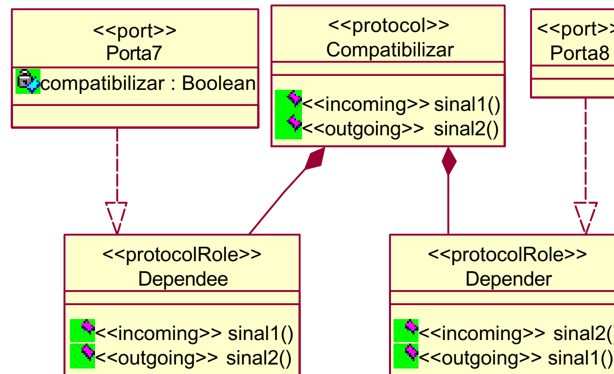


Figura A.39 - Protocolo Compatibilizar representando uma dependência de meta entre as cápsulas Mediador e Negociador

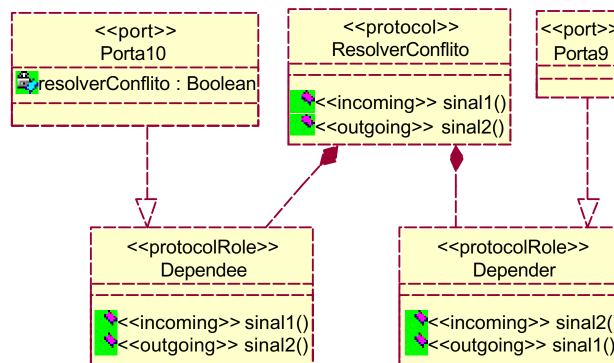


Figura A.40 - Protocolo ResolverConflito representando uma dependência de meta entre as cápsulas Ponderador e Controlador

O estilo C*omprimento de Braço (Arm's Length)* implica em acordos entre componentes independentes e competitivos, porém parceiros.

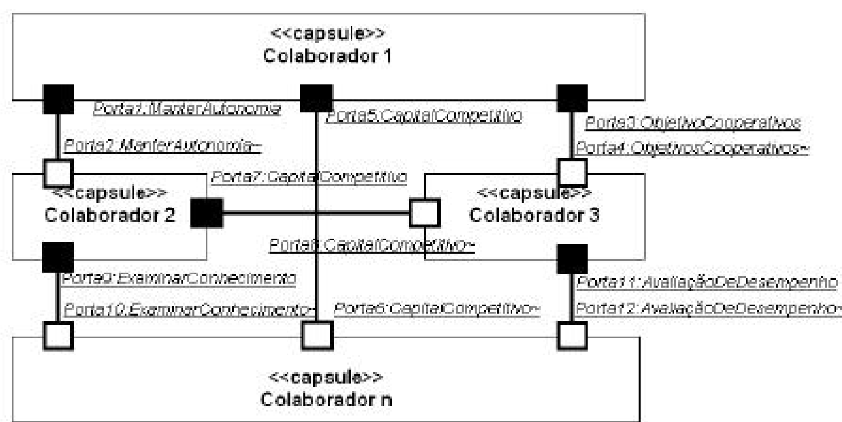


Figura A.41 - Estilo Arquitetural Organizacional Comprimento de Braço representado em UML-RT

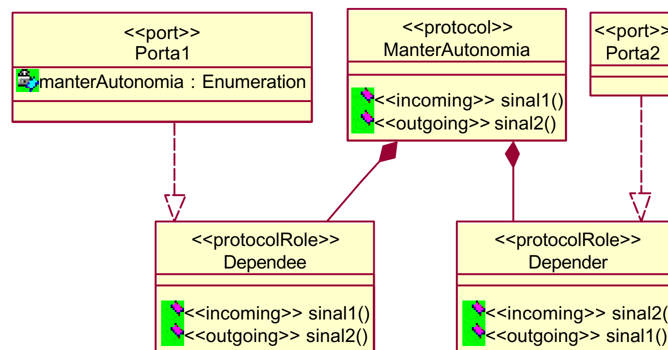


Figura A.42 - Protocolo ManterAutonomia representando uma dependência de meta-soft entre as cápsulas Colaborador 2 e Colaborador 1

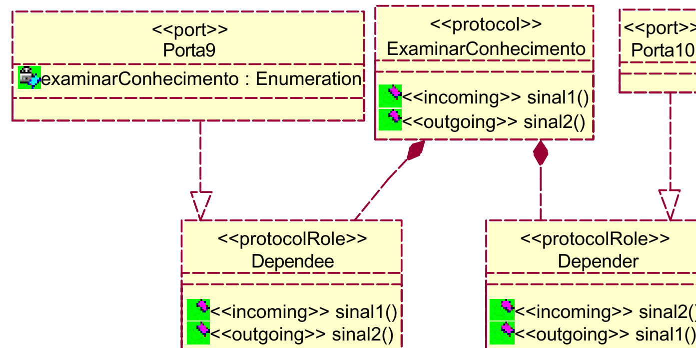


Figura A.43 - Protocolo ExaminarConhecimento representando uma dependência de meta-soft entre as cápsulas Colaborador n e Colaborador 2

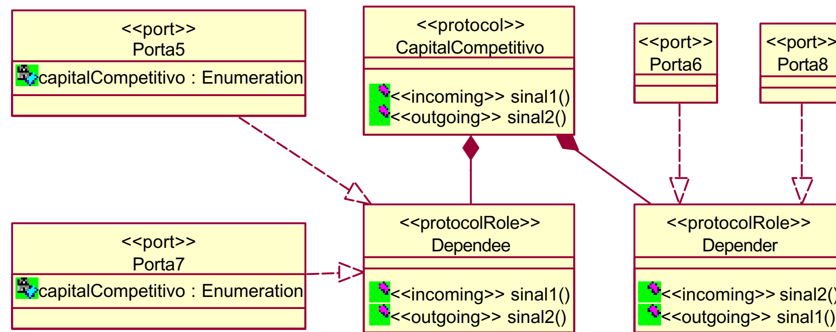


Figura A.44 - Protocolo CapitalCompetitivo representando uma dependência de meta-soft entre as cápsulas Colaborador 1, Colaborador 2, Colaborador 3 e Colaborador n

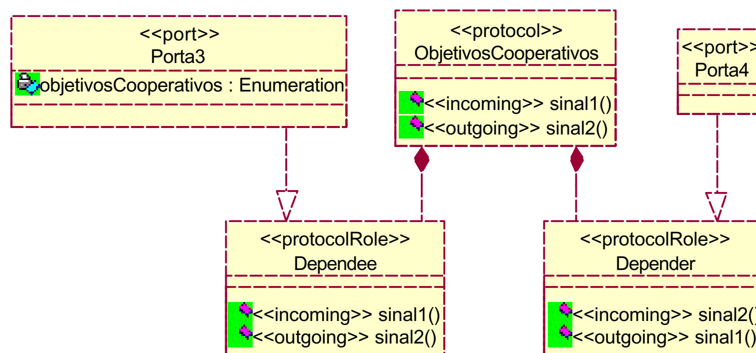


Figura A.45 - Protocolo ObjetivosCorporativos representando uma dependência de meta-soft entre as cápsulas Colaborador 3 e Colaborador 1

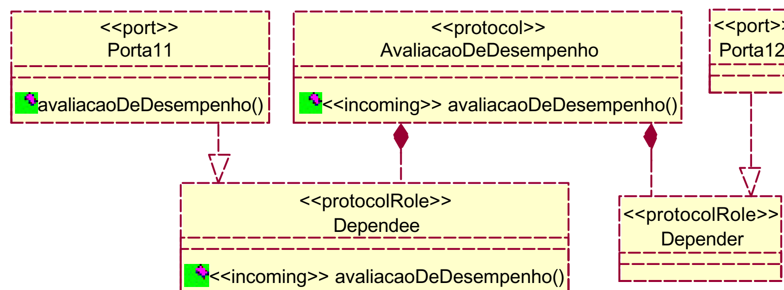


Figura A.46 - Protocolo AvaliacaoDeDesempenho representando uma dependência de tarefa entre as cápsulas Colaborador n e Colaborador 3

O estilo *Tomada de Controle (Takeover)* envolve a delegação total de autoridade e gerenciamento de dois ou mais parceiros para um único componente *Tomador de Controle* coletivo.

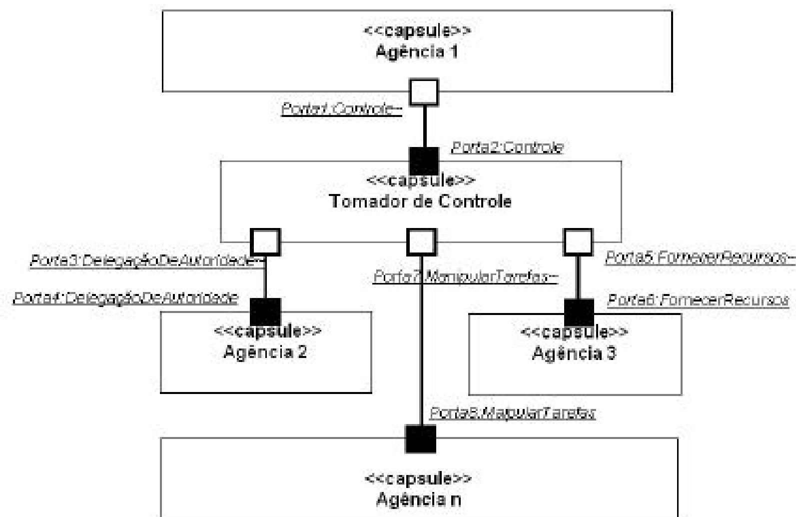


Figura A.47 - Estilo Arquitetural Organizacional Tomada de Controle representado em UML-RT

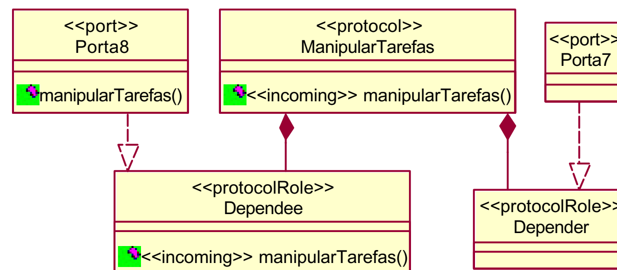


Figura A.48 - Protocolo ManipularTarefas representando uma dependência de tarefa entre as cápsulas Tomador de Controle e Agência n

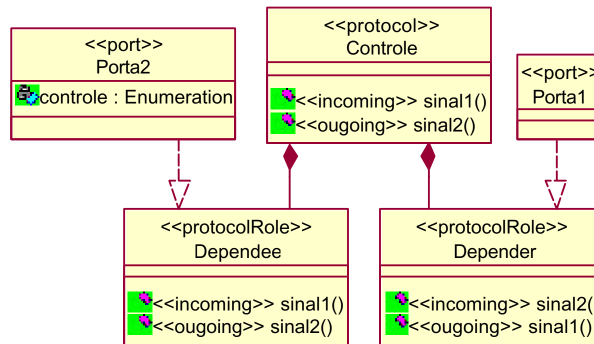


Figura A.49 - Protocolo Controle representando uma dependência de meta entre as cápsulas Agência 1 e Tomador de Controle

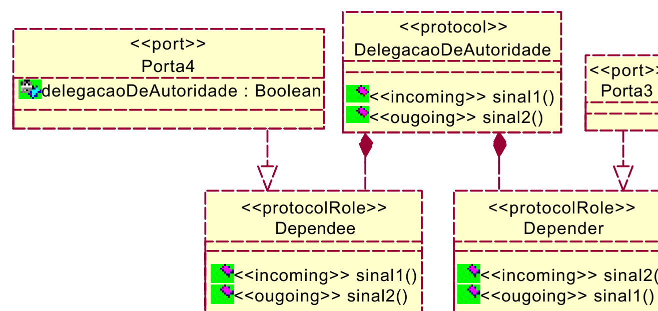


Figura A.50 - Protocolo Delegação de Autoridade representando uma dependência de meta entre as cápsulas Tomador de Controle e Agência 2

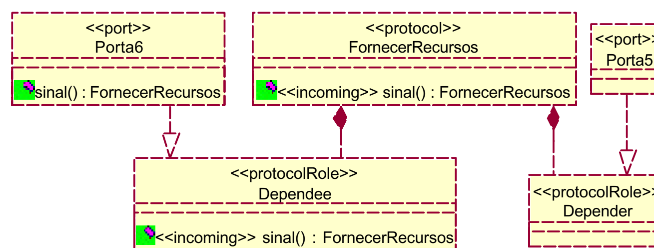


Figura A.51 - Protocolo FornecerRecursos representando uma dependência de recurso entre as cápsulas Tomador de Controle e Agência 3

O estilo *Pirâmide (Pyramid)* é uma estrutura bem conhecida de autoridade hierárquica exercida com limites organizacionais.

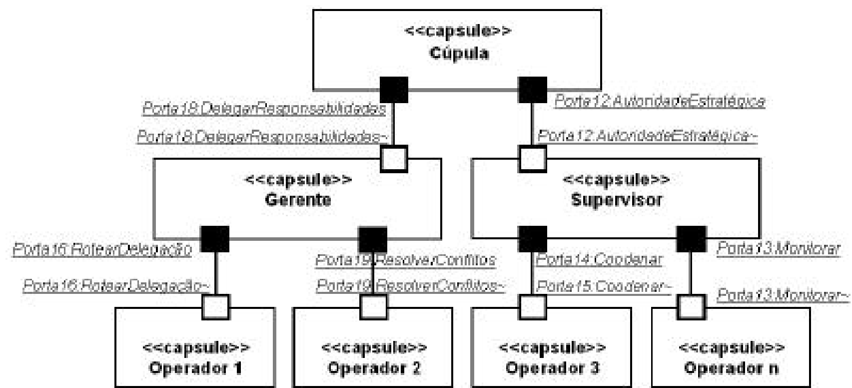


Figura A.52 - Estilo Arquitetural Organizacional Pirâmide representado em UML-RT

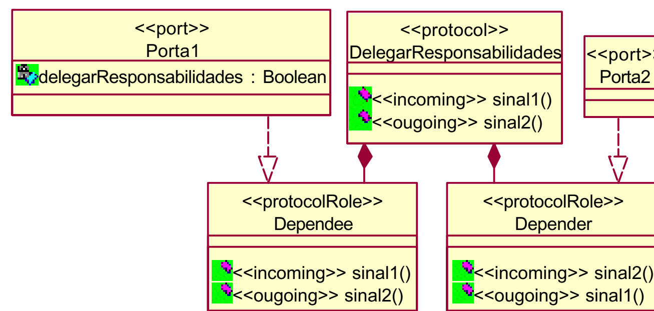


Figura A.53 - Protocolo DelegarResponsabilidades representando uma dependência de meta entre as cápsulas Gerente e Cúpula

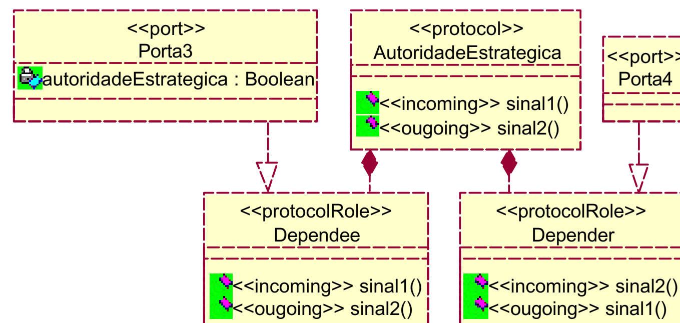


Figura A.54 - Protocolo AutoridadeEstrategica representando uma dependência de meta entre as cápsulas Supervisor e Cúpula

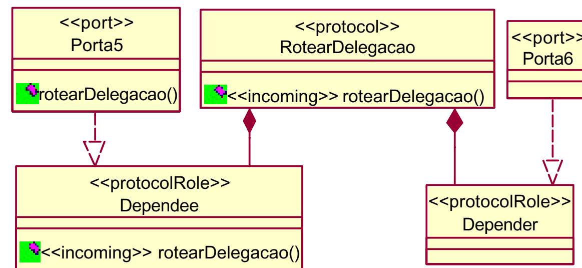


Figura A.55 - Protocolo RotearDelegacao representando uma dependência de tarefa entre as cápsulas Operador 1 e Gerente

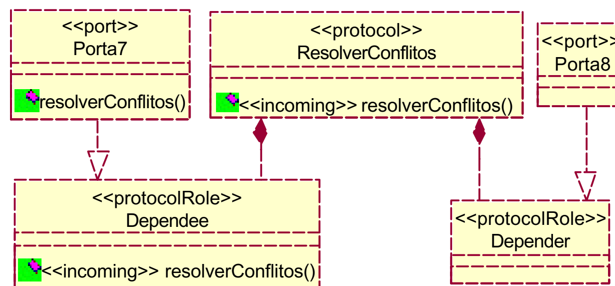


Figura A.56 - Protocolo ResolverConflitos representando uma dependência de tarefa entre as cápsulas Operador 2 e Gerente

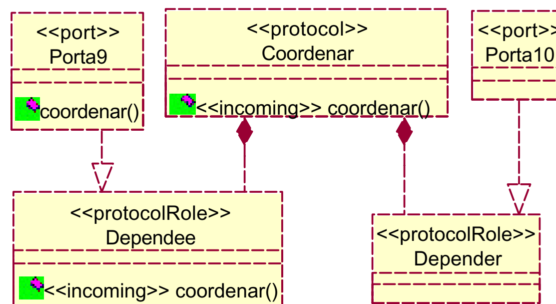


Figura A.57 - Protocolo Coordenar representando uma dependência de tarefa entre as cápsulas Tomador Operador 3 e Supervisor

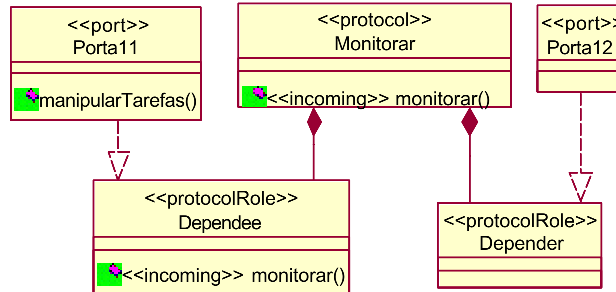


Figura A.58 - Protocolo Monitorar representando uma dependência de tarefa entre as cápsulas Operador n e Supervisor

O estilo *Estrutura Plana (Flat Structure)* não possui estrutura fixa e assume que nenhum componente tenha controle sobre outro.

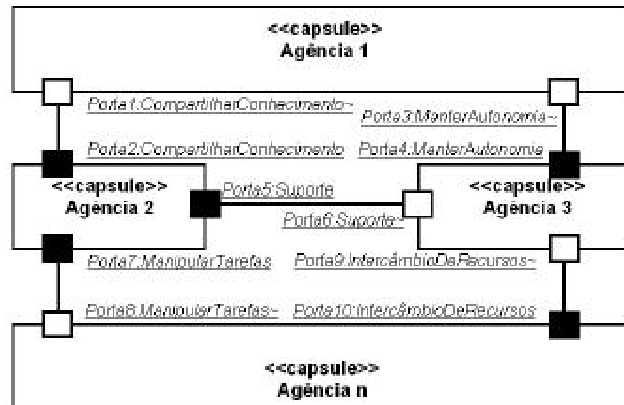


Figura A.59 - Estilo Arquitetural Organizacional Estrutura Plana representado em UML-RT

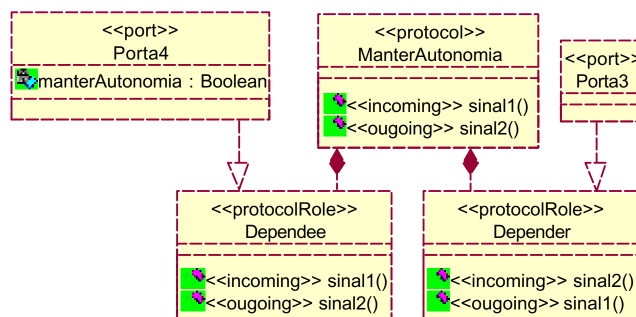


Figura A.60 - Protocolo ManterAutonomia representando uma dependência de meta entre as cápsulas Agência 1 e Agência 3

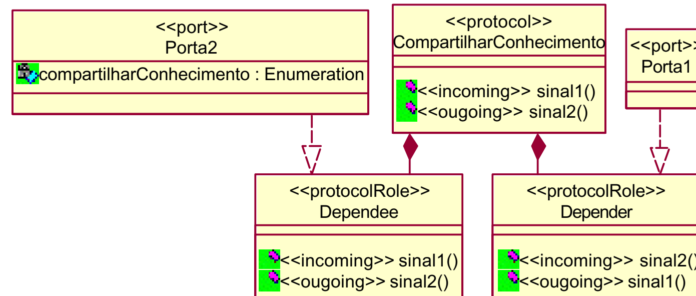


Figura A.61 - Protocolo CompartilharConhecimento representando uma dependência de meta-soft entre as cápsulas Agência 1 e Agência 2

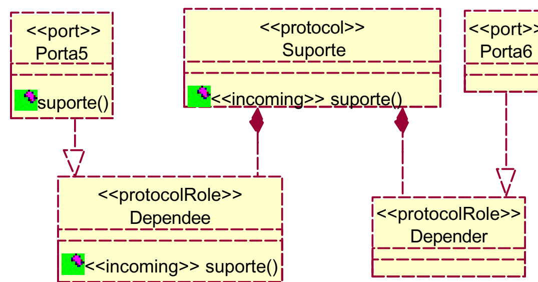


Figura A.62 - Protocolo Suporte representando uma dependência de tarefa entre as cápsulas Agência 3 e Agência 2

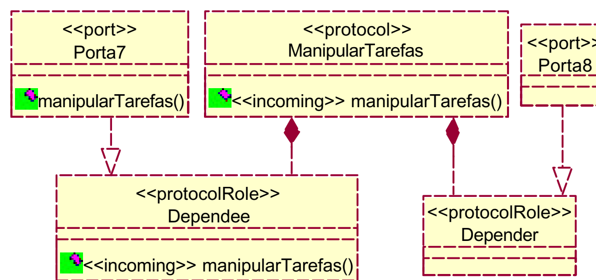


Figura A.63 - Protocolo ManipularTarefas representando uma dependência de tarefa entre as cápsulas Agência n e Agência 2

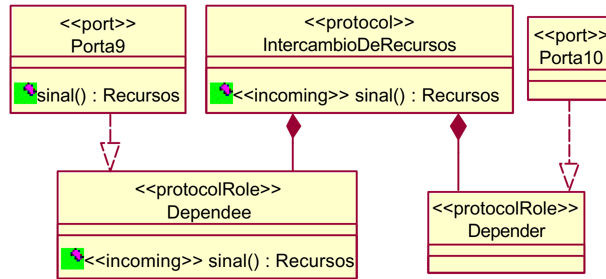


Figura A.64 - Protocolo IntercambioDeRecursos representando uma dependência de recurso entre as cápsulas Agência 3 e Agência n

Apêndice B

A seguir, encontram-se os detalhes de todos os protocolos definidos pela arquitetura do software desenvolvido como estudo de caso nesta dissertação.

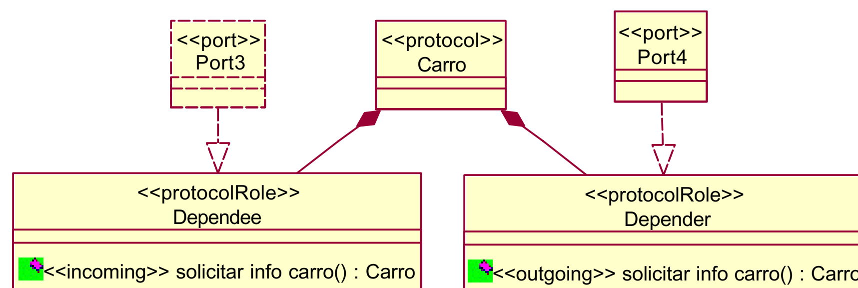


Figura B.1 - Protocolo Carro representando uma dependência de recurso entre as cápsulas Processador de Fatura e Frente de Loja

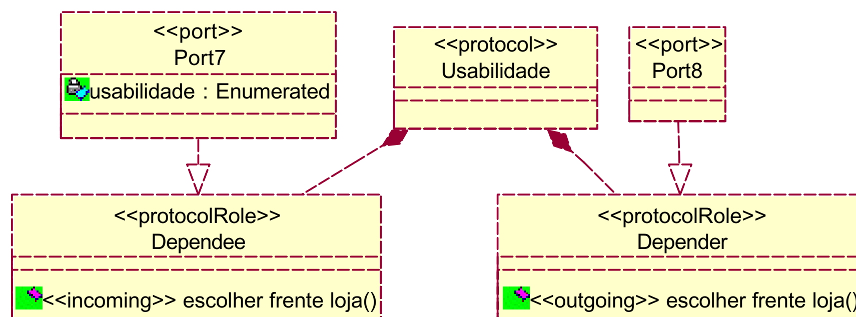


Figura B.2 - Protocolo Usabilidade representando uma dependência de meta-soft entre as cápsulas Frente de Loja e Gerente Comum

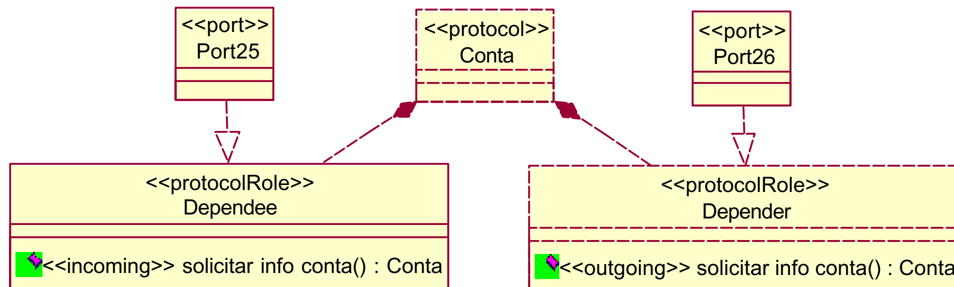


Figura B.3 - Protocolo Conta representando uma dependência de recurso entre as cápsulas Retaguarda de Loja e Processador de Fatura

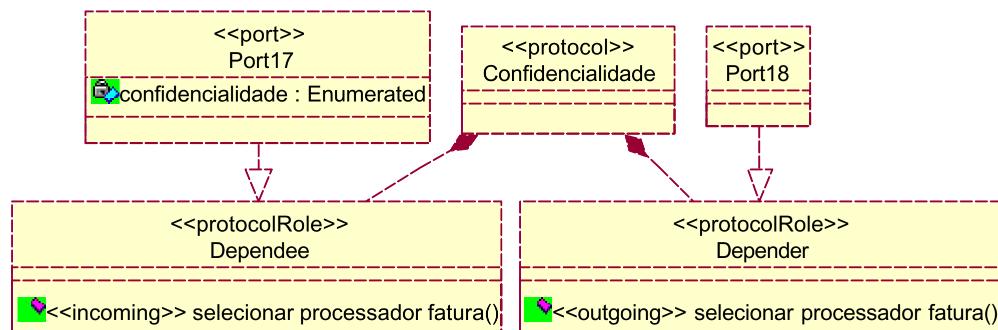


Figura B.4 - Protocolo Confidencialidade representando uma dependência de meta-soft entre as cápsulas Processador de Fatura e Gerente Comum

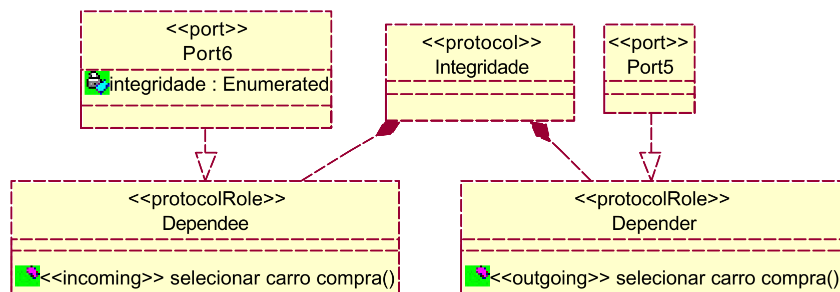


Figura B.5 - Protocolo Integridade representando uma dependência de meta-soft entre as cápsulas Frente de Loja e Gerente Comum

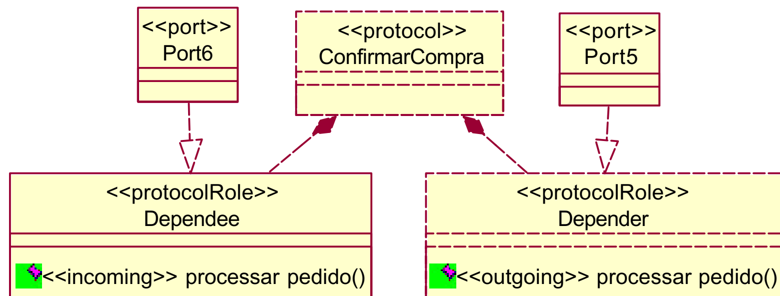


Figura B.6 - Protocolo ConfirmarCompra representando uma dependência de tarefa entre as cápsulas Processador de Fatura e Frente de Loja

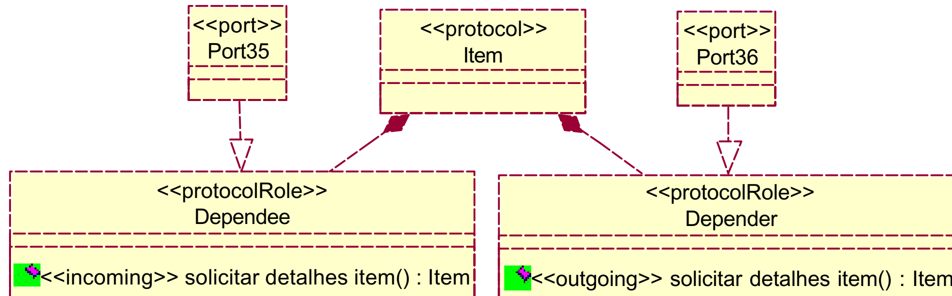


Figura B.7 - Protocolo Item representando uma dependência de recurso entre as sub-cápsulas Carro de Compra e Catálogo On-line

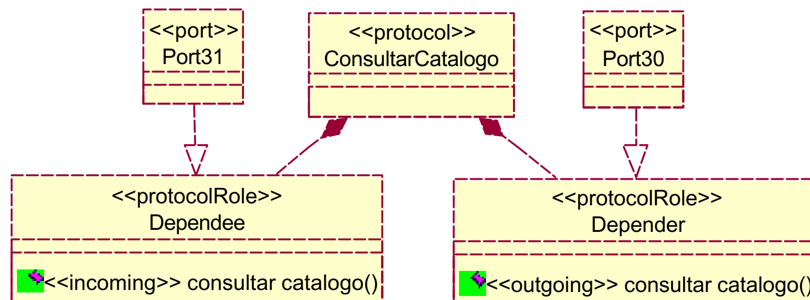


Figura B.8 - Protocolo ConsultarCatalogo representando uma dependência de tarefa entre as sub-cápsulas Navegador de Item e Catálogo On-line

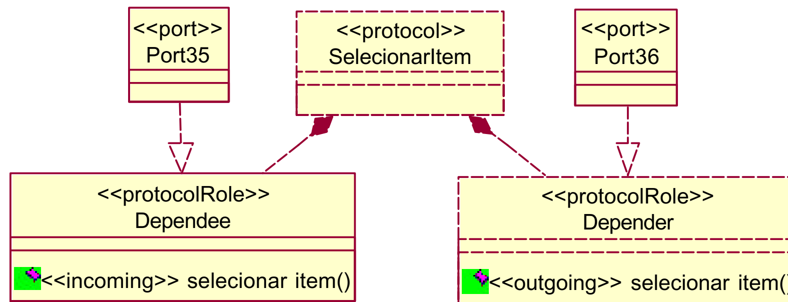


Figura B.9 - Protocolo Selecionar Item representando uma dependência de recurso entre as sub-cápsulas Carro de Compra e Navegador de Item

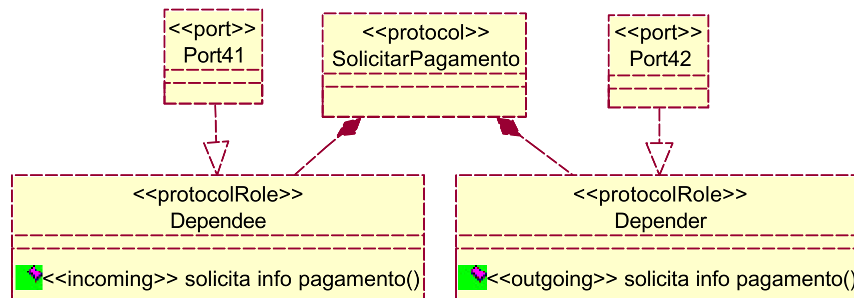


Figura B.10 - Protocolo SolicitarPagamento representando uma dependência de tarefa entre as sub-cápsulas Processador de Pedido e Processador de Conta

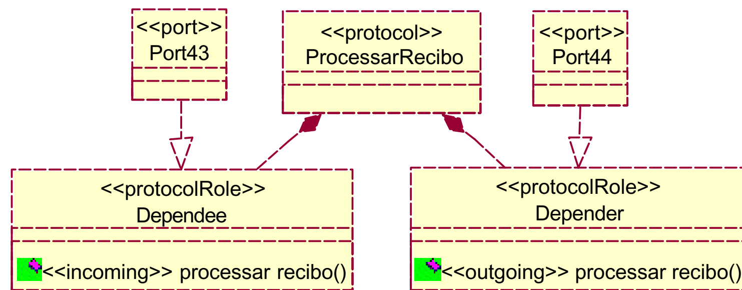


Figura B.11 - Protocolo ProcessarRecibo representando uma dependência de tarefa entre as sub-cápsulas Processador de Pedido e Processador de Recibo

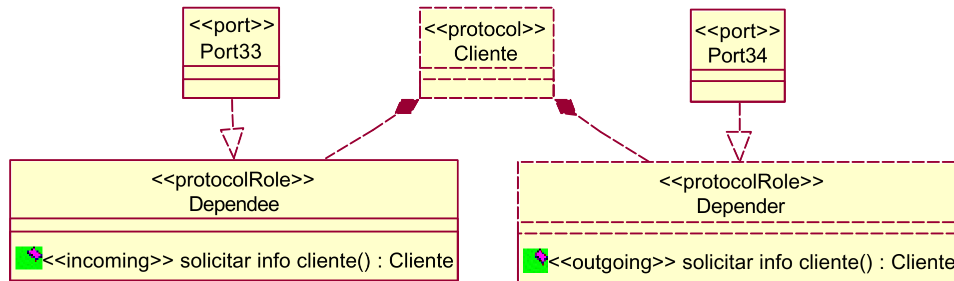


Figura B.12 - Protocolo Cliente representando uma dependência de recurso entre as sub-cápsulas Carro de Compra e Criador de Perfil

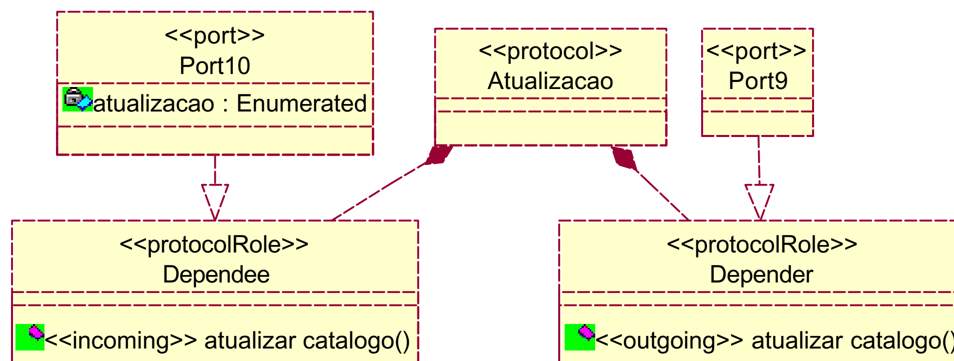


Figura B.13 - Protocolo Atualização representando uma dependência de meta-soft entre as cápsulas Frente de Loja e Gerente Comum

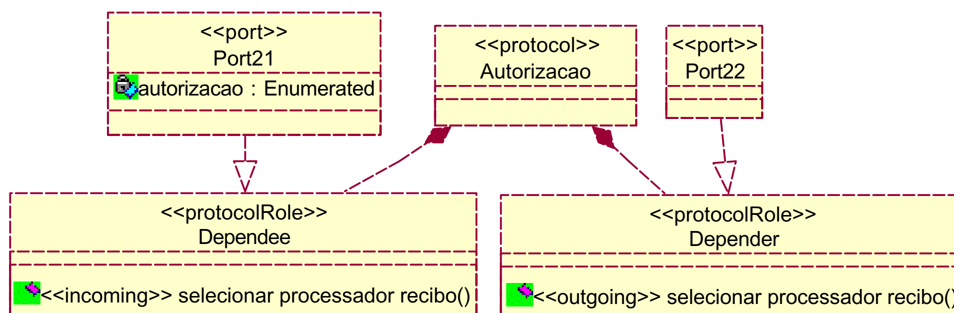


Figura B.14 - Protocolo Autorização representando uma dependência de meta-soft entre as cápsulas Processador de Fatura e Gerente Comum

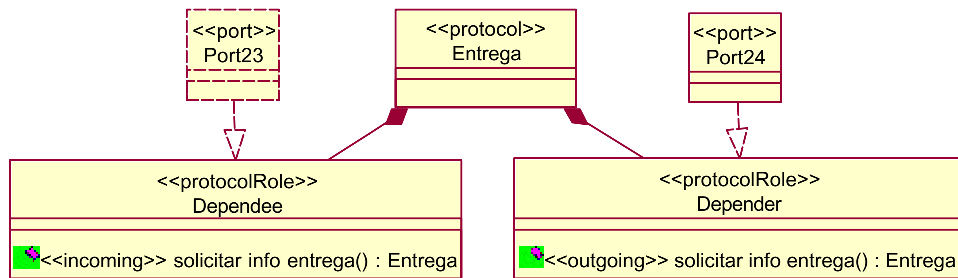


Figura B.15 - Protocolo Entrega representando uma dependência de recurso entre as cápsulas Retaguarda de Loja e Processador de Fatura

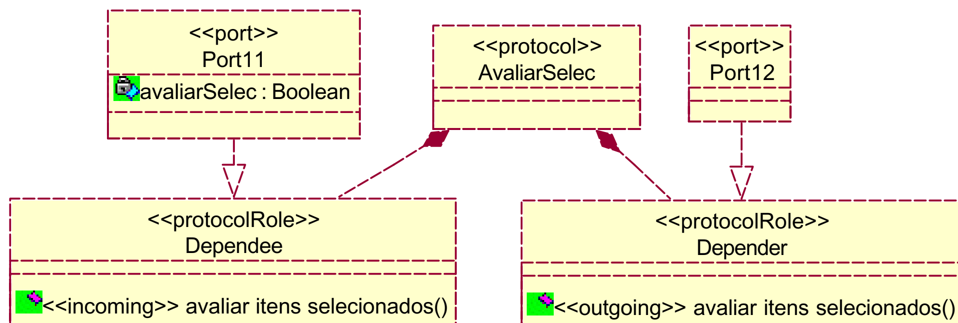


Figura B.16 - Protocolo AvaliarSelec representando uma dependência de meta entre as cápsulas Retaguarda de Loja e Frente de Loja

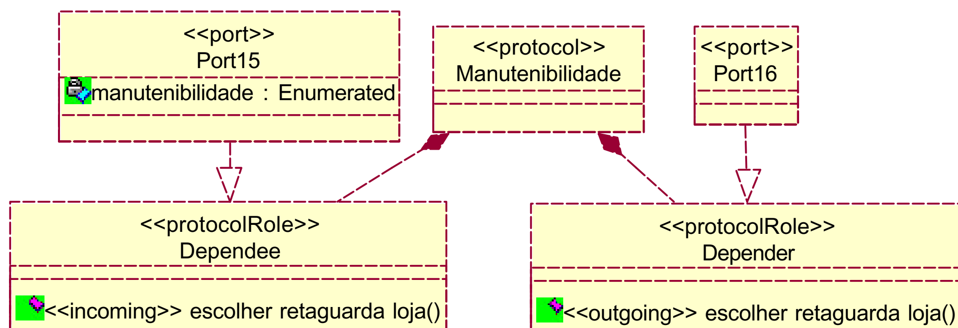


Figura B.17 - Protocolo Manutenibilidade representando uma dependência de meta-soft entre as cápsulas Frente de Loja e Gerente Comum

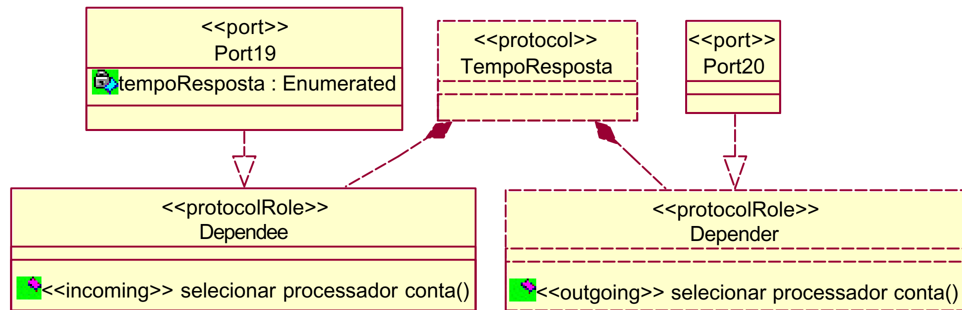


Figura B.18 - Protocolo TempoResposta representando uma dependência de meta-soft entre as cápsulas Processador de Fatura e Gerente Comum

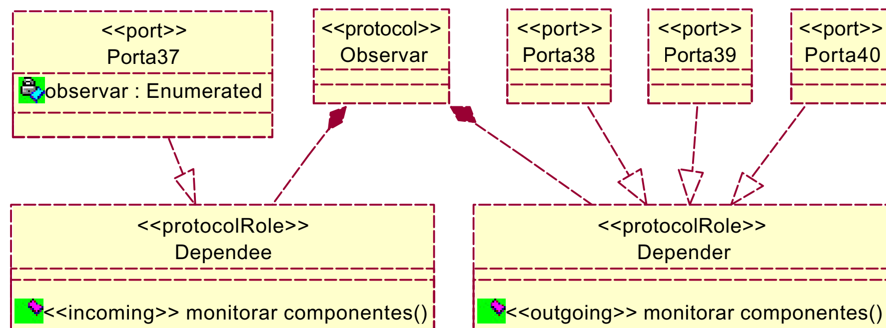


Figura B.19 - Protocolo Observar representando uma dependência de meta-soft entre as sub-cápsulas Gerente de Disponibilidade, Controle de Segurança, Gerente de Adaptação e Monitor