PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

JEFERSON LIBRELOTTO PREVEDELLO

USO DE PLUG-IN PARA INTERAÇÕES MULTIPARTICIPANTES CONFIÁVEIS

Porto Alegre

2008

Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Uso de Plug-in para Interações Multiparticipantes Confiáveis

Jeferson Librelotto Prevedello

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Avelino Francisco Zorzo

Porto Alegre
2008

Dados Internacionais de Catalogação na Publicação (CIP)

P944u Prevedello, Jeferson Librelotto

Uso de plug-in para interações multiparticipantes confiáveis / Jéferson Librelotto Prevedello. – Porto Alegre, 2008.

75 f.

Diss. (Mestrado em Ciência da Computação) – Fac. de Informática, PUCRS.

Orientação: Prof. Dr. Avelino Francisco Zorzo.

1. Informática. 2. Tolerância a Falhas (Computação). 3. Sistemas Distribuídos. I. Zorzo, Avelino Francisco.

CDD 004.36

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada *"**Uso de Plug-ins para Geração de Interações Multiparticipantes Confiáveis**"*, apresentada por Jeferson Librelotto Prevedello, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 18/03/08 pela Comissão Examinadora:

Prof. Dr. Avelino Francisco Zorzo – Orientador — PPGCC/PUCRS

Prof. Dr. Eduardo Augusto Bezerra – — PPGCC/PUCRS

Prof. Dr. Fabian Luis Vargas – — FENG/PUCRS

Homologada em 19/08/08, conforme Ata No 012/08 pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

PUCRS

Campus Central
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

Agradecimentos

Eu gostaria de expressar minha sincera gratidão com algumas pessoas que contribuíram de diversas maneiras para a conclusão desta dissertação. Primeiramente e principalmente agradeço ao meu orientador, Professor Avelino Francisco Zorzo, pela dedicação e responsabilidade com que exerceu sua função. Em todos os momentos se mostrou uma pessoa extremamente flexível, de fácil acesso e sempre disposta a cooperar e ajudar independente da situação. Seus conselhos, sua paciência, seu conhecimento e principalmente sua amizade foram fundamentais para a conclusão deste trabalho.

Meu agradecimento especial à todos os colegas do mestrado, em especial aos amigos Fabio Diniz, Rafael Antonioli e Felipe Franciosi pela ajuda e apoio nos momentos mais difíceis do curso.

Não poderia deixar de mencionar a compreensão e o incentivo recebido da minha namorada Simone. Seu apoio incondicional em todos os momentos foi decisivo para que esta dissertação fosse finalizada.

Meu Muito Obrigado a empresa *Terra Networks*, em especial ao meu gerente Arisneto Cavalcante, pela flexibilização das jornadas de trabalho e pelo constante incentivo.

Por fim, mas não menos importante meu sincero agradecimento à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) a qual forneceu suporte financeiro integral durante todo o período do mestrado.

Resumo

A complexidade de sistemas de software tem aumentado devido aos novos requisitos impostos pelas aplicações modernas, tais como confiabilidade, segurança e disponibilidade. Sistemas confiáveis são sistemas que mantém seu funcionamento de acordo com sua especificação mesmo na presença de situações excepcionais. Na tentativa de implementar sistemas mais robustos e confiáveis, torna-se imprescindível a utilização de mecanismos capazes de lidar com problemas que potencialmente possam afetar seu perfeito funcionamento.

Variados tipos de defeitos e situações inesperadas podem ocorrer em aplicações que rodam sobre sistemas distribuídos. Para que seja atingido um grau satisfatório de utilização destes sistemas é extremamente importante que sejam utilizadas técnicas objetivando coibir ou minimizar a existência de falhas. Tolerância a Falhas é uma técnica que tem por objetivo oferecer alternativas que permitam ao sistema manter o funcionamento conforme sua especificação, mesmo na ocorrência de situações indesejadas.

A literatura descreve diversos tipos de mecanismos que auxiliam no desenvolvimento de aplicações que possuem diversas atividades acontecendo simultaneamente. Em geral, um mecanismo composto por diversos participantes (objetos ou processos) executando um conjunto de atividades paralelamente é chamado de interação multiparticipante. Em uma interação multiparticipante diversos participantes de alguma maneira "se unem"para produzir um estado combinado, intermediário e temporário e utilizam este estado para executar alguma atividade. Após a atividade executada a interação é desfeita e então cada participante prossegue sua execução.

Entretanto, diversas vezes a interação entre os participantes pode levar a situações onde toda a execução tem que ser refeita (efeito dominó). Para evitar este tipo de situação e para auxiliar no tratamento de exceções concorrentes que podem ocorrer nos diversos participantes de uma interação pode-se utilizar, por exemplo, o mecanismo de interações multiparticipantes confiáveis (*Dependable Multiparty Interactions - DMIs*). Este mecanismo tem sido utilizado para o desenvolvimento de aplicações em diversas áreas. Todavia, percebemos que todos os estudos de casos desenvolvidos utilizando DMIs foram implementados manualmente, ou seja, sem a utilização de nenhuma ferramenta de apoio. Tal situação além de acarretar um elevado tempo de desenvolvimento também facilita à inclusão de falhas no sistema.

Nesta dissertação apresentamos uma proposta de desenvolvimento de aplicações confiáveis que utilizam o mecanismo de DMIs. Utilizando o IDE Eclipse desenvolvemos uma feramenta capaz de automatizar o processo de criação de aplicações que utilizam DMIs para tolerar falhas, proporcionando aos desenvolvedores ganho de produtividade, redução da possibilidade de inserção de falhas no código, assim como facilitar a compreensão dos elementos que compõem uma DMI e a maneira como os mesmos estão relacionados.

Abstract

Current software systems are complex. This complexity is augmented due to reliability, availability and security needs. Dependable systems are systems that work acording to their specification despite the existence of faults. In order to implement such systems, it is important to use mechanisms that cope with problems that can happen during their execution.

Diverse types of defects and non-expected situations can happen in an application executing in a distributed manner. To cope with these situations or defects one should use techniques to avoid or reduce there effects. Fault tolerance is one of such techniques. Its main goal is to offer alternatives to a system to keep working according to its specification despite residual faults in the system. The complexity to achieve dependability is even greater in systems in which several activites can be happening at the same time (parallel systems).

Several mechanisms that are used to develop parallel applications are described in the literature. Usually, these mechanisms are called multiparty interactions. These multiparty interactions are composed by several parties (objects or processes) that somehow come together to produce a temporary combined state and execute a joint activity. When the activity is finished they continue their own execution.

However, several times the interacton among the participants can lead the systems to a situation in which all the execution has to be redone (the domino effect). To avoid such situation and to help in the handling of concurrent exceptions when several participants are working together it is possible to use, for example, the Dependable Multiparty Interactions (DMIs). This mechanism has been used in the development of several applications in different areas. Nevertheless we noticed that in all applications, DMIs were applied in a ad hoc situation, that is, they were hard coded manually. Such situation can make the development phase quite tiresome and can also be prone to the introduction of faults in the final system.

This work presents a proposal of an environment for the development of dependable applications that use the DMI mechanism. This environment uses the Eclipse Plug-in Development Environment (PDE). We include to the Eclipse PDE a new plug-in whose main goal is to automatize the development of applications that use DMIs as the means for fault tolerance. This proposal intends to improve developers productivity, to reduce the number of residual faults in the code, and also to easy the understanding of DMIs components and their inter-relations.

Lista de Figuras

Figura 1 Interação Multiparticipante Confiável. 23
Figura 2 Representação UML do *framework*. 25
Figura 3 Gerentes Instanciados para a mesma DMI. 26
Figura 4 Representação de ações atômicas coordenadas. 29
Figura 5 Os componentes e seus relacionamentos. 30
Figura 6 Estruturação das CAAs para o sistema de tratamento de insulina. 31
Figura 7 Possível execução de CAA aninhadas no sistema de controle da Célula de Produção. 32
Figura 8 CAA que controlam a Célula de Produção Tolerante a Falhas. 33
Figura 9 Arquitetura da Plataforma Eclipse. 36
Figura 10 Estrutura dos *plug-ins* do Eclipse. 40
Figura 11 Arquivo *plug-in.properties*. 40
Figura 12 Ligação do plug-in com o Eclipse. 48
Figura 13 Dependências para criação do *plug-in* DMI. 49
Figura 14 DMI com dois papéis. 49
Figura 15 *Template* para a geração dos objetos `Manager`. 51
Figura 16 Estrutura dos pacotes/classes. 52
Figura 17 Estrutura dos pacotes/classes. 53
Figura 18 Ponto de Extensão para invocar o *plug-in* DMI. 53
Figura 19 Tela inicial do *plug-in* DMI. 54
Figura 20 Criando uma DMI. 54
Figura 21 Criando papéis. 55
Figura 22 Criando Objeto Compartilhado. 56
Figura 23 Papéis que receberão Objeto Compartilhado. 56
Figura 24 Criando Exceção. 57
Figura 25 Relacionando Exceção com DMI. 58
Figura 26 Associando papéis a uma DMI Aninhada. 58
Figura 27 Aninhando DMIs. 59
Figura 28 Problema do Jantar dos Filósofos. 60
Figura 29 Ação Comer. 60
Figura 30 DMI Janta - engloba todos filósofos e garfos. 63
Figura 31 DMI aninhada Comer - executada por um filósofo e dois garfos. 64
Figura 32 Criação dos papéis para as DMIs Janta e Comer. 64
Figura 33 Associação da DMI Comer como aninhada da DMI Janta. 65
Figura 34 Criação de um tratador de exceção para DMI Janta. 65
Figura 35 Tratador de exceção para DMI Janta. 66
Figura 36 Criação de objetos compartilhados. 66
Figura 37 Estrutura de diretórios e pacotes para DMIs do problema dos filósofos. . 67
Figura 38 Definição do papel da DMI Comer que usa objetos compartilhados. . . 68

Figura 39 Código gerado o papel G0 - responsável pela criação da DMI Comer. . 70
Figura 40 Código gerado para o papel Filósofo da DMI Comer. 71
Figura 41 Código gerado para o papel GarfoEsquerda da DMI Comer. 71
Figura 42 Código gerado para o papel GarfoDireita da DMI Comer. 72

Lista de Siglas

DMIs	*Dependable Multiparty Interactions*	18
PDE	*Plug-in Development Environment*	19
UML	*Unified Modelling Language*	24
CAA	*Coordinated Atomic Actions*	28
IDE	*Integrated Development Environment*	35
SWT	*Standard Widget Toolkit*	35
JDT	*Java Development Tooling*	39

SUMÁRIO

1 **INTRODUÇÃO** ... 6

2 **INTERAÇÕES MULTIPARTICIPANTES CONFIÁVEIS** ... 21
2.1 IMPLEMENTAÇÃO DE DMI ... 23
2.2 FRAMEWORK PARA IMPLEMENTAÇÃO DE DMI ... 24
2.2.1 **Gerentes** ... 26
2.2.2 **Papéis** ... 26
2.2.3 **Manipulação de Exceções** ... 27
2.3 TRABALHOS RELACIONADOS ... 27
2.3.1 **Construindo outros Mecanismos: CAA** ... 28
2.3.2 **Bomba de Injeção de Insulina Tolerante a Falhas** ... 29
2.3.3 **Célula de Produção** ... 31
2.3.4 **Célula de Produção Tolerante a Falhas** ... 32
2.3.5 **Desenvolvimento de Aplicações Web** ... 33

3 **ECLIPSE** ... 35
3.1 ARQUITETURA DA PLATAFORMA ... 35
3.1.1 **Plataforma de Execução** ... 36
3.1.2 **Workspaces** ... 36
3.1.3 **Workbench** ... 37
3.1.4 **Sistema de Ajuda e Suporte a Grupo** ... 38
3.1.5 **JDE e PDE** ... 39
3.2 PLUG-INS ... 39
3.2.1 **Plug-ins implementados no Eclipse** ... 41
3.2.2 **JIRiSS - Um plug-in Eclipse para exploração de código fonte** ... 42
3.3.3 **Sangam - Um plug-in Eclipse para Duplas de Programadores Distribuídos** ... 42
3.3.4 **KenyaEclipse: Aprendendo a Programar no Eclipse** ... 43
3.3.5 **Plug-in para Monitorar o Comportamento do Programador** ... 43
3.3.6 **SpeechClipse: Comandos de fala para Eclipse** ... 44
3.3.7 **Usando Eclipse para Ensinar Engenharia de Software a Distância** ... 44
3.3.8 **Jazzing: Eclipse como Ferramenta Colaborativa** ... 45
3.3.9 **DrJava: Uma Interface Amigável para Eclipse** ... 45

4 **DESCRIÇÃO DA PROPOSTA** ... 47
4.1 IMPLEMENTAÇÃO ... 48
4.2 USO DO PLUG-IN DMI ... 52
4.3 ESTUDO DE CASO: O JANTAR DOS FILÓSOFOS ... 59

5 CONCLUSÃO 61

REFERÊNCIAS 73

APÊNDICE A – Telas do Plug-in para geração do Estudo de Caso 63
APÊNDICE B – Código gerado pelo Plug-in 69

1 Introdução

Com a expansão das redes de computadores, atividades envolvendo comunicação de computadores estão tornando-se cada vez mais distribuídas. Tal distribuição pode incluir processamento, controle, gerenciamento de rede e segurança. Embora a distribuição possa melhorar a confiabilidade de um sistema por replicar componentes, as vezes um aumento na distribuição pode introduzir falhas.

Em diversos ambientes a introdução de falhas ou e existência de falhas é inaceitável. Por exemplo, sistemas onde a vida de pessoas podem estar em jogo (exemplo, Therac 25), ou mesmo onde recursos financeiros muito elevados foram aplicados (exemplo, Ariane 5). Estes sistemas são em geral chamados sistemas confiáveis. Sistemas confiáveis são sistemas que mantêm seu funcionamento de acordo com sua especificação mesmo na presença de situações excepcionais [1].

O comportamento deste sistema, mediante a ocorrência de situações inesperadas, é chamado de comportamento excepcional, ou comportamento anômalo, que define a forma como o sistema irá se comportar na tentativa de tratar situações excepcionais. Visando o desenvolvimento de sistemas mais robustos e confiáveis são aplicadas algumas técnicas complementares que auxiliam neste processo, tais como:

- Previsão de falhas: tem por objetivo verificar através da utilização de modelos matemáticos a possibilidade ou probabilidade da existência de falhas que ainda não se manifestaram no sistema.
- Prevenção de falhas: tem por objetivo prevenir a ocorrência ou introdução de falhas, através da utilização de técnicas de programação e por implementações baseadas em linguagens de alto nível.
- Remoção de Falhas: tem por objetivo minimizar o número ou a severidade das falhas, através de diversas técnicas de teste.

Em algumas situações é praticamente impossível evitar uma falha, como por exemplo, falhas em componentes físicos (hardware) deteriorados com o passar do tempo. Desta forma, faz-se necessário o emprego de técnicas de Tolerância a Falhas, que visam manter o sistema em funcionamento mesmo na presença de falhas. Vale salientar que até mesmo para que tolerância a falhas possa ser empregada é importante que as falhas sejam antecipadas e suas conseqüências identificadas para que medidas apropriadas de tolerância a falhas possam ser empregadas para detectar sua ocorrência e manter o correto funcionamento do sistema.

Sistemas tolerantes a falhas diferem em relação ao seu modo de operação na presença de falhas e principalmente em relação aos tipos de falhas que devem ser toleradas. Em alguns casos, o objetivo é continuar a proporcionar o desempenho e a capacidade funcional total do sistema, em outros o desempenho degradado e a capacidade funcional reduzida são aceitáveis, até a remoção da falha.

Os métodos de tolerância a falhas são realizados basicamente através de técnicas de tratamentos de erros e de falhas. As técnicas de tratamento de erros destinam-se a eliminá-los se possível antes que ocorra um defeito. O tratamento de falhas destina-se a evitar que falhas sejam reativadas. De forma geral, procedimentos de manutenção ou reparo estão associados ao tratamento de falhas.

Para reduzir o risco de introdução de falhas em sistemas distribuídos é importante que tais sistemas sejam implementados de maneira organizada. Uma maneira de organizar por exemplo, aplicações orientadas a objetos distribuídos é modelar operações que envolvem mais que um objeto como ações separadas que coordenam as interações necessárias entre os objetos participantes, facilitando a programação e utilização destes objetos.

Um mecanismo que inclui diversos participantes (objetos ou processos) executando um conjunto de atividades paralelamente é chamado de interação multiparticipante [2]. Em uma interação multiparticipante diversos participantes de alguma maneira "se unem"para produzir um estado combinado, intermediário e temporário e utilizam este estado para executar alguma atividade. Após a atividade executada a interação é desfeita e então cada participante prossegue a execução normal.

Diversas vezes a interação entre os participantes pode levar a situações onde toda a execução tem que ser refeita, gerando o efeito dominó [3]. Para evitar este tipo de situação e para auxiliar no tratamento de exceções concorrentes que podem ocorrer nos diversos participantes de uma interação pode-se utilizar, por exemplo, o mecanismo de interações multiparticipantes confiáveis (*Dependable Multiparty Interactions - DMIs*) . Este mecanismo tem sido utilizado para o desenvolvimento de diversos estudos de caso [4] [5] [6] [7] [8].

Entretanto, todos os estudos de casos desenvolvidos utilizando DMIs foram implementados de maneira manual, ou seja, sem a utilização de ferramenta de apoio. Tal situação além de acarretar um elevado tempo de desenvolvimento também facilita à inclusão de falhas no sistema. Este problema poderia ser reduzido com a utilização de algum ambiente para geração automática de código.

Atualmente existem diversos ambientes para automatizar o processo de desenvolvimento de software. Um exemplo que pode ser mencionado é o Eclipse [9]. O Eclipse é um projeto de código aberto que fornece muitos dos principais benefícios de ambientes de desenvolvimento comerciais, tais como JCreator [10] ou JetBrains [11]. Além de ser um ambiente disponível para toda a comunidade de maneira aberta, o Eclipse também fornece a possibilidade de extensibilidade através do desenvolvimento ou customização de *plug-ins*. Atualmente existe uma enorme variedade de *plug-ins* (gratuitos e comerciais) desenvolvidos para atender a diversos

fins do conhecimento humano. Percebemos que existe uma enorme carência de ferramentas (*plug-ins*) relacionados a atividades de desenvolvimento de aplicações tolerantes a falhas.

Esta dissertação apresenta uma forma de facilitar o desenvolvimento de aplicações tolerante a falhas que utilizem o mecanismo DMI. A proposta se baseia na possibilidade de utilizar ambientes de desenvolvimento de software e incluir nos mesmos a possibilidade de utilização de mecanismos que tratam de falhas. Como forma de demonstrar esta possiblidade, apresentamos um *plug-in* desenvolvido sobre o ambiente PDE (*Plug-in Development Environment*) do Eclipse que tem por objetivo automatizar o processo de criação de aplicações que utilizam o mecanismo de interações multiparticipantes confiáveis para tolerar falhas.

A utilização deste *plug-in* acarreta uma série de benefícios aos desenvolvedores. Podemos destacar a possibilidade de ganho de produtividade, visto que parte do código para criação de DMIs passa a ser automatizado; a redução da possibilidade de inserção de falhas no código responsável pela criação de DMIs, visto que código é gerado automaticamente; compreensão dos elementos que compõem uma DMI e a maneira como os mesmos estão relacionados.

Essa dissertação está organizada da seguinte forma. O Capítulo 2 apresenta o conceito de Interações Multiparticipantes Confiáveis (DMI). São descritos todos os componentes que compõem uma DMI, assim como a descrição de um *framework* para implementação de aplicações que utilizam DMIs. Por fim, são apresentados alguns trabalhos implementados utilizando DMIs para tolerar falhas. O Capítulo 3 apresenta os principais componentes que compõem o Eclipse (arquitetura); detalha os ambientes da plataforma; apresenta o conceito de *plug-ins* no Eclipse; e alguns trabalhos (*plug-ins*) desenvolvidos no PDE do Eclipse. O Capítulo 4 apresenta a descrição da implementação do *plug-in* desenvolvido e um estudo de caso comprovando os benefícios da utilização desta ferramenta. O Capítulo 5 traz conclusões do trabalho e aponta para possíveis trabalhos futuros.

2 Interações Multiparticipantes Confiáveis

Existe uma enorme variedade de linguagens de programação disponíveis para projetar e implementar sistemas computacionais. Estes sistemas podem conter diversas interações multiparticipantes durante sua execução, e usualmente tais interações são dispersas no código do sistema. Neste capítulo apresentaremos uma forma de implementar DMIs utilzando linguagens orientadas a objetos.

Conforme mencionado no Capítulo 1, um mecanismo composto por múltiplos participantes (objetos ou processos) e que executam atividades em paralelo é chamado de interação multiparticipante [2]. Em interações multiparticipantes, diversos participantes de alguma maneira "se unem" para produzir um estado combinado, intermediário e temporário, e utilizam este estado para executar alguma atividade. Após a atividade executada a interação é desfeita e então cada participante prossegue sua execução individual.

Interações multiparticipantes que tratam de problemas que podem ocorrer durante sua execução devem prover algumas propriedades básicas. As principais propriedades para a interação multiparticipantes desta categoria são [12]:

- sincronização dos participantes da interação[1];
- utilização de uma condição para verificar as pré-condições para executar a interação;
- confirmação após finalizar a interação, para garantir que um conjunto de pós-condições foi satisfeito pela execução da interação;
- atomicidade dos dados externos para assegurar que os resultados intermediários não sejam passados para outros processos antes que a interação acabe.

Geralmente, há também uma discussão sobre o uso do estado temporário pelos participantes da interação, a maneira como a interação é dividida (geralmente a interação é dividida em mais de uma parte, e cada uma destas partes é chamada de uma parte da interação), ou a forma como o número de participantes é especificado na interação.

Geralmente não são esperadas falhas durante a execução de um programa, porém as mesmas existem e são rotineiramente tratadas pelos programadores como exceções. Para assegurar o tratamento de exceções que podem ocorrer durante a execução de um programa, um mecanismo de tratamento de exceções é usualmente desenvolvido. Este mecanismo permite que

[1]Poder-se-ia utilizar uma estratégia mais otimista, liberando os participantes para iniciar a interação mesmo que diversos não tenham chegado, mas esta situação poderia trazer um custo muito alto para manter o estado de todos os possíveis participantes para uma eventual recuperação.

um programador descreva um fluxo excepcional que substitua o fluxo normal de um programa sempre que uma exceção for detectada nesse programa.

O termo Interação Multiparticipante Confiável (*Dependable Multiparty Interaction*) é utilizado para Interações Multiparticipantes que provêm facilidades para a manipulação de exceções, em particular incluindo meios de:

- **Manipulação de Exceções Concorrentes:** quando uma exceção ocorre em um dos participantes, e se não for tratada por esse participante, a exceção deve ser propagada a todos os participantes [13] [14]. Uma DMI também deve prover uma maneira de tratar as exceções que podem ser geradas por um ou mais participantes. Finalmente, se diversos tipos de exceções são gerados simultaneamente, então o mecanismo DMI utiliza um processo de resolução de exceção para decidir qual exceção será repassada para todos os participantes. Com respeito a como os participantes de uma DMI serão envolvidos na resolução e manipulação de exceções, foram propostos dois possíveis esquemas [15] [16]: síncrono ou assíncrono. No esquema síncrono cada participante deve ter terminado sua atividade e estar pronto para tratar a excessão. No esquema assíncrono os participantes não esperam até finalizarem suas execuções ou gerarem uma exceção para participar da manipulação de exceção, uma vez que uma exceção é gerada em vários participantes do DMI, todos os outros participantes são interrompidos e tratam as exceções geradas juntos. Embora a implementação de esquemas síncronos ser mais fácil que a implementação de esquemas assíncronos, uma vez que todos os participantes estão prontos para executar a manipulação de exceções, o esquema síncrono pode trazer o indesejável risco de *deadlock*.

- **Sincronização na Saída**: todos os participantes têm que esperar até que as interações acabem completamente. Um participante somente pode deixar a interação quando todos terminaram seus papéis e os objetos externos estiverem em um estado consistente. Esta propriedade garante que se aconteceu algo de errado na atividade executada por um dos participantes, então todos os participantes podem tentar recuperar os erros possíveis[2].

A idéia principal para manipulação de exceções é construir uma DMI a partir de uma corrente de Interações Multiparticipantes onde cada elo é o tratador de exceções do elo anterior. A Figura 1 mostra como uma interação multiparticipante básica e interações multiparticipantes que tratam exceções (*exception handling*) são encadeadas para formar uma interação multiparticipante composta, de fato o que denomina-se uma DMI. Como mostrado na figura, a interação multiparticipante pode acabar normalmente; gerando exceções que são tratadas por interações multiparticipantes; ou gerando exceções que não são tratadas na DMI. Se a interação multiparticipante acabar normalmente, o fluxo de controle é passado para os chamadores da DMI. Se uma exceção é gerada, então há dois caminhos possíveis de execução a serem seguidos:

[2] Situação similar ao que foi descrito para a sincronização na entrada.

- se houver uma manipulação da exceção da interação multiparticipante para tratar esta exceção, então é ativado para todos os papéis da DMI;
- se não houver tratamento de exceção, então esta exceção é sinalizada para os invocadores da DMI.

O conjunto composto pela interação multiparticipante básica, e pelos tratadores de exceções da interação multiparticipante são representados como uma entidade composta de interações multiparticipantes. As exceções que são geradas pela interação multiparticipante básica ou pelo tratador, devem ser as mesmas para todos os participantes da DMI. Se diversos participantes gerarem diferentes exceções concorrentes, o mecanismo DMI ativa um algoritmo de resolução de exceções baseado em [13] para decidir qual exceção comum deve ser tratada.

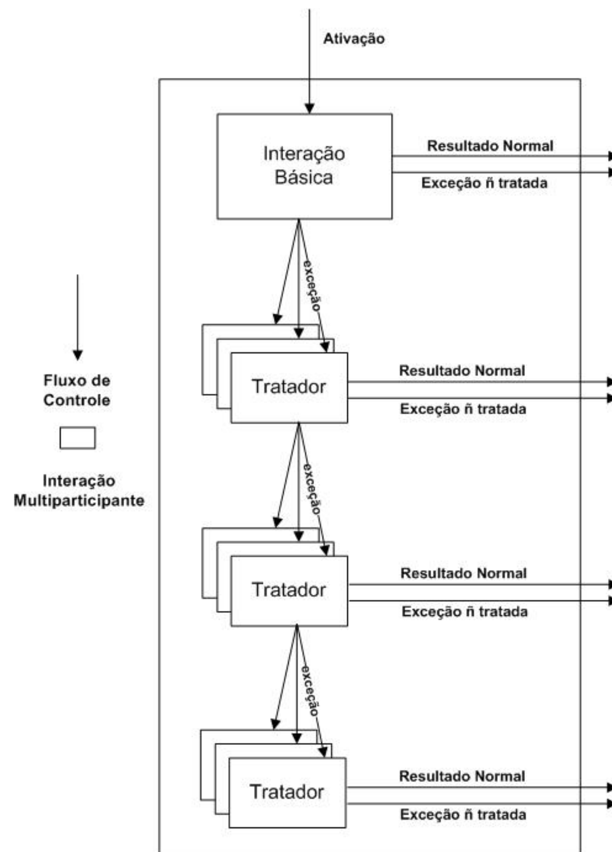

Figura 1 – Interação Multiparticipante Confiável.

2.1 Implementação de DMI.

DMIs podem ser implementadas de diversas maneiras. Nesta seção nós discutimos uma alternativa para fornecer uma implementação distribuída de DMIs em linguagens orientadas a

objetos [17]. Primeiramente, os diversos componentes que podem ser usados para implementar uma DMI são descritos. Basicamente, uma DMI é composta por:

- Gerente (*Manager*) : um componente (ou mais) utilizado para controlar todos os protocolos dentro da DMI, tais como: pré e pós-sincronização de participantes que participam da DMI; manipulação de exceções entre os processos; manutenção do controle interno e externo de dados para a DMI; etc.
- Papéis (*roles*): diversos segmentos de códigos de aplicação, cada um deles é executado por um participante da DMI;
- Dados externos: dados que são externos à DMI. Este tipo de dado pode ser acessado de maneira competitiva por alguém que não está participando da DMI. Um controle de acesso especial é requerido (em geral um sistema de transações);
- Dados locais: dados que são locais à DMI. Participantes que não estão executando um papel (*role*) na DMI não têm acesso a estes dados. Estes objetos, em geral, são utilizados pelos participantes para troca de informação ou para sincronização.

Em linguagens orientadas a objetos, diversas possibilidades de implementar gerentes e papéis podem ser projetadas, por exemplo, papéis como objetos separados ou como funções de um objeto gerente. Dados locais e dados externos são representados como objetos. Uma maior descrição pode ser visto em [17].

2.2 *Framework* para implementação de DMIs

No início do Capítulo 2 mostramos como componentes poderiam ser lincados para implementar DMIs. Nesta seção um *framework* genérico orientado a objetos para implementar DMIs é descrito. O *framework* proposto é composto por quatro tipos de objetos (como descrito na Seção 2.1): papéis, gerentes, objetos compartilhados e objetos externos. Cada um dos objetos descritos pode ser distribuído em diferentes locais. Cada DMI é representado por vários conjuntos destes objetos remotos: um conjunto para interações quando não houver nenhuma falha, *i.e.* interação básica, e vários conjuntos para tratar as exceções que possam ser geradas durante a execução da interação (durante a interação básica ou durante uma interação da manipulação de exceção).

A Figura 2 refere-se a uma versão reduzida de um diagrama de classes UML (*Unified Modelling Language*) que representa o *framework*. A figura mostra que cada papel está associado a somente um gerente, e que cada gerente controla apenas um papel. Os gerentes são associados a um gerente especial, chamado de gerente líder. O gerente líder é o único que é associado com objetos locais compartilhados. Objetos locais compartilhados são criados pelos papéis,

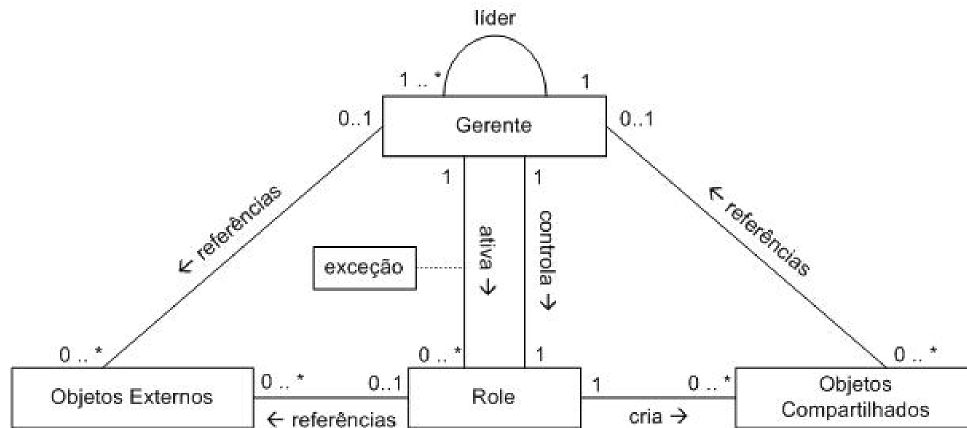

Figura 2 – Representação UML do *framework*.

que exportam eles para o gerente líder e desta forma tornam-se acessíveis para outros papéis. Objetos externos são associados com ambos gerentes e papéis. Os gerentes manterão estes objetos para possível processo de recuperação. Como mostrado na figura, não há nenhuma classe para representar uma interação multiparticipante básica ou uma DMI. Uma DMI pode ser construída usando uma seqüência de passos de programação para relacionar os componentes quando forem instanciados (criados). Este processo conforme já mencionado pode inserir falhas no desenvolvimento do sistema. Este é o aspecto que resolvemos neste trabalho conforme será descrito no Capítulo 4.

Para programar uma nova DMI, utilizando o *framework*, o primeiro passo é definir uma nova classe que extenda a classe `Role` para cada parte da interação. A classe estendida de `Role` deve redefinir ao menos um método: o método *body*. Este método conterá um conjunto de operações que será executado pelo participante ativo do papel. No momento da criação de cada papel deve ser informado o gerente que estará gerenciando este papel. Um gerente que controla um objeto `Role` é uma instância de uma classe Manager. A classe Manager fornece uma base para coordenar os participantes em uma interação multiparticipante. A separação do gerente dos papéis permite que o código da aplicação do papel possa ser distribuído para um local diferente do gerente. Esta estratégia ajudará a evitar sobrecarga de um local com o controle da DMI e o código da aplicação.

Os gerentes de todos os papéis irão compor o controle da interação. Cada gerente no momento da criação é informado de qual gerente vai atuar como líder na interação. O líder é responsável por controlar protocolos de sincronização entre gerentes, para o algoritmo de resolução de exceção, e para manter-se informado sobre os objetos locais compartilhados. Todos os gerentes são um potencial líder no *framework*, evitando a possibilidade de ponto único de falha.

2.2.1 Gerentes

A classe `Manager` é a principal classe do *framework*. Cada papel tem que ser controlado por um gerente diferente. Quando instanciar um objeto `Manager`, o gerente tem que ser informado de seu nome, o nome da interação, o líder da interação (um gerente sem um líder é seu próprio líder) e uma lista de exceções que será tratada pelo gerente. Cada exceção na lista é associada com um papel para manipular exceções de interações. A lista de exceções anexada para os gerentes é a ligação entre as interações multiparticipantes de uma DMI. Exceções que são geradas em interações multiparticipantes, e que não são tratadas, são propagadas.

Existem 3 maneiras para criar um objeto `Manager`: criando um objeto `Manager` líder que trata exceções; criando um objeto `Manager` que trata exceções e é controlado por um líder; ou, criando um objeto `Manager` que não suporta exceções e é controlado por um líder.

A Figura 3 mostra dois gerentes que foram criados para uma mesma DMI (com `mgr1` atuando como líder de uma DMI e `mgr2` sendo liderado por `mgr1`). A tabela `eh1` contém a lista de exceções que são tratadas pelo `mgr1` e os papéis que são ativados no caso de uma das exceções contidas na lista ser levantada. Todas tabelas com exceções devem conter a mesma lista de exceções a serem tratadas em uma mesma DMI. Se as tabelas não contiverem a mesma lista de exceções, então uma exceção é levantada em tempo de criação.

```
Manager mgr1=new Manager ("mgr1", "DMIname", eh1, null)
Manager mgr2=new Manager ("mgr2", "DMIname", eh2, null)
```

Figura 3 – Gerentes Instanciados para a mesma DMI.

2.2.2 Papéis

Depois que um novo objeto `Manager` foi criado, o programador da interação multiparticipante tem que criar um objeto `Role` que será controlado por este gerente. Este objeto `Role` tem que ser uma instância de uma nova classe `Role` derivada de uma classe `Role` fornecida pelo *framework*. Cada nova classe derivada de `Role` contém o código principal para um dos papéis que compõem a interação multiparticipante. Somente objetos cujos tipos derivam do objeto `Role` podem pertencer a uma interação multiparticipante. Quando derivar uma nova classe para a classe `Role` o programador deve implementar ao menos um método: o método `body`, o qual conterá o código da aplicação principal para este papel. Este método não retorna nenhum valor. Ele recebe uma lista de objetos externos como parâmetro. Se uma exceção é originada dentro deste papel, então esta exceção pode ser tratada localmente pelo papel, se esta exceção não afetar outros papéis. Se a exceção gerada tiver algum efeito nos outros papéis, deve ser repassada para o gerente deste papel, que notificará o líder e irá interromper todos os

papéis da DMI. Depois que todos os papéis forem interrompidos o líder resolve qual exceção será tratada por todos.

Extensões de classe `Role` são também responsáveis por declarar objetos locais compartilhados usados para coordenar os papéis dentro de uma interação particular, e para verificar partes das pré e pós-condições da interação. Depois que os objetos locais compartilhados foram criados, o papel deve informar seu gerente sobre estes objetos, usando o método `sharedObject`. Este método publica os objetos locais compartilhados para que outros papéis nesta interação possam usá-los mais tarde.

As pré e pós-condições de uma interação também podem ser verificadas de uma maneira distribuída; cada papel verifica parte das condições, ou um papel pode ser delegado para verificar todas as pré e pós-condições da interação. A delegação pode ser conseguida utilizando objetos locais compartilhados entre os papéis. Os métodos em que os testes de pré e pós-condições são programados são chamados: `preCondition` e `postCondition`. Este métodos podem ser refinados na nova classe `Role`.

2.2.3 Manipulação de Exceções

Por padrão, a classe `Manager` fornece um mecanismo de resolução de exceções concorrentes interno, baseado em [14]. Este mecanismo trabalha da seguinte forma. Quando um papel gera uma exceção, seu gerente correspondente é notificado desta exceção. O gerente então informa o líder que interrompe todos os papéis que não geraram exceções. Depois que todos os papéis foram interrompidos ou notificaram o gerente do líder sobre uma exceção (exceções podem ser geradas concorrentemente), um algoritmo de resolução de exceções é executado pelo líder. Este algoritmo tenta encontrar uma exceção comum para todas as exceções levantadas. Quando uma exceção comum é encontrada, o líder informa todos os gerentes sobre a exceção e um manipulador de exceções é ativado. Se não houver nenhum tratador para esta exceção, um tratador para exceções de mais alto nível é ativado, i.e. classe `Exception`. Se não houver nenhum tratador para a exceção, então a exceção é sinalizada para quem ativou a DMI.

2.3 Trabalhos Relacionados

Nesta seção são apresentados alguns trabalhos que foram implementados utilizando o mecanismo de Interação Multiparticipante Confiável para tolerar falhas. Conforme poderá ser observado, DMIs podem ser utilizadas para tolerar falhas em uma grande variedade de sistemas, desde sistemas médicos até sistemas manufaturados. A primeira seção mostra como DMIs podem ser utilizadas para desenvolver outro mecanismo, ações atômicas coordenadas (CAAs), que possuem requisitos mais restritivos que DMIs. Este mecanismo (CAA) foi utilizado para o

desenvolvimento das aplicações citadas neste capítulo.

2.3.1 Construindo outros mecanismos: CAA

DMIs foram desenvolvidas para serem aplicadas no desenvolvimento de sistemas confiáveis. Entretando, em determinadas situações é necessário a adoção de formas mais restritas para a manipulação de exceções. Ações Atômicas Coordenadas (*Coordinated Atomic Actions - CAA*) [15] são um exemplo destas abstrações.

Assim como DMIs, CAA é um mecanismo utilizado para coordenar interações multiparticipantes e garantir acesso consistente a objetos na presença de concorrência e falhas potenciais. Pode ser considerado como uma disciplina de programação que proporciona facilidades para transações concorrentes e tratamento de exceções.

CAA envolvendo múltiplos papéis em uma ação devem concordar sobre seu resultado que podem ser um de quatro possíveis: normal, exceção, abortar e falhar. Uma CAA termina normalmente se for capaz de satisfazer suas pós-condições. Se uma CAA não terminar normalmente, então cada papel deve sinalizar uma exceção para indicar o resultado. Os papéis devem concordar sobre o resultado de forma que cada papel deve sinalizar a mesma exceção. Se uma exceção é gerada durante a execução da CAA, é desencadeado um processo de tratamento de exceção. Dependendo do sucesso, a CAA pode recuperar a partir da exceção, pode terminar normalmente ou excepcionalmente. Se a recuperação de erro não for possivel, a CAA pode tentar recuperar o estado dos objetos externos e sinalizar abortar. Se a recuperação do estado for mal sucedida, então a CAA deve sinalizar um defeito. Este aspecto diferencia CAA de DMIs, ou seja, em DMIs seria possível continuar tratando exceções, enquanto que em CAA o processo de tratamento de exceções deve terminar (uma cadeia de exceções não é possível em CAA).

Assim como em DMIs, o aninhamento de CAA é permitido. Ou seja, uma CAA pode ser composta por diversas CAA.

A Figura 4 mostra como o mecanismo de CAA é construído usando o mecanismo de interações multiparticipantes. Na figura, as caixas retangulares representam as interações multiparticipantes, as setas representam o fluxo de controle e a caixa maior representa o mecanismo CAA. Como pode ser observado, CAA são implementados com três tipos de interações multiparticipantes: uma interação para a execução normal da CAA; um conjunto de tratadores de interações multiparticipantes para lidar com as exceções que podem ser geradas durante a execução normal da CAA; e a interação que lida com o processo de recuperação da CAA.

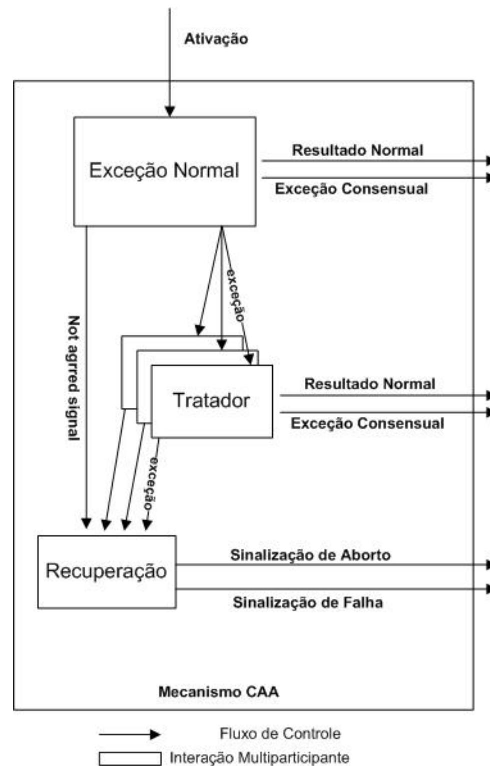

Figura 4 – Representação de ações atômicas coordenadas.

2.3.2 Bomba de Injeção de Insulina Tolerante a Falhas

Neste trabalho [7] é descrito a implementação de um sistema de controle médico que requer um alto grau de confiabilidade. O sistema é desenvolvido utilizando Ações Atômicas Coordenadas (CAA) implementadas com DMIs (conforme descrito na Seção 2.3.1). O sistema baseia-se nas técnicas de injeção subcutânea de insulina envolvendo diferentes sensores afim de garantir a execução contínua do tratamento, assim como detectar defeitos.

O grande desafio a ser transposto foi desenvolver um sistema que mantém a entrega de insulina mesmo na presença de um grande número e variedade de falhas de *hardware* e *software*. A implementação deste tipo de controle foi desenvolvida em Java utilizando uma extensão do *framework* para DMIs.

O objetivo deste trabalho é demonstrar como CAA podem ser utilizados para o desenvolvimento de sistemas médicos que requerem resiliência e alta diponibilidade. O estudo de caso em questão demonstra como um sistema de controle para diabetes fornece insulina ao paciente. A equipe médica define os parâmetros do tratamento e o sistema, sensores e bombas verificam o *status* do paciente e administram a insulina. É de extrema importância para o bem estar do paciente que a aplicação opere 24 horas por dia sem nenhum tipo de interrupção.

O sistema de controle de diabetes faz uso de diferentes tipos de dispositivos, que combinam alta performance, baixo consumo de energia e comunicação sem fio, aumentando a inteligência

dos sensores e atuadores.

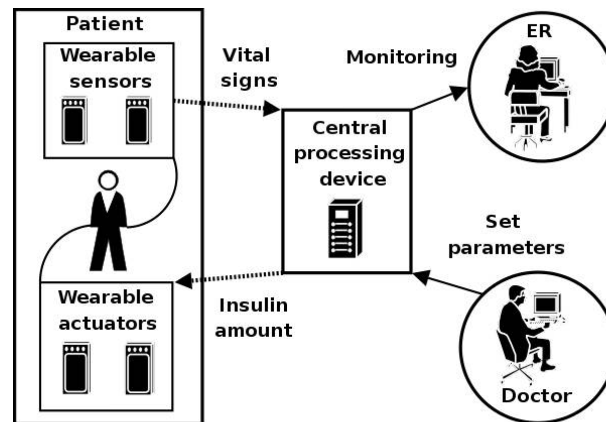

Figura 5 – Os componentes e seus relacionamentos.

A Figura 5 [7] mostra os diferentes componentes presentes no cenário. O principal componente deste cenário é o paciente que está recebendo o tratamento, e onde os dispositivos estão conectados (sensores e atuadores). O médico deve definir os parâmetros do tratamento enquanto os dispositivos operam de acordo com o tratamento especificado que o paciente deve receber. Esta informação será armazenada nos registros pessoais do paciente e serão consultadas pela aplicação. Entretando, as facilidades para o médico alterar e consultar as informações sobre o tratamento devem ser projetadas para ser tolerante a falhas.

O último componente é a sala de emergência (ER), onde encarregados estão continuamente monitorando os sinais vitais dos pacientes. Eles serão os primeiros a saber se existe algum problema com o tratamento que o paciente está recebendo. As setas pontilhadas representam a conexão sem fio e mostram como os dispositivos são conectados com o dispositivo de processamento central.

O sistema de controle da bomba de insulina foi projetado com a utilização de 7 CAAs. Estas CAAs foram projetadas utilizando aninhamento e composição. Existe uma CAA que controla todo o sistema chamada `CAA-Cycle`. Esta CAA é responsável por fazer a verificação dos níveis de insulina do paciente e dos sensores da bomba de insulina. Esta atividade é executada pela CAA aninhada `CAA-checking`. A `CAA-checking` por sua vez ativa uma CAA composta (`CAA-sensors`) que é responsável pela verificação dos sensores da bomba de insulina. A `CAA-Cycle` também é reponsável pela administração de insulina ao paciente. Esta atividade e de responsabilidade de outra CAA aninhada chamada `CAA-executing`. A `CAA-Executing` por outro lado ativa uma CAA composta (`CAA-actuators`) que é responsável pelo controle dos atuadores da bomba. A Figura 6 [7] demonstra todo este processo.

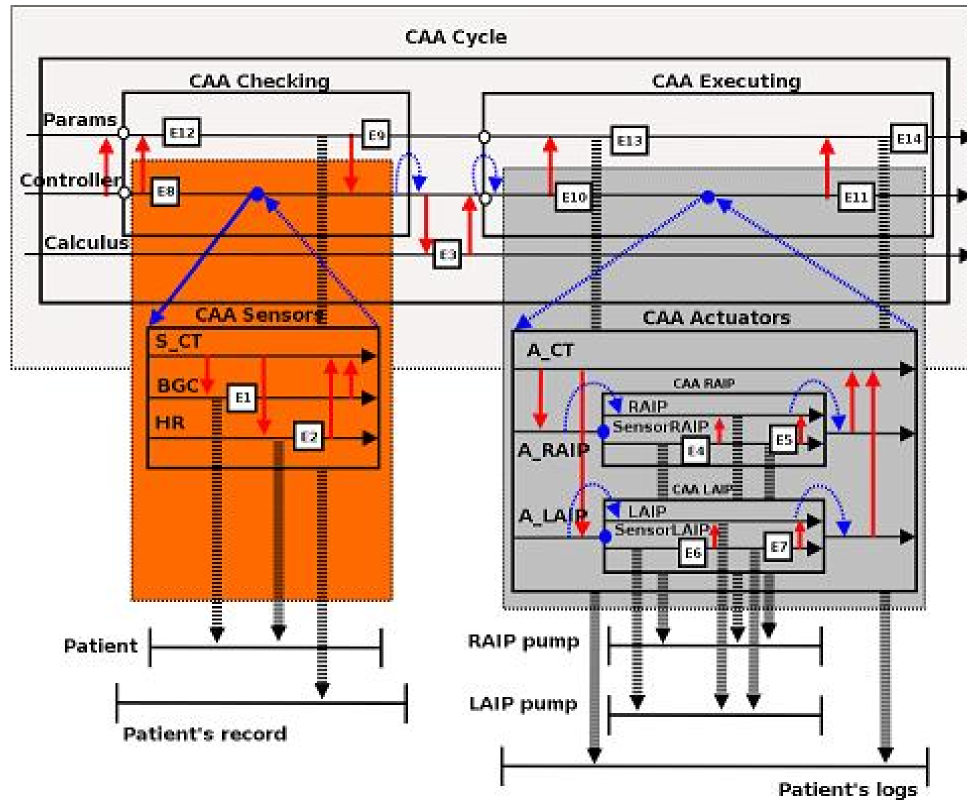

Figura 6 – Estruturação das CAAs para o sistema de tratamento de insulina.

2.3.3 Célula de Produção

Esta célula de produção é baseada em uma célula real para processamento de metais e foi inicialmente criado pelo FZI in 1993 [18] e o objetivo deste modelo era avaliar e comparar diferentes métodos formais e explorar a praticabilidade (possibilidade de execução) destes modelos para aplicações industriais. Desde então, o estudo de caso em questão atraíu muito a atenção e já foi investigado por mais de 35 distintos grupos de pesquisadores.

O trabalho apresentado nesta seção, descreve um sistema para controle desta célula de produção. Neste sistema é aplicado o conceito de CAA para desenvolver e implementar um sistema que tolere falhas [4]. O sistema é desenvolvido em dois níveis: o primeiro nível incide na negociação com a sincronização de CAA, e o segundo nível lida com a interação entre os dispositivos. Tanto a sincronização de CAA aninhadas como as interações de dispositivos são realizados dentro de CAA. Tratamento de exceções e recuperação de erros são incorporados dentro de CAA afim de satisfazer os requisitos de tolerância a falhas. O sistema de controle foi desenvolvido na linguagem Java e é utilizado para controlar um simulador gráfico.

O projeto desta Célula de Produção aborda requisitos de segurança, funcionalidade e desempenho. O requisito de segurança é satisfeito com a adoção de CAA, enquanto os demais requisitos são garantidos dentro das CAA através da programação de acesso aos dispositivos e sensores. Esta programação pode ser realizada de diversas maneiras.

A Figura 7 mostra a CAA mais externa (`Production Cell CA action`) e uma possível execução de diversas CAA aninhadas (`UnloadTable, LoadTable, LoadPress, UnloadPress, ForgePlate, TransportPlate, UnloadDepositBelt`). Os diversos participantes (`FeedBelt, Table, Robot, Press, DepositBelt, Crane`) executam os papéis na CAA mais externa e depois ficam no interior desta executando as CAA aninhadas dependendo das pré-condições de cada CAA aninhada. Por exemplo, para descarregar a mesa (`UnloadTable`) é necessário que exista uma peça de metal sobre a mesma e que o robô esteja livre. Estas condições são informadas pelos participantes via objetos compartilhados.

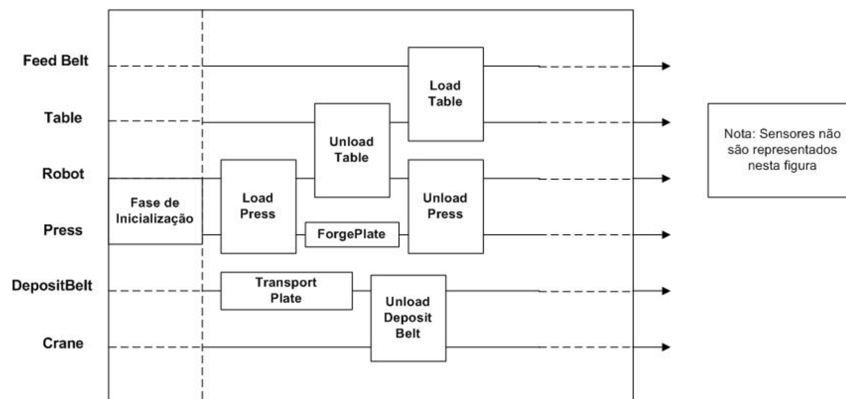

Figura 7 – Possível execução de CAA aninhadas no sistema de controle da Célula de Produção.

2.3.4 Célula de Produção Tolerante a Falhas

Em 1996, o FZI apresentou a especificação de uma versão extendida da célula de produção original, denominada "Célula de Produção Tolerante a Falhas ou Célula de Produção II" [19]. Este segundo modelo possui uma prensa adicional, sensores extras, e luzes nos sistemas de alerta que facilitam a detecção e tolerância a falhas. Este sistema é muito mais complexo e realistíco que o primeiro modelo de Célula de Produção [18]. Diferente do primeiro modelo, falhas de componentes eletromecânicos e sensores em produção são a grande preocupação. Em particular, a célula de produção tem por objetivo fornecer serviço continuamente mesmo que uma das suas duas prensas esteja fora de ordem.

O trabalho é baseado em uma extensão do modelo de célula de produção, a especificação e simulação para o qual foram definidos pelo FZI (*Forschungszentrum Informatik, Germany*). Esta célula de produção tolerante a falhas representa o processo de fabricação envolvendo redundância de dispositivos mecânicos (garantindo a produção contínua mesmo na presença de falhas mecânicas). O desafio proposto pelo modelo especificado é desenvolver um sistema de controle capaz de manter seu funcionamento de acordo com sua especificação mesmo na pre-

sença de um grande número e variedade de falhas (dispositivos, sensores...).

No modelo original, a Célula de Produção não dispunha de qualquer dispositivo ou sensor para verificar a ocorrência de defeitos. Em tal hipótese foi utilizado o conceito de CAA para organizar e projetar um programa de controle, e posteriormente implementá-lo em Java. O programa de controle desenvolvido foi então aplicado para o simulador. Para a Célula de Produção Tolerante a Falhas a mesma estratégia foi aplicada, mas agora buscando satisfazer os requisitos de tolerância a falhas extendido [20].

Assim como no sistema para a primeira Célula de Produção, CAA são aplicadas sempre que ocorre alguma interação entre os os dispositivos. Por exemplo, quando uma peça vai passar da mesa para o robô os controladores do robô e da mesa executam uma CAA chamada *UnloadTable*. Esta mesma estratégia é utilizada quando a peça é colocada ou removida das prensas, colocada na esteira de alimentação, e assim por diante. A Figura 8 [20] apresenta o conjunto de dispositivos e também o conjunto de CAA utilizadas para controlar a célula. Sempre que dois retângulos, que representam CAA, intersectam, significa que duas CAA não irão executar paralelamente pois o participante de uma CAA também é participante de outra.

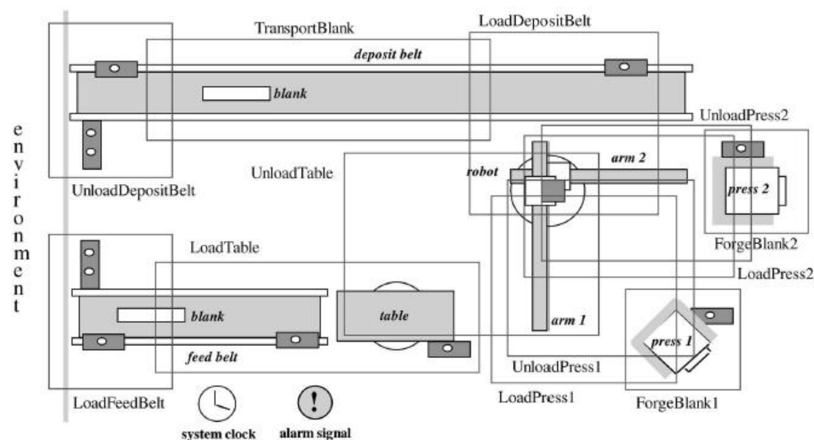

Figura 8 – CAA que controlam a Célula de Produção Tolerante a Falhas.

2.3.5 Desenvolvimento de Aplicações Web

Este trabalho [6] discute algumas das características típicas das aplicações Web modernas e analisa alguns problemas que os desenvolvedores enfrentam na concepção deste tipo de sistema. Para esta aplicação Web diversos serviços Web independentes são integrados. A integração destes serviços independentes pode envolver interações que podem ser bastante complexas em atividades concorrentes.

O conceito de CAA é utilizado para estruturar estas atividades e para fornecer tolerância a falhas utilizando tratamento de exceções. O trabalho detalha importantes decisões de projeto, implementações adotadas no desenvolvimento do estudo de caso referente a uma Agência de

Viagens e o esforço para permitir que CAA fossem facilmente aplicadas para a construção de aplicações Web seguras que integram diversos serviços Web que um usuário pode desejar acessar.

O sistema é composto por duas partes: um lado servidor e um lado cliente. O lado do cliente é executado na máquina do usuário e coleta as informações fornecidas pelo usuário. Estas informações são enviadas para o sevidor que é responsável pela ativação de sistemas legados. A execução de todo o sistema é estruturado através de CAA's, e todas as informações trocadas entre o cliente e o servidor são realizadas dentro de CAA através de objetos locais. Assim, toda vez que um cliente conecta ao sistema de agência de viagens uma CAA é iniciada no lado do servidor. Esta ação é chamada de sessão. A CAA sessão é composta por duas sessões aninhadas. Uma para verificar a disponibilidade dos serviços solicitados pelo cliente, e outra para fazer a reserva propriamente dito. Dentro da CAA para verificar a disponibilidade existe uma ação aninhada para consultar os servicos legados. Cada servico legado é verificado por uma CAA composta, por exemplo uma CAA que possui diversos papéis, um para cada companhia outro para verificar a disponibilidade de vôos.

3 Eclipse

O Eclipse [9] é composto por uma plataforma totalmente modular, baseada em *plug-ins*, que foi projetada para fornecer integração confiável e robusta no desenvolvimento de aplicações para diferentes sistemas operacionais. É atualmente um dos IDEs (*Integrated Development Environment*) mais polulares para desenvolvimento em plataforma Java e considerado uma das principais iniciativas *Open-Source*.

O Eclipse foi um projeto inicialmente desenvolvido pela IBM, que posteriormente foi doado para a Fundação Eclipse. O projeto ganhou popularidade e força rapidamente na comunidade de desenvolvedores Java por diversos motivos:

- Ser software gratuito, livre (*free software*) e de código aberto (*open source*).
- Oferecer amplos e inovadores recursos de produtividade (*plug-ins*).
- Consistir em um projeto sério e ativo, bem organizado e coordenado, além do amplo apoio da comunidade e de grandes empresas e instituições.
- Ter ambiente gráfico construído com a biblioteca de componentes SWT (*Standard Widget Toolkit*) própria do projeto Eclipse, combinando grande riqueza de componentes gráficos e interface de usuário com desempenho e leveza.
- Possuir uma arquitetura de software aberta e extensível, permitindo que *plug-ins* sejam criados e facilmente integrados ao mesmo.

Para pesquisadores e educadores o Eclipse oferece um valor significativo: uma infra-estrutura para condução de pesquisas e desenvolvimento curricular em muitas áreas da computação, com relevância particular em linguagem de programação, ferramentas de desenvolvimento e colaboração.

3.1 Arquitetura da Plataforma

Mais do que apenas uma IDE (*Integrated Development Environment*) [21] [22], o Eclipse é basicamente uma plataforma de integração de sistemas, visto que fornece potencialidades não só para desenvolver produtos, mas para criar ferramentas para construir produtos. Estas ferramentas (*plug-ins*) são pacotes estruturados de códigos que atribuem funções ao sistema,

estendendo as funcionalidades do Eclipse e ilustrando o maior diferencial desta plataforma, a possibilidade de expansão. A característica de expansão permite ao Eclipse integrar novas funcionalidades para diferentes fabricantes de *software*, enquanto também assegura um ambiente de coesão.

Alguns *plug-ins* adicionam características visíveis à plataforma utilizando o modelo de extensão [23], enquanto outros fornecem classes de bibliotecas que podem ser usadas para implementar extensões do sistema. A Figura 9 detalha a arquitetura básica da plataforma do Eclipse.

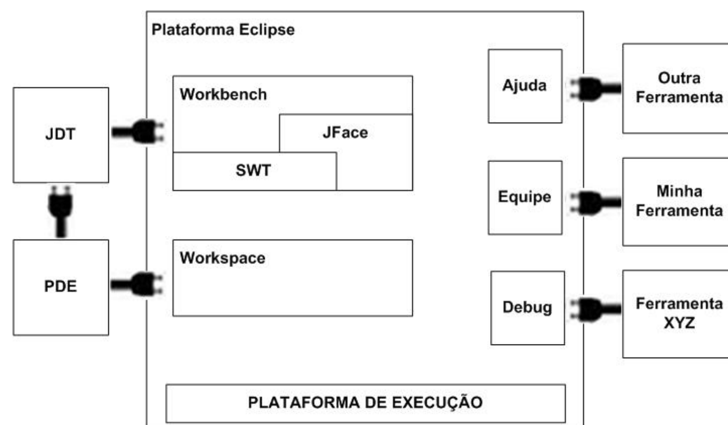

Figura 9 – Arquitetura da Plataforma Eclipse.

3.1.1 Platforma de Execução

A plataforma de execução é a única parte do Eclipse que não é um *plug-in*, e é responsável por carregar o sistema base e descobrir dinamicamente os *plug-ins* básicos necessários ao Eclipse. A plataforma de execução mantém um registro referente aos *plug-ins* instalados, assim como a função que eles fornecem.

Novas funcionalidades são adicionadas ao sistema através de um modelo comum de extensão, denominado pontos de extensão. Pontos de extensão são basicamente implementados em Java, através da utilização de APIs (*Application Programming Interface*) disponibilizadas pela plataforma, entretando, alguns pontos de extensão são implementados sobre componentes ActiveX [24], ou em linguagens baseadas em *script* [25].

3.1.2 Workspaces

A plataforma do Eclipse incorpora o conceito de *workspace*, que refere-se a um diretório mantido localmente na própria estação do desenvolvedor onde estão armazenadas informações referentes ao projeto em desenvolvimento, assim como também define uma API para criar e

gerenciar recursos, tais como identificar, notificar e manter o histórico de mudanças.

Este paradigma auxilia o processo de desenvolvimento visto que todas informações referentes a um determinado projeto estão contidas em um diretório comum, e sempre que são criados novos projetos as informações referentes a tais projetos são mapeadas para subdiretórios dentro do diretório *workspace*.

O uso do *workspace* local permite a colaboração muito mais eficiente de uma equipe através dos repositórios que podem ser disponibilizados na Internet ou em uma rede corporativa.

3.1.3 Workbench

O *workbench* é a interface do Eclipse com o usuário. É através dele que o desenvolvedor interage com o sistema, seja para compilar um projeto, para fazer o teste de um *plug-in* ou apenas para visualizar as propriedades de determinado arquivo. Ele é formado por uma série de componentes, sendo a maioria deles de interface gráfica do usuário (GUI) com ações associadas. Esses componentes se classificam basicamente em perspectivas, visões e editores.

- Perspectiva - Uma perspectiva controla a visibilidade inicial da visão, a visibilidade da ação e o leiaute inicial das visões e editores da janela do *workbench*. Um *workbench* pode ter uma ou mais perspectivas, porém apenas uma é apresentada ao usuário de cada vez. Cada perspectiva tem seu próprio conjunto de vistas, porém o conjunto de editores é compartilhado entre todas elas. As perspectivas controlam o que aparece em certos menus e barras de ferramentas, além de definirem os conjuntos de ações (*action sets*) visíveis, que podem ser alterados para personalizar a perspectiva. Uma perspectiva alterada pode ser armazenada para ser aberta posteriormente. Cada perspectiva oferece um conjunto de funcionalidades para realizar um tipo de tarefa ou para trabalhar com um tipo específico de recurso.

- Visões - Uma visão (*view*) é um componente de interface gráfica dentro do *workbench*. Ela é tipicamente utilizada para navegar em uma hierarquia de informações (como um sistema de arquivos), abrir um editor ou mostrar propriedades para um editor ativo. Modificações feitas em uma visão (como alteração de parâmetros) são salvas imediatamente. As visões também têm seus próprios menus e suas barras de ferramentas, que afetam apenas os ítens encontrados dentro da respectiva visão. Elas podem aparecer sozinhas ou agrupadas em forma de guias (*tabs*).

- Editores - Um editor também é um componente visual no *workbench*. Ele é utilizado para abrir, editar, salvar ou navegar através de recurso. Modificações feitas em um editor só são salvas quando o usuário decidir, diferentemente das vistas. Um editor pode estar associado com diferentes tipos de arquivos. Se não há um editor associado com o determinado tipo de arquivo dentro da plataforma, é possível abrir um editor externo, fora do

workbench. Vários recursos podem ser abertos para edição ao mesmo tempo, porém só um pode ser visualizado por vez. Guias na área do editor indicam os nomes dos recursos abertos. Um asterisco junto ao nome do recurso mostra que existem informações não salvas.

O *workbench* fornece uma completa estrutura que disponibiliza elementos necessários que constituem a interface entre o desenvolvedor e a plataforma. O *workbench* é a principal janela para todas as funcionalidades que os *plug-ins* podem fornecer. Através dele é possível navegar em projetos, pastas e arquivos, visualizar e editar o conteúdo e propriedades destes recursos, assim como adicionar pontos de extensão a interface.

Do ponto de vista do desenvolvedor, uma aplicação é composta por editores, visões, e perspectivas que podem ser customizadas, possibilitando assim uma total liberdade para alterar a interface conforme sua necessidade/vontade.

A API *workbench* é implementada usando ambos SWT [26] e JFACE [27]:

- SWT (*Standard Widget Toolkit*) [26] é uma biblioteca de componentes para interface gráfica que disponibiliza à plataforma Eclipse funcionalidades nativas para componentes gráficos de modo independente de sistema operacional. Provê um conjunto de componentes visuais, que engloba desde botões e listas, até componentes que adicionam ícones nas barras de tarefa do sistema operacional. É análogo ao AWT/Swing com uma diferença, o SWT usa um vasto conjunto de componentes nativos.

- JFACE é um conjunto de classes que serve como apoio para o desenvolvimento de aplicações SWT. Durante a fase inicial do processo de desenvolvimento do Eclipse e do desenvolvimento do SWT, os desenvolvedores da plataforma perceberam que poderiam facilitar o uso do SWT com algumas novas funcionalidades, porém mantendo os componentes mais enxutos possíveis, então foi criado o JFACE.

  Entre algumas das funcionalidades oferecidas pelo JFACE estão a simplificação no controle de eventos. Em determinados momentos durante o desenvolvimento de uma aplicação, o desenvolvedor possui vários componentes visuais diferentes (botões, menus, campos) que executam uma mesma lógica, tendo que replicar o mesmo código para vários componentes. Usando JFACE basta criar uma ação e utilizar ela ao mesmo tempo em todos os componentes.

3.1.4 Sistema de Ajuda e Suporte a Grupo

A implementação do servidor *Web* de ajuda é feita pelo *plug-in* Help. Através deste *plug-in* são definidos pontos de extensão que outros *plug-ins* podem utilizar para contribuir com ajuda ou outra documentação do *plug-in*, como livros online. O servidor *web* de documentação inclui

facilidades especiais para permitir que os *plug-ins* referenciem arquivos por uso lógico, URLs baseadas em *plug-in* em vez do sistema de arquivos URLs.

O *plug-in* de suporte a grupo (*Team*) permite que outros *plug-ins* definam e registrem implementações para grupos de programação, acesso de repositório e controle de versões e recursos.

3.1.5 *JDT e PDE*

O JDT (*Java Development Tooling*) . extende o *workbench* da plataforma, disponibilizando um completo ambiente de desenvolvimento Java. O ambiente fornece um conjunto de ferramentas que viabilizam uma maior produtividade no processo de desenvolvimento de uma aplicação, possibilitando a automatização de uma série de tarefas. É composto por um conjunto de *plug-ins* nativos do Eclipse que auxiliam a visualizar, editar, compilar, depurar e rodar código Java. Algumas das melhorias obtidas com a utilização do ambiente JDT são os editores com características que destacam a sintaxe e completam o código, assim como as janelas que possibilitam a visualização da estrutura do código.

Devido a enorme variedade/diversidade de *plug-ins* disponíveis atualmente, desenvolvedores podem customizar seu ambiente de desenvolvimento de acordo com suas necessidade, automatizando/acelerando uma série de tarefas. A própria Fundação Eclipse é segmentada em projetos visando prover componentes estruturais ou ferramentas para várias destas finalidades. Também existem muitos *plug-ins* gratuitos e comerciais para o Eclipse desenvolvidos por terceiros.

O PDE (*Plug-in Development Environment*) fornece ferramentas que automatizam a criação, manipulação, depuração e instalação dos *plug-ins* desenvolvidos. Ele também é formado por uma série de *plug-ins* inseridos no Eclipse SDK (*System Development Kit*).

3.2 Plug-ins

Um *plug-in* é a menor unidade da plataforma funcional do Eclipse que pode ser desenvolvido separadamente. Geralmente uma pequena ferramenta é implementada em um simples *plug-in*, enquanto uma ferramenta mais complexa tem suas funcionalidades divididas sobre vários *plug-ins*.

Estruturalmente, todos os *plug-ins* estão localizados dentro do diretório raiz do Eclipse. Neste diretório existem diversos subdiretórios, cada um destes destinado a alocar *plug-ins* distintos. Para evitar conflitos de versões, e também para definir um método de normatização, é definido que o nome de cada diretório deve ser a junção do nome do *plug-in*, precedido de sua versão. A Figura 10 apresenta a estrutura básica dos *plug-ins* do Eclipse.

Como pode ser observado na figura, todo o *plug-in* é composto por pelo menos quatro

```
1: C:\Eclipse
2:  + --- plugins
3:  |        + --- com.prevedello.dmi_1.0.0
4:  |        |    + --- icons
5:  |        |    + --- lib
6:  |        |    + --- META-INF
7:  |        |    + --- src
8:  |        |    plugin.properties
9:  |        |    plugin.xml
10: |        + --- org.vaulttec.velocity.ui_1.0.2
11: |        |
12: |        ...
13: + --- readme
```

Figura 10 – Estrutura dos *plug-ins* do Eclipse.

arquivos:

- plugin.xml - Arquivo de manifesto do *plug-in* onde estão contidas informações descritivas que são utilizadas pelo Eclipse para integrar o *plug-in* com o *Framework*. Neste arquivo são definidos as extensões, pontos de extensão, pontos abertos para extensão (se existir) e a(s) classe(s) Java que será(ão) invocada(s) para a disponibilização do *plug-in*. O Eclipse disponibiliza uma ferramenta denominada *Plug-in Manifest Editor* que auxilia na criação e configuração deste arquivo.

- plugin.properties - Arquivo que contém informações de apoio ao *plug-in*. Na seção 4.1 será descrita a implementação de um *plug-in* onde este arquivo foi utilizado como base para a definição de mensagens textuais (linhas 2, 5, 8 e 11) e como *path* para arquivos utilizados como *template* para geração de código (linha 13).

```
1:  titulo=DMI Wizard
2:  descricao=Criação de DMI, Role, Shared Object, DMI Aninhada
3:
4:  titulo2=Shared Objects
5:  descricao2=Exportar um Shared Object para uma ou mais Roles
6:
7:  titulo3=Exceptions
8:  descricao3=Associar uma Exception a DMI's distintas
9:
10: titulo4=DMI Aninhadas
11: descricao4=Associar as Roles da DMI Aninhada com a DMI
12:
13: templates.dir=c:/templates/dmi
14:
15: #mode=development
```

Figura 11 – Arquivo *plug-in.properties*.

- arquivos .jar - Código Java necessário para o *plug-in*;

- ícones, arquivos de documentação e demais arquivos de apoio necessários ao *plug-in*;

Sempre que o Eclipse é inicializado a plataforma de execução descobre dinamicamente todos os *plug-ins* disponíveis, e monta uma lista com tais registros. Os *plug-ins* só são efetivamente ativados quando há requisições de suas funcionalidades (pontos de extensão são invoca-

dos), desta forma, diminuindo o tempo de inicialização do Eclipse e a utilização em demasia de recursos de *hardware*, em especial memória RAM.

Atualmente não foi definido nenhum método capaz de descarregar *plug-ins*, porém existem projetos em andamento para possibilitar a carga e descarga de *plug-ins* sob demanda. Este é um dos objetivos do projeto Equinox [28], que explora aproximações e tecnologias para aumentar a flexibilidade da plataforma de execução do Eclipse.

3.3 Plug-ins implementados no Eclipse

O Eclipse é composto por uma plataforma totalmente modular, composta por diversos *plug-ins*. Um destes *plug-ins* é denominado PDE (*Plug-in Development Environment*), através dele é disponibilizado aos desenvolvedores de aplicações um ambiente totalmente voltado para a criação de novos *plug-ins*, ou a customização de *plug-ins* já existentes.

Esta seção apresenta alguns projetos (*plug-ins*) desenvolvidos sobre este ambiente, comprovando assim a importância e os benefícios obtidos com a adoção deste paradigma, tanto no meio acadêmico como no meio profissional.

3.3.1 Penumbra: Um plug-in Eclipse para Introdução a Programação.

PENUMBRA [29] é um *plug-in* para o Eclipse desenvolvido na universidade de Purdue, com objetivo de facilitar o processo de aprendizagem de alunos do curso de Ciência da Computação aos conceitos de Programação.

Este *plug-in* possibilita simplificar o uso das funcionalidades do Eclipse, apresentando um ambiente simples, intuitivo e conseqüentemente mais produtivo. A perspectiva do *plug-in* é bastante simples, e nela estão contidos apenas os elementos necessários para introdução dos conceitos básicos de programação. Como exemplo da simplificação do ambiente, podemos levar em consideração a alteração na maneira de executar uma aplicação, bastando um simples clique no botão do mouse.

A utilização do *plug-in* em classe evidenciou que os benefícios apresentados justificam o investimento de tempo e esforço para a capacitar os alunos, visto que as respostas obtidas foram muito positivas. Os alunos uma vez familiarizados com a IDE conseguiram identificar e resolver de uma forma mais eficaz e rápida os problemas propostos na disciplina.

Outros *plug-ins* diretamente focados em assuntos pedagógicos de ambientes de programação, são BlueJ [30] e GILD [31].

3.3.2 JIRiSS - Um plug-in Eclipse para exploração de código fonte

JIRiSS (*Information Retrieval based Search for Java*) [32] é uma ferramenta de exploração de software que utiliza um mecanismo de indexação baseado em um método da recuperação de informação. JIRiSS é um melhoramento do plug-in IRiSS [33] que foi desenvolvido na paltaforma MS Visual Studio, com o objetivo de analisar códigos implementados em C++.

JIRiSS é implementado como um *plug-in* Eclipse e permite ao usuário pesquisar código fonte Java para a implementação de conceitos formulados através de consultas baseadas em linguagem natural. Os resultados das consultas são apresentados através de uma espessa lista de métodos ou classes, ordenados pela similaridade da consulta do usuário.

JIRiSS também incorpora algumas características avançadas, tais como, pesquisas baseadas em fragmentos, verificação da ortografia das consultas executadas, sugestão de palavras para aprimorar a consulta e opções avançadas para customizar e indexar processos.

Existem diversas ferramentas capazes de suportar pesquisas, algumas delas baseadas em *plug-ins* de diferentes IDEs, dentre as quais detacam-se: FEAT [34], que é um *plug-in* Eclipse que captura o conhecimento relacionado à implementação de código fonte. Este *plug-in* suporta localização, descrição e análise de código relacionados ao código de implementação em Java, e Jripples, que é um *plug-in* Eclipse que suporta diferentes estágios de trocas incrementais e fornece aos programadores meios para a investigação e compreensão do programa, permitindo a visualização do *software* usando o grafo de dependência do programa.

3.3.3 Sangam - Um plug-in Eclipse para Duplas de Programadores Distribuídos

O *plug-in* Sangam [35] foi desenvolvido com objetivo de integrar pares de programadores remotamente distribuídos. O *plug-in* possibilita que usuários da plataforma Eclipse localizados remotamente possam compartilhar a mesma área de trabalho (*workspace*), como se estivessem utilizando o mesmo computador ao mesmo tempo.

O *plug-in* fornece uma interface específica para pares de programadores distribuídos (*Distributed Pair Program*), e sincroniza o ambiente de desenvolvimento para ambos os programadores. O *plug-in* é constantemente atualizado, e contempla algumas importantes funcionalidades, tais como:

- Sincronização do Editor: Isto inclui digitação, seleção, abertura, fechamento, e sincronização das visões.

- Sincronização de execução: Os programadores podem rodar ou depurar a mesma aplicação Java ao mesmo tempo.

3.3.4 KenyaEclipse: Aprendendo a Programar no Eclipse

KenyaEclipse [36] é um *plug-in* para o Eclipse que tem por objetivo auxiliar o processo de aprendizagem dos conceitos básicos de programação à estudantes iniciantes dos cursos da área de Ciências da Computação do *Imperial London College*.

Este *plug-in* possibilita aos estudantes a utilização de um ambiente IDE de produção, onde são diponibilizados diversos recursos que auxiliam no processo de aprendizagem e desenvolvimento de software, tais como, sugestões de melhorias no estilo de programação e vistas (janelas) que possibilitam uma visão mais clara do sistema como um todo.

A utilização do *KenyaEclipse* além de auxiliar o processo de aprendizagem também facilita a transição dos estudantes para qualquer outra linguagem de programação que utiliza ambientes similares, tal como Java.

Anteriormente ao desenvolvimento do *plug-in KenyaEclipse* as noções básicas de programação eram passadas aos estudantes do Colégio Imperial de Londres através da utilização da linguagem de programação *Kenya* [37]. Tal linguagem não oferecia aos estudantes muitos dos recursos disponíveis em uma IDE.

Outros *plug-ins* desenvolvidos com o objetivo de auxiliar no processo de aprendizagem dos conceitos básicos de programação e que merecem ser mencionados são: *Penumbra* [29], *DrJava* [38] e *Watcher* [39].

3.3.5 Plug-in para monitorar o comportamento do programador

Watcher [39] é um *plug-in* desenvolvido para o Eclipse que tem como principal objetivo monitorar o comportamento dos programadores. O *plug-in* rastreia comportamentos significantes como movimentos do mouse, movimentos da barra de rolagem e alterações no documento que ocorrem durante a sessão. A ferramenta é composta por três componentes distintos, e a combinação destes componentes produz um detalhado conjunto de informações que pode ser utilizado para apontar áreas de desigualdade entre diferentes programadores e técnicas de programação.

O Watcher é inicializado em *background* sempre que o Eclipse é carregado. Toda vez que este *plug-in* for invocado é automaticamente criado um arquivo *xml* que armazena todos os eventos gerados sobre a plataforma. O programador não tem nenhuma interação direta com o *plug-in*, desta forma, assegurando-se a integridade das informações coletadas.

Após coletadas as informações, estas são repassadas para uma ferramenta denominada *Xml2Excel*, que recebe os dados coletados do *plug-in* Watcher, abstraí os dados considerados relevantes e converte estes dados para um formato Excel.

O terceiro e último componente da solução consiste da uma ferramenta denominada *Watcher-*

Macros que consiste basicamente de macros desenvolvidos dentro do próprio Excel, que possibilitam que os dados colhidos em etapas anteriores sejam expressos de forma gráfica.

O *plug-in* tem se mostrado bastante eficaz principalmente no meio acadêmico, uma vez que possibilita a identificação das áreas em que os estudantes estão tendo maiores dificuldades e conseqüentemente perdendo mais tempo, permitindo assim, que os professores possam explorar estas deficiências em sala de aula ou indicar material de apoio apropriado.

3.3.6 SpeechClipse: Comandos de fala para Eclipse

SpeechClipse [40] é um *plug-in* para o Eclipse que possibilita que desenvolvedores de *software* possam efetuar várias outras tarefas ao mesmo tempo em que estão programando. Este *plug-in* possibilita que sejam passados comandos para o Eclipse através de voz, sem que haja a necessidade de utilização de qualquer dispositivo de entrada, tais como mouse ou teclado. Tal funcionalidade além de permitir um aumento de produtividade, também se mostra especialmente útil a desenvolvedores de software portadores de deficiências físicas.

O *plug-in* ainda precisa ser aprimorado, visto que é extremamente sensível a barulhos externos, acarretando a invocação de ações indesejadas na IDE. Possíveis melhorias podem ser atingidas através da utilização de um dispositivo capaz de limitar determinados tipos de ruídos ou simplemente melhorar a gramática para ajudar a distinguir entre um comando válido ou inválido.

A configuração de *hardware* miníma requerida para que o *plug-in* opere de forma satisfatória é referente a um computador com processador Pentim III, com *clock* de 800 MHz e 512 MB de memória RAM. A configuração ideal é um computador com com processador Pentim 4 com 512 MB de memória RAM.

3.3.7 Usando Eclipse para ensinar engenharia de software a distância

Algumas ferramentas utilizadas conjuntamente podem fornecer os requisitos necessários para dar suporte a estudantes remotamente distantes, fornecendo um ambiente de educação a distância colaborativa e um sistema de gerência de projeto de software.

Tais funcionalidades podem ser atingidas através da integração de duas ferramentas: o CURE [41]que consiste de uma plataforma de colaboração padrão e a solução de gerenciamento de projeto CodeBeamer. O CURE [] foi integrado por um *plug-in* desenvolvido recentemente que possibilita a utilização de ferramentas como wiki, mail, chat com Eclipse.

Juntos com *plug-ins* adicionais, esta plataforma facilita o projeto de engenharia de software em todas as fases, e os estudantes poderão colaborar mais intensamente, gerenciar seus projetos mais facilmente e garantir software de alta qualidade.

3.3.8 Jazzing: Eclipse como ferramenta colaborativa

A comunicação entre um grupo de desenvolvedores que compartilham um mesmo espaço físico é extremamente eficiente, visto a possibilidade de integração entre dos membros. A troca de idéias e o compartilhamento de informações se dá de forma rápida, clara e ágil.

O *plug-in* Jazz [42] tem por objetivo possibilitar que um pequeno número de desenvolvedores remotamente localizados, possam trabalhar conjuntamente de forma eficiente e produtiva. O *plug-in* fornece ferramentas que auxiliam no processo de comunicação e coordenação entre os membros da equipe.

3.3.9 DrJava: Uma interface amigável para Eclipse

Estudantes que são forçados a aprender Java utilizando um editor de texto convencional e linhas de comando como interface de execução, apresentam uma enorme dificuldade em lidar com tais mecanismos, tornando o processo de aprendizagem mais lento e tortuoso.

Por esta razão muitos professores optam pela utilização de um ambiente de desenvolvimento integrado (IDE) para Java, afim de auxiliar seus alunos durante o processo de compreensão dos conceitos fundamentais de orientação a objetos e o desenvolvimento de programas.

DrJava [43] fornece um ambiente mais simples, amigável, altamente interativo e totalmente focado no processo de ensinar Java para alunos iniciantes, amenizando os desafios envolvidos em utilizar uma IDE. O *plug-in* consiste basicamente de duas janelas, uma onde é adicionado e editado o código do programa, e a outra onde são avaliadas as sentenças e expressões adicionadas. Esta simples interface libera os estudantes da complicação de definição de métodos e os encoraja a conduzir simples experiências.

4 Descrição da Proposta

Conforme mencionado anteriormente, até o presente momento diversas aplicações já foram desenvolvidas com a utilização de DMIs. Entretanto, o desenvolvimento de aplicações com DMIs podem ser suscetível a inserção de falhas durante a fase de programação, pois para o uso de DMIs o programador deve seguir uma sequência de passos sem margem para erros, por exemplo, se um dos gerentes passados por parâmetro durante a criação de um objeto `Role` for errado, o processo de sincronização no início da DMI entrará em *deadlock*, pois faltará um participante para sincronizar. O mesmo vale para o momento da criação da tabela de exceções utilizada para o tratamento de exceções. Se uma entrada da tabela estiver equivocada, todo o processo de tratamento de exceções poderá gerar problemas. Diversas outras situações poderiam ser citadas.

Situação semelhante acontece em relação ao desenvolvimento de expansões para o Eclipse, uma vez que diversos *plug-ins* já foram desenvolvidos para diversas áreas, mas até onde sabemos nenhum trabalha com a possibilidade de desenvolver aplicações tolerante a falhas que utilizem mecanismos como DMIs, bloco de recuperações ou programação n-versões, por exemplo.

Desta forma a idéia principal deste capítulo é a união da possibilidade de desenvolvimento de aplicações tolerante a falhas com uma maneira automatizada de fazê-lo em um ambiente que auxilie o desenvolvimento de software. O sistema que será descrito neste capítulo é composto da implementação de um *plug-in* para o Eclipse que será responsável pela leitura da descrição de todas as DMIs a serem instanciadas para o *framework* apresentado na Seção 2.2. A descrição das DMIs mais o código que utiliza estas DMIs formarão o código completo da aplicação tolerante a falhas.

O *plug-in* é invocado através de um ponto de extensão adicionado na barra de ferramentas do Eclipse (conforme será descrito na Seção 4.2). A partir de então é disponibilizada uma interface gráfica onde é possível definir quais componentes serão criados e de que maneira os mesmos estarão relacionados. A interface gráfica eliminará toda a parte de instanciação de objetos e mesmo exceções e suas relações no momento da criação dos objetos.

Uma vez definidos os componentes (DMI´s, Roles, Objetos Compartilhados...) e seus eventuais relacionamentos, o *plug-in* gera a estrutura de arquivos (pacotes e classes) condizente com os relacionamentos criados (conforme será descrito na Seção 4.2) e em seguida as classes geradas são populadas com o código correspondente.

O código inserido nas classes é oriundo das informações adicionadas na interface gráfica do *plug-in*, associadas a arquivos de *templates* criados em um editor de texto convencional (formato

.txt) e estruturados de acordo com a linguagem de *templates* denominada *Velocity* [44].

A Figura 12 demonstra como o *plug-in* é conectado com o ambiente do Eclipse.

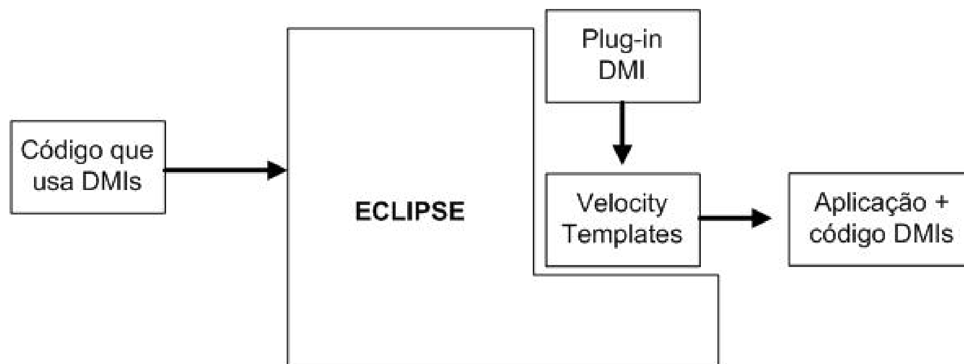

Figura 12 – Ligação do plug-in com o Eclipse.

O *plug-in* DMI foi implementado utilizando a linguagem de programação Java. A escolha foi motivada por dois fatores: esta linguagem é largamente utilizada por desenvolvedores de *plug-ins*, assim como é a mesma linguagem utilizada para o desenvolvimento do *framework* conforme descrito no Capítulo 2.

4.1 Implementação

Conforme descrito no Capítulo 3, o Eclipse é um sistema composto por uma enorme variedade de *plug-ins*, onde cada um destes *plug-ins* contribuem de uma maneira distinta com o ambiente. Alguns *plug-ins* disponibilizam funcionalidades relacionadas à implementação de interfaces com usuário, outros fornecem suporte para gerenciar os recursos do sistema, e assim por diante.

No desenvolvimento do *plug-in* DMI foi necessário a utilização de seis *plug-ins*. Entretanto, o sistema foi basicamente implementado sobre o *plug-in* (`org.eclipse.ui`). Este *plug-in* disponibiliza uma completa API (*Application Programming Interface*) para implementação de componentes de interface com usuário. A Figura 13 demonstra os *plug-ins* utilizados e as extensões definidas sobre o *plug-in* `org.eclipse.ui`. A extensão `org.eclipse.ui.newWizards` foi definida para possibilitar a utilização de *wizards* em nosso *plug-in*, enquanto a extensão `org.eclipse.ui.actionSets` foi definida de forma a possibilitar a inclusão de um ícone (onde o *plug-in* é invocado) na barra de ferramentas do Eclipse.

Conforme já mencionado, a interface do *plug-in* foi implementada sob o formato de *wizard*. Optamos pela adoção deste modelo visto a praticidade de implementação fornecida pela API do *plug-in* `org.eclipse.ui`. A utilização de um *wizard* também direciona a construção das DMIs, facilitando o entendimento do conceito de DMIs.

O *wizard* é composto por um total de quatro telas. Todos os componentes são criados na

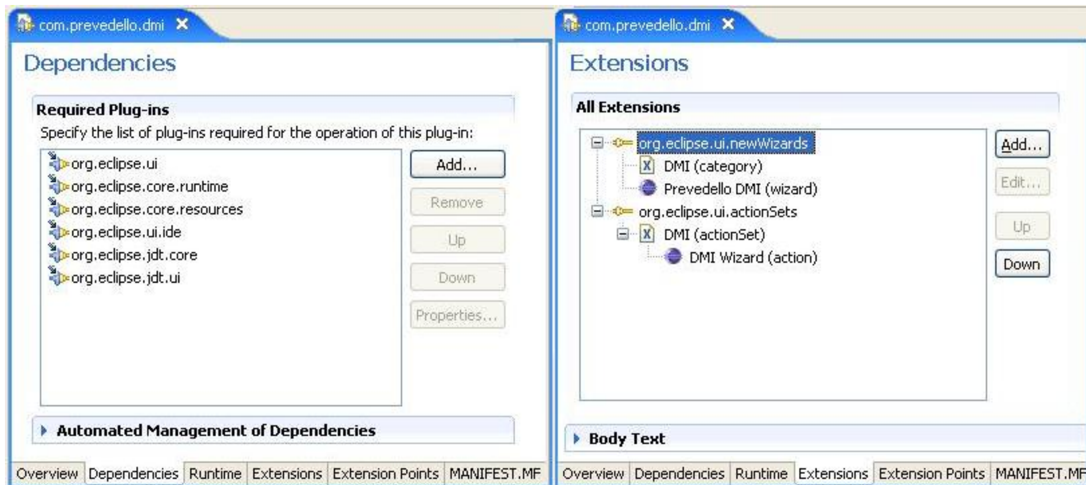

Figura 13 – Dependências para criação do *plug-in* DMI.

tela inicial, enquanto nas telas posteriores somente é definido como os componentes se relacionam entre si. Os componentes são basicamente os elementos que compõem o *framework* para construção de DMIs.

Todos os componentes inseridos na tela inicial do *wizard* são armazenados em um estrutura de dados denominada *HashMap*. Esta estrutura é composta por três níveis que operam de forma encadeada. O primeiro nível contém um indicador para a tabela de cada tipo de elementos criados no *wizard*, ou seja, "DMI, "Role", "SharedObject", "Exception"e "DMIAninhada". O segundo nível contém os elementos de cada tipo. O terceiro nível contém os valores associados a cada um dos elementos.

Para facilitar a compreensão do modelo proposto vamos supor a existência de uma DMI (`DMI0`) composta por dois papéis (`Role0` e `Role1`). A Figura 14 exemplifica a situação descrita.

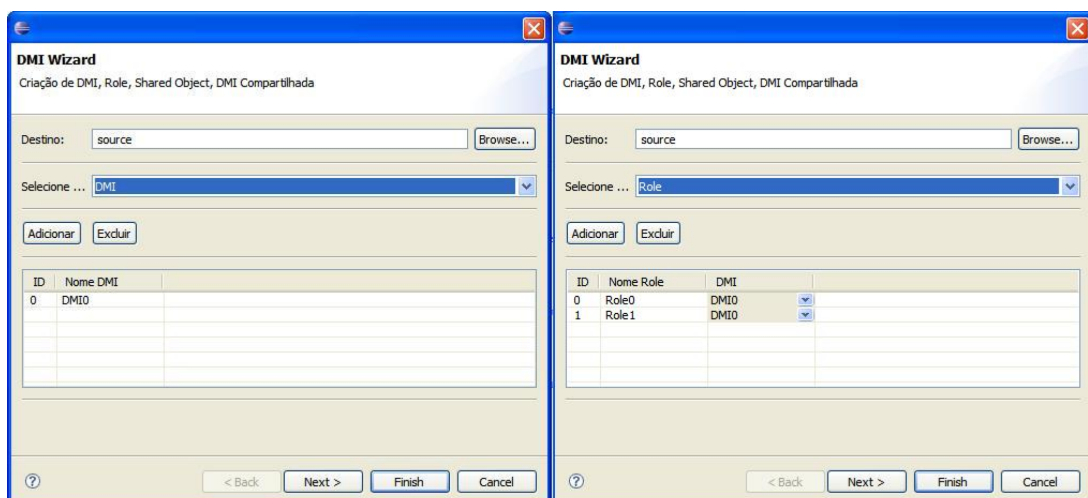

Figura 14 – DMI com dois papéis.

Os componentes inseridos na primeira tela do *wizard* são estruturados de acordo com a

estrutura que será apresentada abaixo. No código abaixo representamos a DMI utilizando XML. Este mesmo formato é utilizado para armazenar as DMIs criadas pelo *plug-in*. Conforme pode ser observado da linha 2 até a linha 19 temos a descrição da DMI existente, enquanto que da linha 20 até a linha 58 temos a descrição dos papéis. Cada nível pode ser identificado pela palavra reservada `entry`.

```
 1:<hashMap>
 2:  <entry>
 3:    <key>DMI</key>          [Elemento]
 4:    <hashMap>
 5:      <entry>
 6:        <key>0</key>        [Linha]
 7:        <hashMap>
 8:          <entry>
 9:           <key>0</key>     [Coluna)
10:           <key>0</key>      (Valor)]
11:          </entry>
12:          <entry>
13:            <key>1</key>    [(Coluna)
14:            <key>DMI0</key>  (Valor)]
15:          </entry>
16:        </hashMap>
17:      </entry>
18:    </hashMap>
19:  </entry>
20:  <entry>
21:    <key>Role</key>         [Elemento]
22:    <hashMap>
23:      <entry>
24:        <key>0</key>        [Linha]
25:        <hashMap>
26:          <entry>
27:            <key>0</key>    [Coluna)
28:            <key>0</key>     (Valor)]
29:          </entry>
30:          <entry>
31:            <key>1</key>    [Coluna)
32:            <key>Role0</key> (Valor)]
33:          </entry>
34:          <entry>
35:            <key>2</key>    [(Coluna)
36:            <key>DMI0</key>  (Valor)]
37:          </entry>
38:        </hashMap>
39:      </entry>
  :        :
57:    </hashMap>
58:  </entry>
59:</hashMap>
```

Uma vez definidos os elementos e seus eventuais relacionamentos estes dados são armazenados em *HashMap*. A partir da *HashMap* é possível efetuar consultas e transformar nomes (inseridos no *wizard*) em objetos.

Outro aspecto importante a ressaltar em relação a implementação é a forma como o código é gerado. Assim como estamos propondo uma estratégia para facilitar a criação de código para as DMIs, evitando assim que falhas sejam inseridas durante a fase de codificação, utilizamos um ambiente para facilitar a geração de código. Para que as classes do *framework* sejam populadas com o código correspondente, o *plug-in* utiliza alguns arquivos de *template* (conforme mostrado na Figura 14). Tais arquivos são criados em um editor de texto convencional e são salvos

em formato textual. Os arquivos são estruturados de acordo com a linguagem de *templates* denominada *velocity* [44].

A utilização do *velocity* torna-se uma alternativa extremamente útil para geração de código, pois viabiliza separar os *templates* (arquivos.txt) do código Java. Utilizando o *velocity* os *templates* ficam muito próximos do resultado final, assim como também torna-se possível a utilização de expressões como `foreach` e `if`, tornando os templates dinâmicos.

```
1:  public static void main(String[] args) {
2:
3:      try {
4:  #foreach($dmi in $dmis)
5:  #if($dmi.hasException())
6:  #parse("Manager-exception.txt")
7:  #else
8:  #parse("Manager-normal.txt")
9:  #end
10: #end
11:          }
12:   catch (Exception e)
13:   {
14:     System.out.println("Error running threads..." + e.getMessage());
15:     e.printStackTrace();
16:   }
17: }
```

Figura 15 – *Template* para a geração dos objetos `Manager`.

A Figura 15 mostra parte do código do *template* `Manager.txt`, onde é feito um `foreach` para percorrer todas as DMIs (linha 4) e gerar o seu respectivo código. Dentro do `loop` é feito um teste onde é verificado se a DMI possui ou não exceções (linha 5). Se a DMI possuir exceções relacionadas a ela, então é utilizado o *template* `Manager-exception` (linha 6), senão é utilizado o *template* `Manager-normal` (linha 8). Após definido o *template* que será utilizado para gerar o código, os parâmetros (váriaveis) armazenados no arquivo *HashMap* são recuperados e inseridos na estrutura do *template*.

O código gerado pelos *templates* é estruturado na forma de pacotes. Para cada DMI inserida no *wizard* é criado um pacote com seu próprio nome, e dentro deste pacote são criadas classes para os papéis associados a esta DMI. O nome das classes corresponde ao nome inserido no *wizard*. A Figura 16 ilustra esta situação, onde a DMI (`DMI0`) é composta por três papéis (`Role0, Role1, Role2`).

A mesma estratégia é utiilizada para DMIs que possuem exceções a serem tratadas. Ou seja, sempre que houver alguma exceção associada à DMI é criado um novo pacote dentro do pacote da DMI correspondente denominado `exceptions`. Dentro do pacote `exceptions` é criado um outro pacote, que corresponde ao nome da exceção inserida no *wizard*. Dentro deste último pacote são criadas as classes para os papéis associados a esta exceção, assim como uma classe que corresponde ao nome da exceção. A Figura 17 ilustra esta situação, onde a DMI (`DMI0`) é composta por três papéis (`Role0, Role1,Role2`) e três exceções

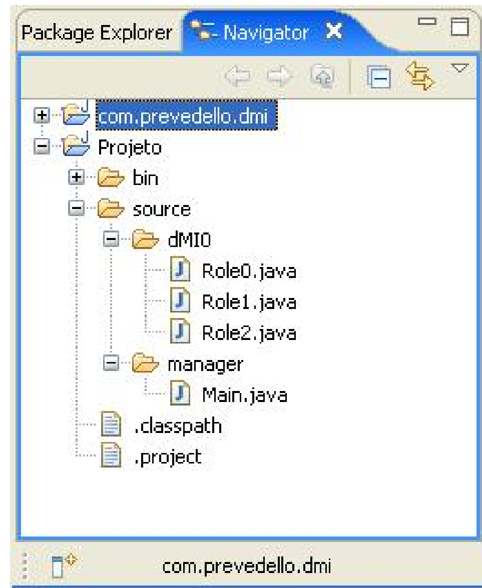

Figura 16 – Estrutura dos pacotes/classes.

(`exception0, exception1, exception2`). As exceções `exception0` e `exception2` são tratadores de exceção para a `DMI0` enquanto a `exception1` é o tratador de exceção para `exception0`.

O último pacote criado durante a geração é denominado `manager`. Nesta fase é gerado o código principal para criação de todas as DMIs definidas pelo usuário do *plug-in*,

4.2 Uso do *plug-in* DMI

Uma vez o *plug-in* instalado, basta inicializar o Eclipse para que a ferramenta esteja disponível aos desenvolvedores. O *plug-in* é invocado através de um ícone localizado na barra de ferramentas do Eclipse, conforme pode ser observado na Figura 18.

Uma vez este ícone pressionado é disponibilizado ao desenvolvedor uma interface gráfica conforme pode ser observado na Figura 19. Inicialmente é necessário que seja definido o nome do projeto corrente (o botão *Browse* pode auxiliar nesta tarefa caso a pasta não esteja previamente selecionada), em seguida basta que sejam definidos os componentes da DMI.

Conforme mencionado anteriormente, a interface do *plug-in* foi implementada sob o formato de *wizard*, e é composta por um total de quatro telas. Neste modelo de implementação todos os componentes necessários para a criação de uma DMI estão centralizados na primeira tela do *plug-in*, enquanto a maioria dos eventuais relacionamentos são definidos nas telas posteriores.

Conforme pode ser observado na Figura 19, o desenvolvedor pode criar até 5 componentes distintos, são eles: DMI, Role, SharedObject, Exception e DMIAninhada. Após definido o tipo de componente a ser criado, é necessário pressionar o botão Adicionar de maneira que

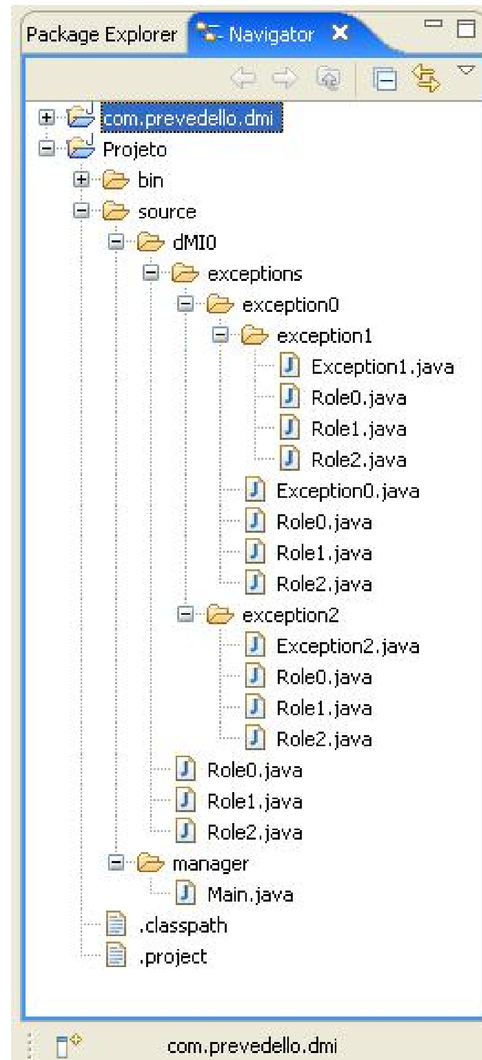

Figura 17 – Estrutura dos pacotes/classes.

Figura 18 – Ponto de Extensão para invocar o *plug-in* DMI.

este componente seja efetivamente criado. A nomenclatura padrão definida para os nomes dos componentes é composta por [componente + id], porém todos os componentes podem ser renomeados a qualquer instante. Para que o nome de um determinado componente possa ser alterado, basta clicar duas vezes (*double click*) com o botão esquerdo do mouse sobre o nome do componente, e em seguida renomea-lo. Sempre que existir a necessidade de remover algum componente criado erroneamente, basta selecionar o componente e clicar no botão Excluir.

A Figura 20 é referente a definição de um componente do tipo DMI, cujo nome do componente é `DMI0` (nomenclatura padrão). Também é importante destacar que o *plug-in* foi im-

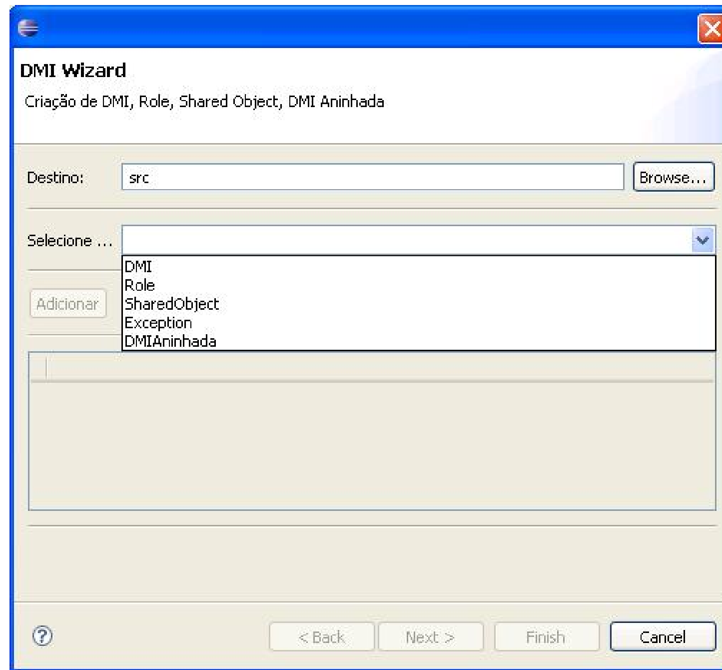

Figura 19 – Tela inicial do *plug-in* DMI.

plementado de maneira a garantir uma ordem de criação dos componentes, neste caso especial percebe-se que somente criando o componente `DMI0` não é possível prosseguir para as telas seguintes (apenas o botão Cancel encontra-se ativo). Na parte superior da tela são apresentadas mensagens textuais que tem por objetivo informar o usuário sobre qual componente ainda se faz necessário porém ainda não foi definido. Neste caso a mensagem de erro apresentada é referente a necessidade de criação de papéis.

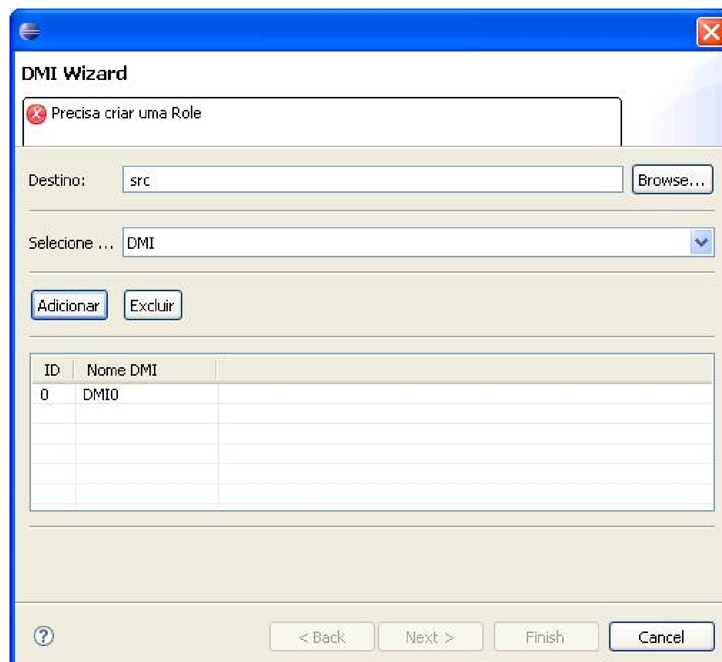

Figura 20 – Criando uma DMI.

O processo para a criação de componentes do tipo `Role` é praticamente idêntico ao de criação da DMI, bastando na tela inicial selecionar o elemento do tipo Role e em seguida adicionar quantos forem necessários. É imprescendível que cada papel seja associado a uma DMI previamente definida. A Figura 21 demonstra a criação de 5 papéis `[role0-role4]` que foram associados a `DMI0`. Conforme pode ser observado na parte superior da Figura 21 consta uma mensagem textual alertando que os papéis devem ser associados a DMI.

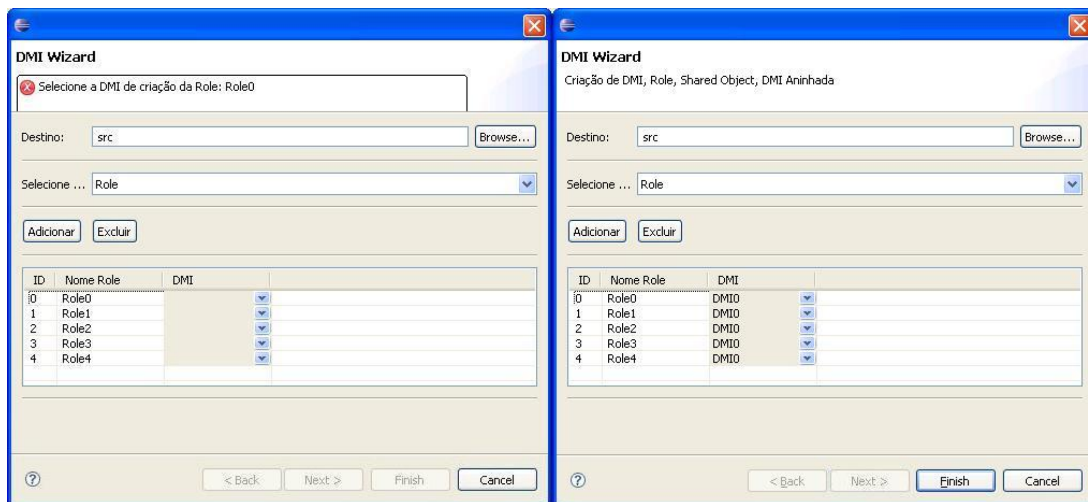

Figura 21 – Criando papéis.

Após realizada tal associação a mensagem textual desaparece, e o botão `Finish` (localizado na parte inferior da tela) é então habilitado. A partir de então basta que o botão `Finish` seja pressionado de forma que a estrutura de arquivos com seus respectivos códigos sejam gerados, conforme mencionado na Seção 4.

Para criar objetos compartilhados basta na tela inicial selecionar a opção `SharedObject`, e em seguida criar tantos elementos quanto necessários. Em seguida é necessário que seja definido qual papel irá gerar e exportar este objeto compartilhado. Vamos tomar como exemplo a criação de apenas um objeto compartilhado, denomiado `SharedObject0` que será criado no `Role0`. A Figura 22 detalha o ambiente descrito.

A partir de então percebe-se que na parte inferior da Figura 22 somente os botões `Next` e `Cancel` estão habilitados. Pressionando o botão `Next` é apresentada a segunda tela do *wizard*. Nesta tela é possível recuperar o objeto compartilhado criado na tela anterior e definir para qual(is) papel(éis) este objeto será exportado.

A tela é composta por duas *listbox*. A primeira delas localizada à esquerda da tela, intitulada "Todas as Roles", é onde estão contidos todos os papéis relacionados a DMI, exceto o papel selecionado na tela anterior, onde o objeto compartilhado é criado e exportado (*Role0*).

A segunda *listbox* localizada à direita da tela é intitulada "Roles Destino", e nela estão contidos todos os papéis que receberão o objeto compartilhado criado na tela anterior. Para que a lista possa ser populada são utilizados dois botões localizados entre as *listbox*, são eles: >> adiciona um papel por vez; >>> adiciona todos os papéis. Para que a lista seja limpa

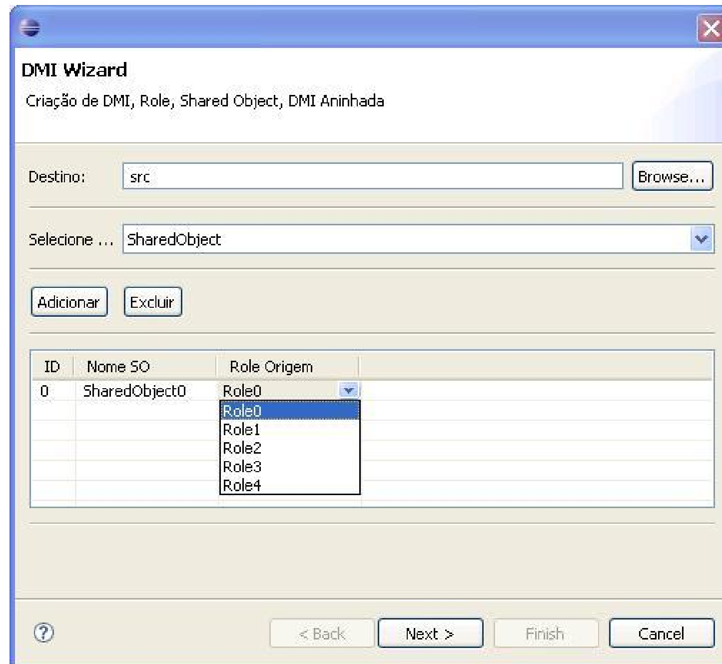

Figura 22 – Criando Objeto Compartilhado.

(totalmente ou parcialmente) são utilizados dois botões também localizados entre as *listbox*, são eles: << remove um papel por vez; e <<< remove todos os papéis. No exemplo apresentado na Figura 23 o objeto compartilhado `SharedObject0` que será criado no `Role0` e será utilizado pelo `Role1` e `Role4`.

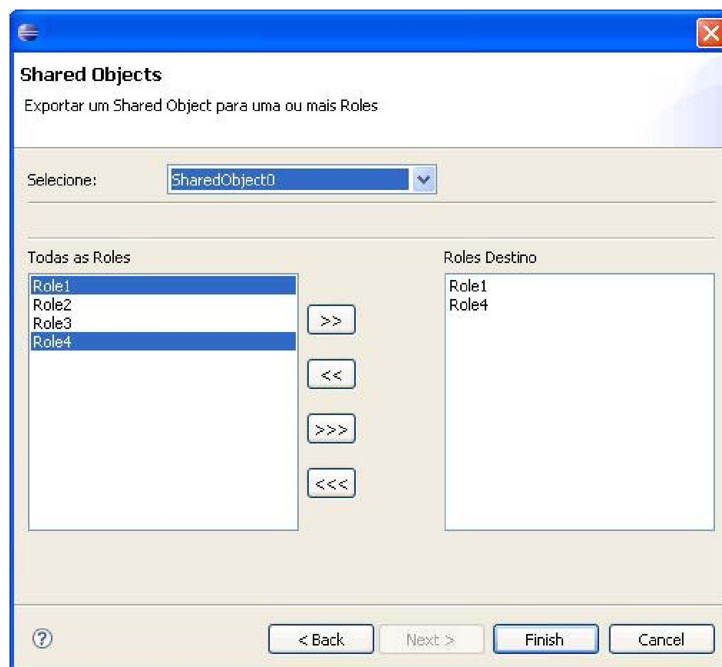

Figura 23 – Papéis que receberão Objeto Compartilhado.

Para criar tratadores de exceções basta na tela inicial do *wizard* selecionar a opção *Exception*, e em seguida criar tantos tratadores de exceções quanto necessário. Para exemplificar a

ação descrita vamos criar 3 tratadores de exceções (*Exception0*, *Exception1* e *Exception2*) (veja Figura 24).

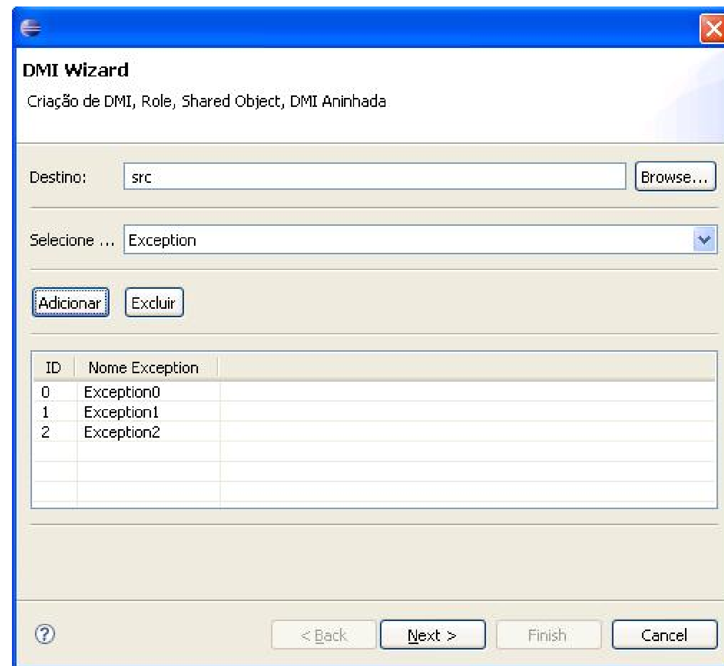

Figura 24 – Criando Exceção.

Na parte inferior desta figura percebe-se que apenas os botões `Next` e `Cancel` estão habilitados. Pressionando o botão `Next` é disponibilizada a terceira tela do *wizard*, uma interface onde é possível associar cada uma das exceções a uma DMI, assim como, se for o caso, associar uma determinada exceção a outra exceção (veja Figura 25).

Inicialmente é necessário selecionar cada uma das exceções criadas anteriormente e associa-las a uma DMI. Para tanto basta selecionar a exceção na *combobox* "Selecione" e em seguida na *listbox* "Todas as DMIs" selecionar a DMI em que esta exceção estará vinculada. Para finalizar a ação é necessário que a *listbox* "DMIs Relacionadas" seja populada com a DMI selecionada. Botões localizados entre as *listbox* executam tal associação. Após todas as exceções terem sido relacionadas a uma determinada DMI, também é possível efetuar a associação de uma exceção a outra exceção. Para tanto basta na *combobox* "Selecione" selecionar qual exceção que será a filha, e na *combobox* "Exception Pai" selecionar qual exceção será pai. Este processo indica que uma exceção será tratada pelo tratador de exceções correspondente. A Figura 25 é composta de duas partes. A figura da esquerda demonstra a situação onde a exceção `Exception0` esta sendo diretamente associada com a `DMI0`, enquanto a figura da direita demonstra a situação onde a exceção `Exception1` é filha da exceção `Exception0`.

Para que seja possível a criação de aninhamento entre DMIs basta na tela inicial selecionar a opção DMIAninhada e em seguida criar tantas DMIs quanto necessário. Para exemplificar a ação descrita vamos criar uma DMI aninhada denominada `DMIAninhada0`. Conforme pode ser observado o *plug-in* não permite a evolução para as telas seguintes, visto a necessidade da

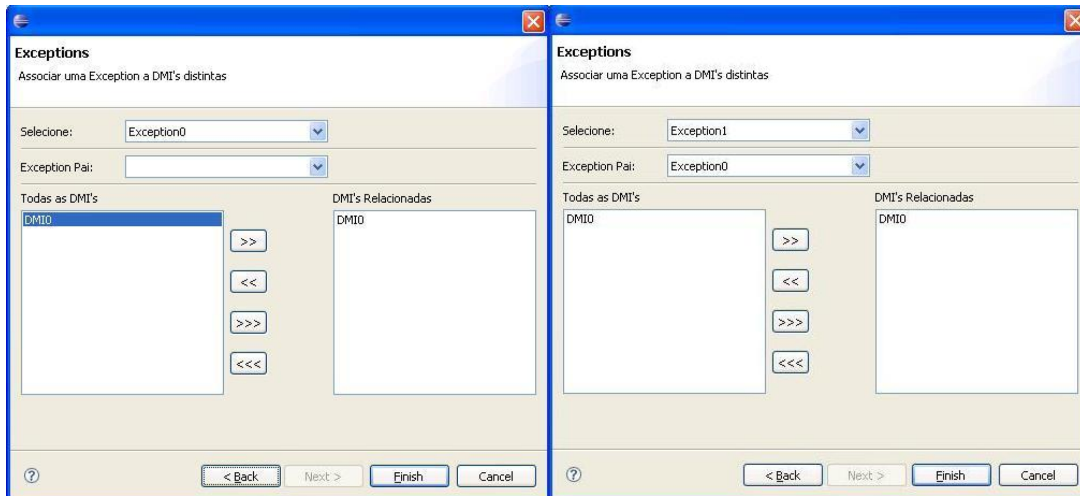

Figura 25 – Relacionando Exceção com DMI.

criação de papéis, que posteriormente devem ser associados a esta DMI. Desta forma, vamos criar dois elementos do tipo Role (`RoleA` e `RoleB`) e associar à DMI aninhada denominada `DMIAninhada0` a estes papéis. A Figura 26 detalha esta ação. A figura é divida em duas partes, a parte da esquerda detalha o criação da `DMIAninhada0` enquanto a parte da direita demonstra os papéis sendo associados a esta DMI.

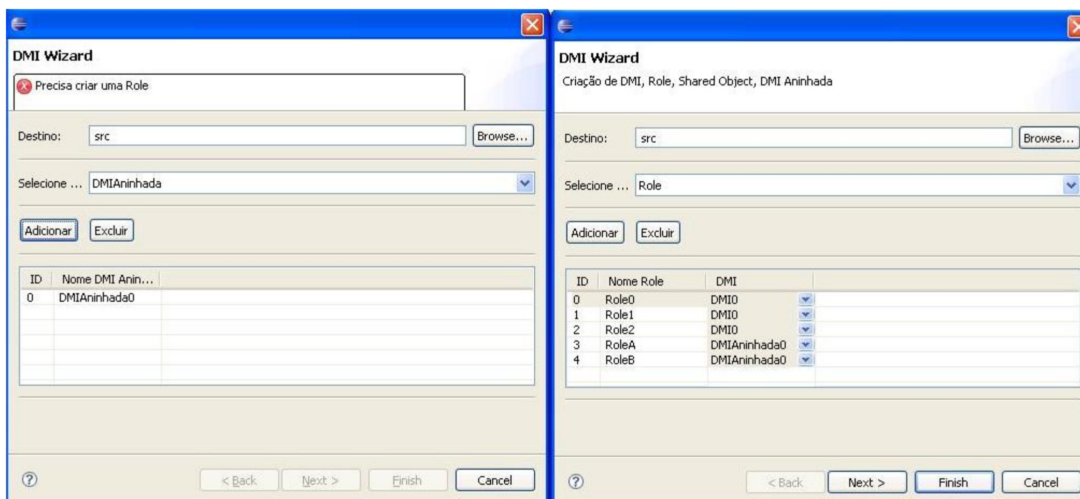

Figura 26 – Associando papéis a uma DMI Aninhada.

Pressionando o botão `Next` é disponibilizada a quarta e última tela do *wizard*, onde é possível recuperar todos os papéis associados à `DMIAninhada0` e `DMI0`. Na *combobox* “DMI-Aninhada” é possível visualizar todas as DMIs aninhadas previamente criadas, e na *combobox* “DMIPai” é possível visualizar todas as DMIs previamente criadas.

Uma vez selecionada qualquer DMI Aninhada no *combobox* “DMIAninhada” o *combobox* “Roles DMI Aninhada” é populado com todos os papéis associados a esta DMI. O mesmo ocorre quando selecionada qualquer DMI presente no *combobox* DMI Pai. Neste caso o *combobox* “Roles DMI Pai” é populado com todos os papéis associados a esta DMI. A Figura 27

demonstra a tela descrita.

Analisando a figura percebe-se que o `RoleA` da `DMIAninhada0` está relacionado com o `Role0` da `DMI0`, assim com o `RoleB` da `DMIAninhada0` esta associado com o `Role1` da `DMI0`.

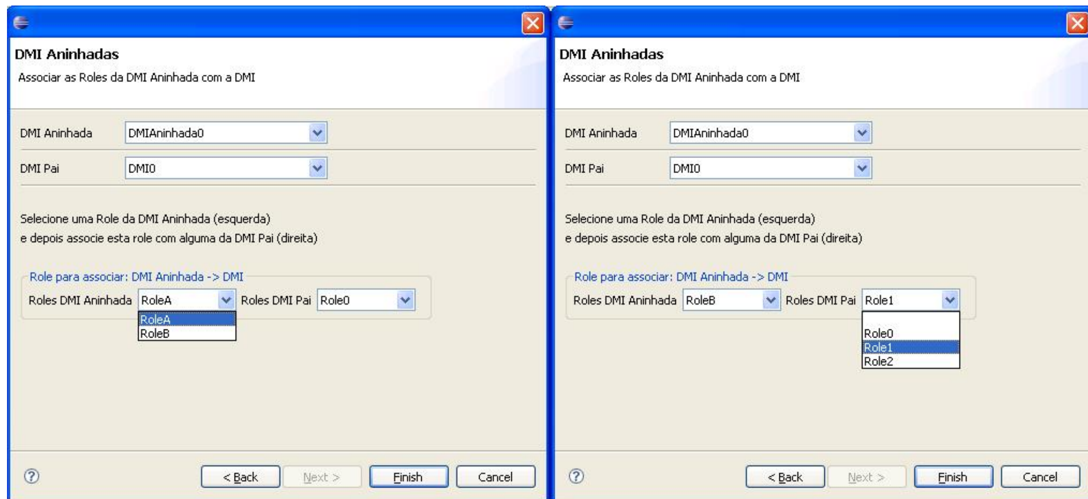

Figura 27 – Aninhando DMIs.

4.3 Estudo de Caso: O Jantar dos Filósofos

O problema do jantar dos filósofos [45] foi introduzido com o intuito de demonstrar como obter sincronização quando um processo precisa de mais de um recurso para executar uma atividade, e.g., um filósofo precisa de 2 garfos para iniciar a operação de comer. Note que na especificação original do problema somente um processo pode iniciar a operação. Para nosso propósito podemos considerar garfos como sendo processos, e a atividade comer acontecerá somente quando as 3 peças, ou seja filósofo e dois garfos, estão prontos para executar.

Definição: Vamos considerar 10 processos divididos em 2 grupos de 5 processos: garfos e filósofos, sendo garfos= garfo0,...,garfo4 e filósofos= filosofo0,..., filosofo4. Estes processos irão durante, a execução do sistema, executar atividades em conjunto, chamadas ações= ação0,...ação4 que são compostas de 2 componentes garfos e um componente filósofo. Cada componente da ação é uma tupla no formato acão*i*= garfo*i*, filosofo*i*, garfo*i*+1, enquanto 0 <= i <=4, and *i* +1 é equivalente a (i+*1*)mod 5. Também vamos considerar que cada ação*i* vai precisar de dados externos chamados prato afim de executar uma atividade comum, e que tais dados externos podem ser acessados somente por uma ação de cada vez. Para melhorar o entendimento do problema veja a Figura 28. Cada ação*i* irá executar uma atividade genérica chamada comer = (garfo-esquerdo, filósofo, garfo-direito) (veja Figura 29). Esta atividade é executada de forma coordenada pelos três participantes. Primeiramente, o filósofo e o garfo da esquerda irão sin-

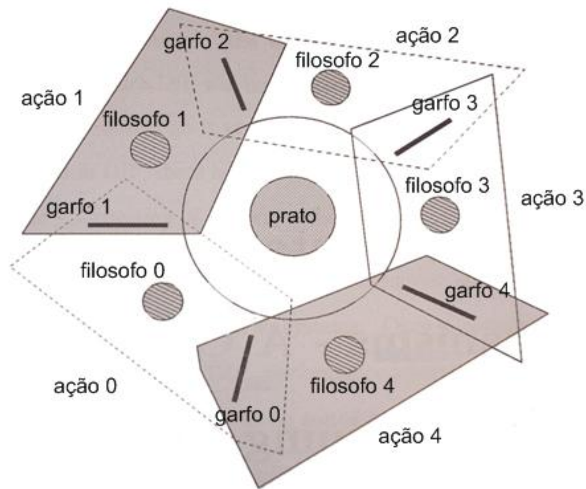

Figura 28 – Problema do Jantar dos Filósofos.

cronizar a execução, e tal sincronização irá representar o filósofo pegando o garfo da esquerda. Uma sincronização similar acontecerá mais tarde, representando o filósofo pegando o garfo da direita. Após, ambos os garfos irão sincronizar suas execuções para acessar os dados externos prato.

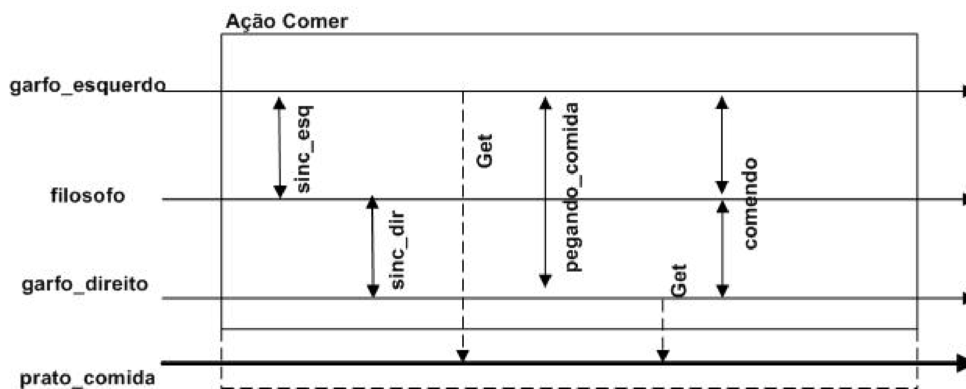

Figura 29 – Ação Comer.

O Apêndice A apresenta o conjunto de telas do *plug-in* para a geração do problema dos filósofos, enquanto que o apêndice B apresenta o código gerado.

5 Conclusão

Durante os últimos anos diversos trabalhos foram desenvolvidos com um mecanismo para tolerância a falhas em sistemas concorrentes, ou seja, com a utilização de Interações Multiparticipantes Confiáveis (DMIs). Estes sistemas foram utilizados para controlar sistemas de manufatura, controle de aplicações web e sistemas de controle de pacientes, entre outros. Entretanto, conforme já mencionamos durante este documento, todos estes trabalhos fizeram a utilização deste mecanismo de uma maneira manual. Apesar deste mecanismo melhorar e auxiliar na construção de sistemas confiáveis, o processo de utilização deste mecanismo pode trazer alguns situações indesejadas. Entre estas situações podemos citar, principalmente, a introdução de falhas no código final gerado pelo sistema. Como o usuário do framework que possibilita a construção de sistemas com este mecanismo não possuir nenhum ambiente de apoio, o processo de utilização também é muito demorado.

Devido a estas carências, este trabalho buscou alternativas para o desenvolvimento de sistemas confiáveis que utilizem DMIs de uma maneira mais rápida e menos propensa a introdução de falhas no mesmo. Durante os estudos realizados, verificamos que o IDE Eclipse poderia ser utilizado de uma maneira que resolvesse os problemas encontrados, conforme já mencionado.

O IDE Eclipse já foi utilizado para auxiliar no desenvolvimento de diversos tipos de aplicações, por exemplo, programação colaborativa, apoio a programação e auxilio na utilização de fala, entre outros. Entretanto, até onde sabemos, não encontramos um apoio para desenvolvimento de aplicações que utilizem mecanismos de tolerância a falhas. Uma grande variedade de *plug-ins* é disponibilizado em [46].

Desta forma desenvolvemos um novo (*plug-in*) para o Eclipse que facilite o desenvolvimento de aplicações que utilizem DMIs. Conforme apresentado neste trabalho, o desenvolvimento de uma aplicação que utilize nosso *plug-in* tem seu tempo reduzido além de ser menos suscetível a introdução de falhas.

Como sugestão de trabalhos futuros, podemos indicar a extensão do *plug-in* para o desenvolvimento de modificações de DMIs que possibilitam a construção de DMIs compostas (não foi estudado nesta dissertação de mestrado). Além disto, poder-se-ia também desenvolver algumas ferramentas de apoio a visualização dos relacionamentos, por exemplo, uma ferramenta que mostrasse as classes ou objetos em diagramas UML.

A Apendice A - Telas do plug-in para geração do estudo de caso.

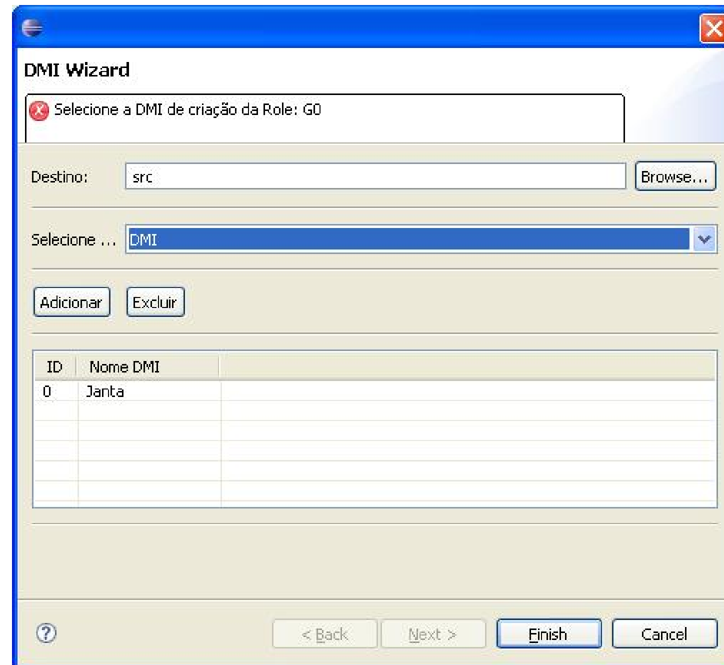

Figura 30 – DMI Janta - engloba todos filósofos e garfos.

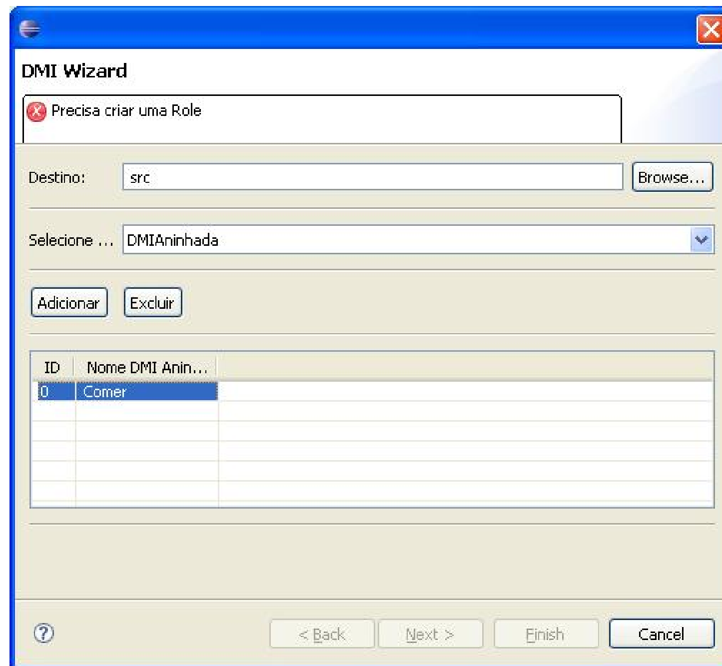

Figura 31 – DMI aninhada Comer - executada por um filósofo e dois garfos.

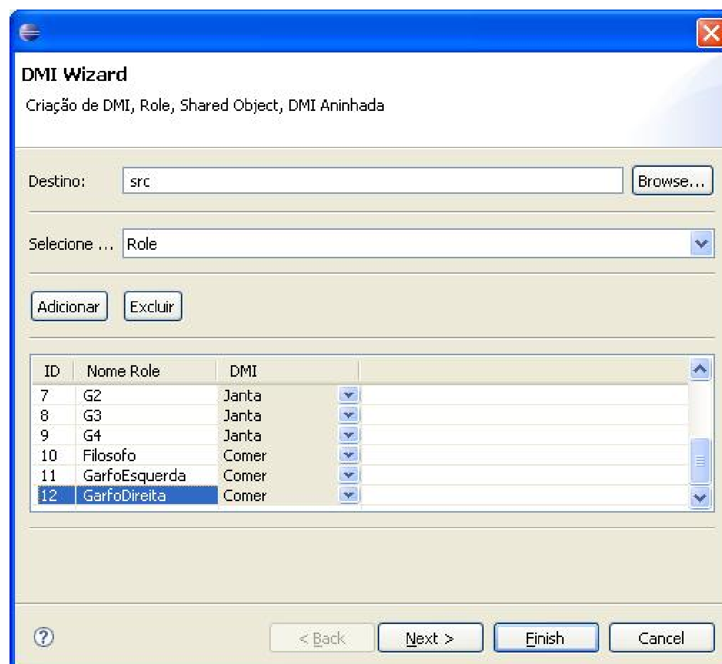

Figura 32 – Criação dos papéis para as DMIs Janta e Comer.

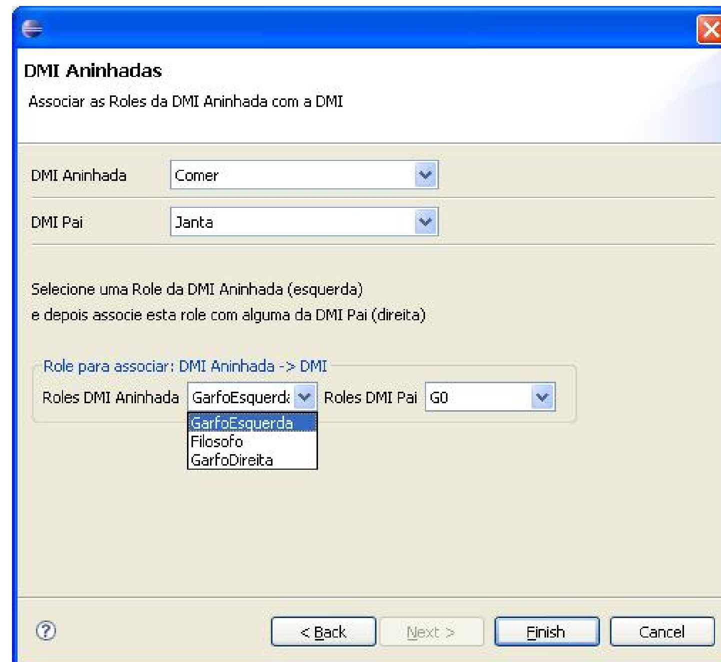

Figura 33 – Associação da DMI Comer como aninhada da DMI Janta.

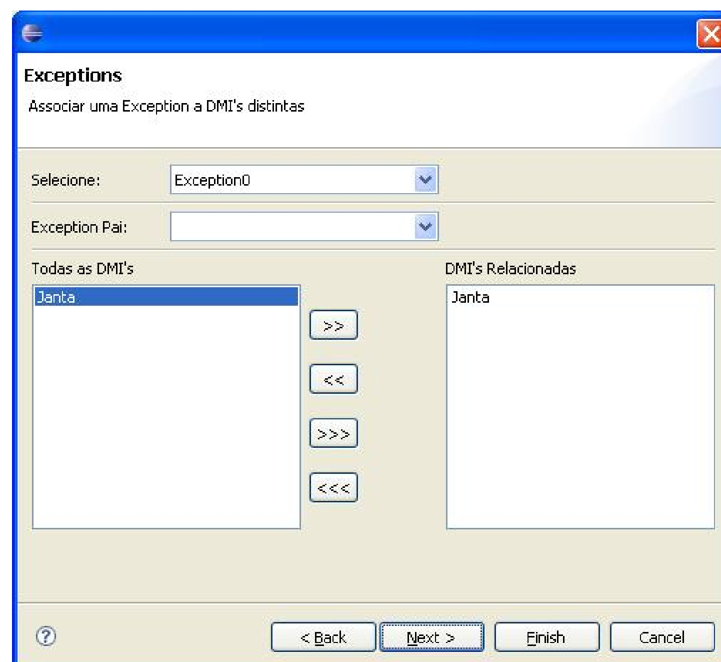

Figura 34 – Criação de um tratador de exceção para DMI Janta.

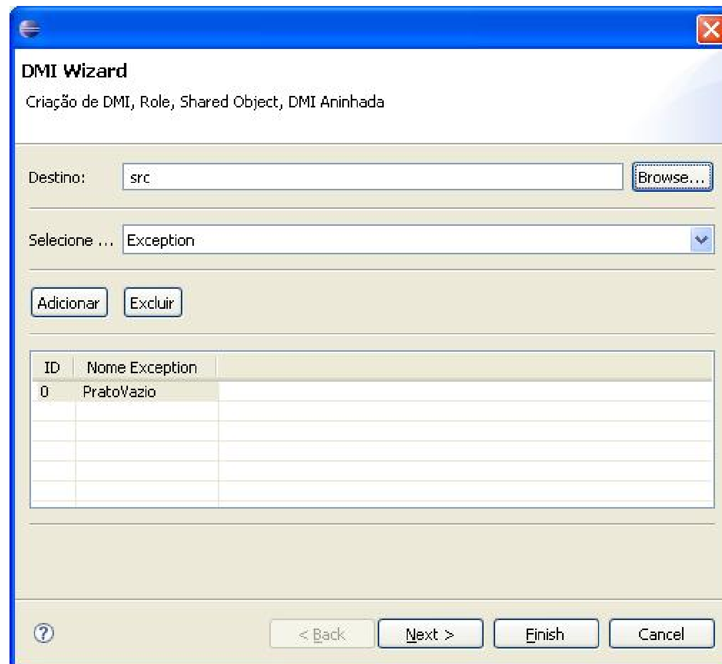

Figura 35 – Tratador de exceção para DMI Janta.

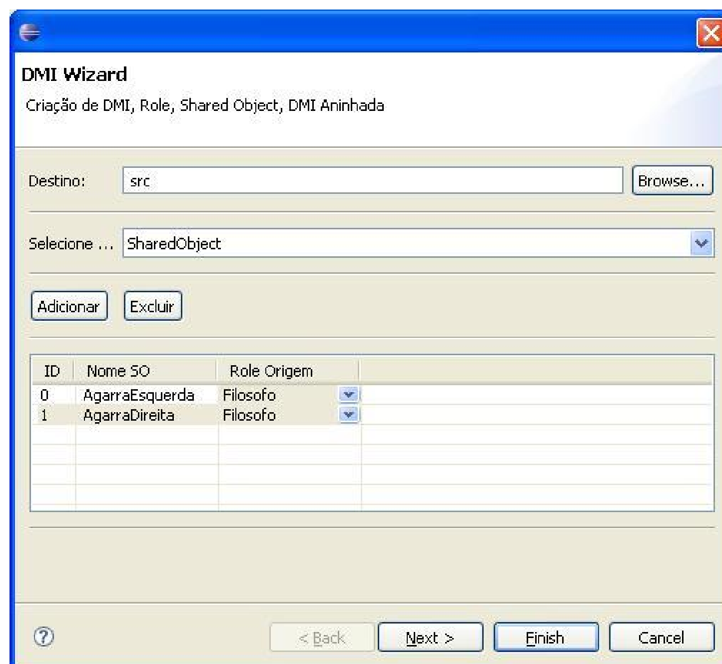

Figura 36 – Criação de objetos compartilhados.

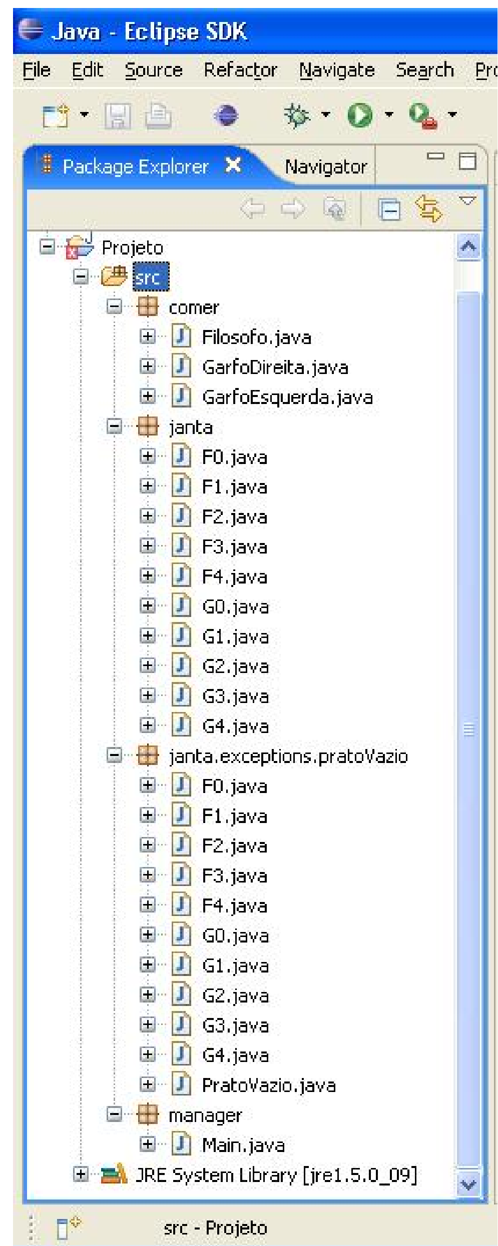

Figura 37 – Estrutura de diretórios e pacotes para DMIs do problema dos filósofos.

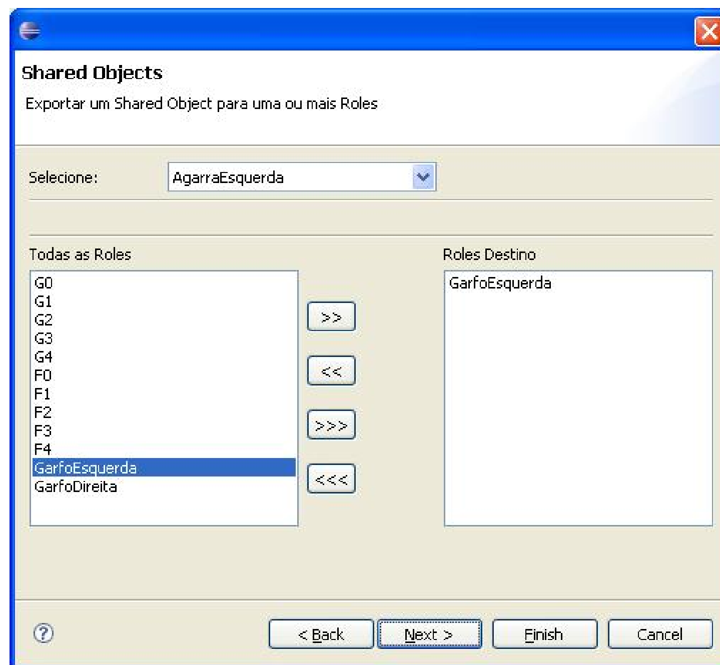

Figura 38 – Definição do papel da DMI Comer que usa objetos compartilhados.

B Apendice B - Código fonte gerado pelo Plug-in

```
package janta;

public class G0 extends RoleImpl {

public G0(String n, Manager mgr, Manager leader) throws
       RemoteException {
super (n, mgr, leader);         // start role with parameters

/** 1º Role da  DMI Normal associada com a Aninhada: "Comer" **/
/** ----------------------------------------------------
 * Comer // - normal >> [GarfoEsquerda, Filosofo, GarfoDireita]
 *---------------------------------------------------- **/

Role garfoEsquerda_Comer;
Role filosofo_Comer;
Role garfoDireita_Comer;

/** ---  Comer  --- **/

// Manager Leader. Parameters: manager name, DMI name
Manager leader_Comer  = new ManagerImpl("leader_Comer", "Comer");

// Create roles: Parameters: role name, role manager, leader manager

garfoEsquerda_Comer =
   new GarfoEsquerda ("GarfoEsquerda_Comer", leader_Comer, leader_Comer);
filosofo_Comer =
   new Filosofo ("Filosofo_Comer",
         new ManagerImpl("leader_Comer","Comer"), leader_Comer);
garfoDireita_Comer =
    new GarfoDireita ("GarfoDireita_Comer",
         new ManagerImpl("leader_Comer","Comer"), leader_Comer);

//exportando outros roles associados ...
mgr.sharedObject("filosofo_Comer", filosofo_Comer);
mgr.sharedObject("garfoDireita_Comer", garfoDireita_Comer);

/** --- Fim código 1º Role ... **/
}

public void body(Object list[]) throws Exception, RemoteException {
try {
garfoEsquerda_Comer.execute();
} catch (Exception e) {
throw e;
}
 }
}
```

Figura 39 – Código gerado o papel G0 - responsável pela criação da DMI Comer.

```
package comer;

public class Filosofo extends RoleImpl {
SharedObject so1_criacao = null;
SharedObject so2_criacao = null;

public Filosofo(String n, Manager mgr, Manager leader)
  throws RemoteException {
super (n, mgr, leader);         // start role with parameters

so1_criacao = new SharedObject (); // create sharedObject
so2_criacao = new SharedObject (); // create sharedObject

mgr.sharedObject ("AgarraEsquerda",so1_criacao) ;   // export sharedObject
mgr.sharedObject ("AgarraDireita",so2_criacao) ;   // export sharedObject

}

public void body(Object list[]) throws Exception, RemoteException {

try {

} catch (Exception e) {
throw e;
}
   }
}
```

Figura 40 – Código gerado para o papel Filósofo da DMI Comer.

```
package comer;

public class GarfoEsquerda extends RoleImpl {
SharedObject dmia_criacao = null;

public GarfoEsquerda(String n, Manager mgr, Manager leader)
  throws RemoteException {
super (n, mgr, leader);         // start role with parameters

}

public void body(Object list[]) throws Exception, RemoteException {

try {

SharedObject so1 = (SharedObject) mgr.getSharedObject("AgarraEsquerda");

} catch (Exception e) {
throw e;
}
   }
}
```

Figura 41 – Código gerado para o papel GarfoEsquerda da DMI Comer.

```
package comer;

public class GarfoDireita extends RoleImpl {

public GarfoDireita(String n, Manager mgr, Manager leader)
  throws RemoteException {
super (n, mgr, leader);         // start role with parameters

}

public void body(Object list[]) throws Exception, RemoteException {

try {

SharedObject so1 = (SharedObject) mgr.getSharedObject("AgarraDireita");

} catch (Exception e) {
throw e;
}
   }
}
```

Figura 42 – Código gerado para o papel GarfoDireita da DMI Comer.

Referências

[1] I. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1996.

[2] I. Forman and F. Nissen. *Interacting Processes - A multiparty approach to coordinated distributed programming*. ACM Publishers, 1996.

[3] B. Randell. Systems structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[4] A. F. Zorzo, A. Romanovsky, B. Randell J. Xu, R. J. Stroud, and I. S. Welch. Using coordinated atomic actions to design safety-critical systems: A production cell case study. *Software: Practice and Experience*, 29(8):677–697, 1999.

[5] Robert J. Stroud Avelino F. Zorzo Ercument Canver Friedrich von Henke Jie Xu, Alexander Romanovsky. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Computer Society*, 2002.

[6] Avelino F. Zorzo, Panayiotis Periorellis, and Alexander Romanovsky. Using co-ordinated atomic actions for building complex web applications: A learning experience. In *WORDS*, pages 288–295, 2003.

[7] Alfredo Capozucca, Nicolas Guelfi, and Patrizio Pelliccione. The fault-tolerant insulin pump therapy. In *RODIN Book*, pages 59–79, 2006.

[8] A. Romanovsky and A. F. Zorzo. Coordinated atomic actions as a technique for implementing distributed GAMMA computation. *Journal of Systems Architecture - Special Issue on New Trends in Programming*, 45(9):79–95, 1999.

[9] Eclipse Web Site. http://www.eclipse.org/. 2007.

[10] JCreator Web Site. http://www.jcreator.com/. 2007.

[11] Jetbrains Web Site. http://www.jetbrains.com/. 2007.

[12] A. F. Zorzo. A language construct for DMIs. In *II Workshop of Tests and Fault Tolerance*, Curitiba, PR, Brazil, 2000.

[13] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.

[14] A. Romanovsky, J. Xu, and B. Randell. Exception handling and resolution in distributed object-oriented systems. In *16th IEEE International Conference on Distributed Computing Systems*, pages 545–552. IEEE Computer Society Press, 1996.

[15] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated atomic actions: from concept to implementation. Technical Report 595, Department of Computing Science, Newcastle upon Tyne, UK, 1997.

[16] A. Romanovsky. Extending conventional languages by distributed/concurrent exception resolution. *Journal of Systems Architecture*, 46(1):79–95, 2000.

[17] A. F. Zorzo. Dependable multiparty interactions: A case study. In *29th Conference on Technology of Object-Oriented Languages and Systems - TOOLS29-Europe*, pages 319–328, Nancy, France, 1999. IEEE Computer Society Press.

[18] C. Lewerentz and T. Lindner, editors. *Formal development of reactive systems: Case study production cell*, volume 891 of *Lecture Notes in Computing Science*. Springer Verlag, Berlin, Germany, 1995.

[19] A. Lötzbeyer and R. Muhlfeld. Task description of a fault-tolerant production cell. Technical report, Forschungszentrum Informatik, Karsruhe, Germany, 1996. http://www.fzi.de/divisions/prost/projects/ korsys/korsys.html.

[20] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Transactions on Computers*, 51(2):164–179, 2002.

[21] Steven J. Vaughan-Nichols. The battle over the universal java ide. *Computer*, 36(4):21–23, 2003.

[22] David Geer. Eclipse becomes the dominant java ide. *Computer*, 38(7):16–18, 2005.

[23] Eclipse Features. http://www.eclipse.org/articles/whitepaper-platform-3.1/eclipse-platform-whitepaper.pdf. 2006.

[24] Elisa Bertino, Silvana Castano, Elena Ferrari, and Marco Mesiti. Protection and administration of xml data sources. *Data Knowl. Eng.*, 43(3):237–260, 2002.

[25] Peter Warren. Teaching programming using scripting languages. *J. Comput. Small Coll.*, 17(2):205–216, 2001.

[26] Dan Rubel. The heart of eclipse. *Queue*, 4(8):36–44, 2006.

[27] Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. *SWT/JFace in Action: GUI Design with Eclipse 3.0 (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[28] Equinox Project Web Site. http://www.sciences.univ-nantes.fr/lina/atl/www/papers/etx2006/19a-stephanhermannrev1.pdf. 2006.

[29] Frank Mueller and Antony L. Hosking. Penumbra: an eclipse plugin for introductory programming. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 65–68, New York, NY, USA, 2003. ACM Press.

[30] Frances Bailie, Glenn Blank, Keitha Murray, and Rathika Rajaravivarma. Java visualization using bluej. *J. Comput. Small Coll.*, 18(3):175–176, 2003.

[31] Margaret-Anne Storey, Daniela Damian, Jeff Michaud, Del Myers, Marcellus Mindel, Daniel German, Mary Sanseverino, and Elizabeth Hargreaves. Improving the usability of eclipse for novice programmers. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 35–39, New York, NY, USA, 2003. ACM Press.

[32] Denys Poshyvanyk, Andrian Marcus, and Yubo Dong. Jiriss - an eclipse plug-in for source code exploration. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 252–255, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. 3d visualization for concept location in source code. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 839–842, New York, NY, USA, 2006. ACM Press.

[34] Martin P. Robillard and Gail C. Murphy. Feat: a tool for locating, describing, and analyzing concerns in source code. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 822–823, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Chih-Wei Ho, Somik Raha, Edward Gehringer, and Laurie Williams. Sangam: a distributed pair programming plug-in for eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 73–77, New York, NY, USA, 2004. ACM Press.

[36] Robert Chatley and Thomas Timbul. Kenyaeclipse: learning to program in eclipse. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 245–248, New York, NY, USA, 2005. ACM Press.

[37] Kenya Web Site. http://www.doc.ic.ac.uk/kenya. 2006.

[38] Charles Reis and Robert Cartwright. A friendly face for eclipse. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 25–29, New York, NY, USA, 2003. ACM Press.

[39] John McKeogh and Chris Exton. Eclipse plug-in to monitor the programmer behaviour. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 93–97, New York, NY, USA, 2004. ACM Press.

[40] Shairaj Shaik, Raymond Corvin, Rajesh Sudarsan, Faizan Javed, Qasim Ijaz, Suman Roychoudhury, Jeff Gray, and Barrett Bryant. Speechclipse: an eclipse speech plug-in. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 84–88, New York, NY, USA, 2003. ACM Press.

[41] Philipp Bouillon and Jens Krinke. Using eclipse in distant teaching of software engineering. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 22–26, New York, NY, USA, 2004. ACM Press.

[42] Li-Te Cheng, Susanne Hupfer, Steven Ross, John Patterson, Bryan Clark, and Cleidson de Souza. Jazz: a collaborative application development environment. In *OOPSLA '03:*

Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 102–103, New York, NY, USA, 2003. ACM Press.

[43] Charles Reis and Robert Cartwright. A friendly face for eclipse. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 25–29, New York, NY, USA, 2003. ACM Press.

[44] Jakarta Eclipse. http://velocity.apache.org/. 2007.

[45] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of ACM*, 8(9):569, 1965.

[46] Eclipse Plugin Central. www.eclipseplugincentral.com. 2007.