UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Soluções Reutilizáveis para a Implementação de Mecanismos de Controle de Atomicidade em Programas Tolerantes a Falhas

por
ACAUAN PEREIRA FERNANDES

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

Profª. Drª. Maria Lúcia Blanck Lisbôa
Orientadora

Porto Alegre, dezembro de 2001

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Fernandes, Acauan Pereira
Soluções Reutilizáveis para a Implementação de Mecanismos de Controle de Atomicidade em Programas Tolerantes a Falhas / por Acauan Pereira Fernandes. – Porto Alegre: PPGC da UFRGS, 2001.
107 p.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2001. Orientadora: Lisbôa, Maria Lúcia Blanck.
1. Atomicidade 2. Tolerância a falhas 3. Reflexão computacional Lisbôa, Maria Lúcia Blanck. II. Título.

Agradecimentos

À minha orientadora, professora doutora Maria Lúcia Blanck Lisbôa, por ter sempre me apoiado durante todo este caminho, não apenas com todo o conhecimento e experiência transmitidos, mas com o exemplo de dedicação e seriedade,

Aos professores e colegas que, em incontáveis ocasiões, contribuíram para este trabalho,

À URCAMP - Universidade da Região da Campanha, que possibilitou a realização deste mestrado,

Aos meus pais, Carlos e Tânia, e minha irmã, Ana Eliza, fontes inesgotáveis de inspiração e carinho,

À minha esposa, Gláucia, pela compreensão e apoio em todos os momentos ...

agradeço.

Dedico este trabalho à memória de meus avós. Sua lembrança me acalenta em tempos difíceis.

Sumário

1 Introdução 12

1.1 Estrutura da dissertação 12
1.2 Mecanismos de gerenciamento de atomicidade 13
1.3 Falhas causadas pela ausência de atomicidade 14

2 Atomicidade, transações e ações atômicas 16

2.1 Transações atômicas 16
2.2 Propriedades ACID: Técnicas de Implementação 17
2.2.1 Conversações 17
2.2.2 Ações atômicas coordenadas 18
2.2.3 Ações atômicas colorizadas 18
2.3 Controle de atomicidade 19
2.3.1 O aspecto da concorrência 19
2.3.2 Técnicas de controle de concorrência 19
2.3.3 Bloqueio em duas fases 20
2.3.4 Timestamps 21
2.3.5 Controle de concorrência otimista 21
2.3.6 Outras técnicas 22
2.3.7 O aspecto da recuperação 22
2.4 Resumo do capítulo 23

3 Emprego de reflexão computacional na introdução de requisitos não funcionais 24

3.1 Apresentação 24
3.2 Reutilização de suporte a requisitos não funcionais 25
3.3 Uso da arquitetura reflexiva para implementação de atomicidade 25
3.4 Reflexão computacional associada a orientação a objetos 25
3.5 Objetivos do gerenciamento reflexivo de atomicidade 27
3.6 Utilização das características do protocolo JavaCore 28
3.6.1 Introdução 28
3.6.2 Características 28
3.6.3 JavaCore e atomicidade 28
3.6.4 Problemas 29
3.7 Aplicação de características do protocolo Guaraná na introdução de atomicidade 29
3.7.1 Introdução 29
3.7.2 Características do Guaraná utilizadas na implementação de atomicidade 31
3.7.3 Utilização de reflexão para salvamento de estados de objetos atômicos 32
3.7.4 Flexibilização via introspecção de métodos/atributos 35
3.7.5 Implementação da técnica de histórico de objetos 35
3.7.6 Recuperação de estados de objetos atômicos 36
3.7.7 Interceptação de mensagens 37
3.7.8 Arquitetura do mecanismo de interceptação de mensagens 39
3.7.9 Interceptação com substituição de métodos ou alteração de resultados 40
3.7.10 Problemas encontrados e sugestões 43
3.8 Outros protocolos de meta-objetos 44
3.9 Resumo do capítulo 45

4 Soluções reflexivas no modelo orientado a objetos 46

4.1 Introdução 46
4.2 Simbiose de conceitos 46
4.3 Modularização, encapsulamento, reutilização e manutenção 47
4.4 Orientação a objetos, reflexão computacional e atomicidade 47
4.4.1 Dados atômicos 47
4.4.2 Objetos como dados atômicos 49
4.4.3 Implementação explícita com uso de herança 49
4.4.4 Implementação implícita com uso de meta-objetos 49
4.4.5 Comparação 50
4.5 Resumo do capítulo 51

5 Utilização de padrões de software para construção de modelos atômicos 52

5.1 Introdução 52
5.2 Padrões e reutilização 52
5.3 Modelos de padrões 52
5.4 Apresentação e organização de padrões 53
5.5 Padrões e *frameworks* 53
5.6 Implementação de um padrão de recuperação de estados utilizando técnicas reflexivas 54
5.6.1 Introdução 54
5.6.2 Descrição do cenário 54
5.6.3 O padrão de recuperação de estados 55
5.6.4 Implementação reflexiva do padrão 55
5.6.5 Estrutura da meta-classe 57
5.6.6 A interface de MC 57
5.6.7 Gravação de estados dos objetos 58
5.6.8 Composição de meta-classes 60
5.6.9 Contribuição da classe e sugestões 60
5.6.10 Reutilização das técnicas empregadas 61
5.6.11 Conclusões 61
5.7 O Padrão Reflexão 62
5.8 Padrões e reflexão computacional 62

6 Cenários de implementações de atomicidade 63

6.1 Introdução 63
6.2 Objetivos 63
6.3 Cenário 1 - Aplicações concorrentes sem acesso competitivo aos dados 64
6.3.1 Descrição do cenário 64
6.3.2 Utilização de thread monitora 65
6.3.3 Threads com término explícito 68
6.3.4 Implementação com thread monitora 68
6.3.5 Implementação com sinalização de término de thread 70
6.3.6 Comparação 71
6.3.7 Reutilização das técnicas empregadas 71
6.3.8 Conclusões 71
6.4 Cenário 2 - Aplicações concorrentes com acesso competitivo aos dados 72
6.4.1 Descrição do cenário 72

6.4.2 Soluções 72
6.4.3 Transações atômicas com sincronização 73
6.4.4 Transações atômicas com guardas 74
6.4.5 Descrição das implementações 75
6.4.6 Implementação 78
6.4.7 Comparação 80
6.4.8 Reutilização das técnicas empregadas 81
6.4.9 Conclusões 81
6.5 Cenário 3 - Meta-Classe Serializadora 81
6.5.1 Introdução 81
6.5.2 Descrição do cenário 82
6.5.3 Arquitetura da classe 82
6.5.4 Comparação 83
6.5.5 Outras utilizações 83
6.5.6 Reutilização das técnicas empregadas 83
6.6 Cenário 4 – Grupos de Meta-Objetos Dinamicamente Reconfiguráveis 84
6.6.1 Introdução 84
6.6.2 Reflexão Computacional e Reconfiguração Dinâmica 84
6.6.3 Um Middleware reflexivo 85
6.6.4 Agrupando Meta-Objetos 86
6.6.5 A Implementação 87
6.6.6 A Visão do Meta-Nível 88
6.6.7 A Arquitetura do *Middleware* 88
6.6.8 Conclusões 90
6.6.9 Reutilização das técnicas empregadas 90
6.6.10 Conclusões 91
6.7 Resumo do capítulo 91

7 Trabalhos correlatos 92

7.1 Introdução 92
7.2 Kan 92
7.2.1 Apresentação 92
7.2.2 Comparação e conclusões 93
7.3 Arjuna 93
7.3.1 Apresentação 93
7.3.2 Comparação e conclusões 94
7.4 Arcabouço atômico orientados a objetos 94
7.4.1 Apresentação 94
7.4.2 Comparação e conclusões 95
7.5 Resumo do capítulo 95

8 Conclusões 96

8.1 Utilização dos paradigmas neste trabalho 96
8.1.1 Padrões de software 96
8.1.2 Reflexão computacional 96
8.1.3 Orientação a objetos 97
8.2 Associações 97
8.3 Considerações finais 98

Bibliografia 101

Lista de Figuras

FIGURA 2.1 – Atualização perdida 14
FIGURA 2.2 – Recuperação inconsistente 14
FIGURA 2.3 – Gravações prematuras 14
FIGURA 2.4 – Leitura prematura 15
FIGURA 3.1 - Requisitos não funcionais no meta-nível (adaptado de [LIS 97a]) 27
FIGURA 3.2 – Instanciação de OperationFactory 30
FIGURA 3.3 – Associação de objetos entre os níveis 32
FIGURA 3.4 – Evitando recursão infinita com Guaraná 33
FIGURA 3.5 – Acesso ao nível base sem recursão infinita 33
FIGURA 3.6 – Reificação de instâncias de classes desconhecidas a priori 34
FIGURA 3.7 – Gravação do estado do objeto em disco 35
FIGURA 3.8 – Operações de leitura no meta-nível 36
FIGURA 3.9 - Restauração de estados de objetos com operações criadas no meta-nível 36
FIGURA 3.10 – Restauração de estados via serialização 37
FIGURA 3.11 – Interceptação de mensagens 37
FIGURA 3.12 - Possibilidade 1. Devolver a execução ao nível base 38
FIGURA 3.13 - Possibilidade 2. Retornar execução ao nível base e testar o resultado 38
FIGURA 3.14 - Possibilidade 3. Substituir a execução no nível base por outra no meta-nível 38
FIGURA 3.15 - Arquitetura de interceptação de mensagens do MOP Guaraná 40
FIGURA 3.16 – Interceptação de um método e substituição de seu resultado 41
FIGURA 3.17 – Substituição da execução de um método por outro 42
FIGURA 4.1 – Interface da classe AtomicMetaObject 48
FIGURA 4.2 – Framework reflexivo para implementação de transações atômicas [FER 00] 51
FIGURA 5.1 – Transição de estados do objeto 54
FIGURA 5.2 - O padrão de recuperação de estados [SIL 96a] 55
FIGURA 5.3 – Implementação da meta-classe 56
FIGURA 5.4 – Método construtor da meta-classe 57
FIGURA 5.5 – Atributos da meta-classe MC 58
FIGURA 5.6 – Reificação de estados de objetos desconhecidos 58
FIGURA 5.7 – Restauração de estados de objetos desconhecidos 58
FIGURA 5.8 – Exemplo de utilização da interface de MC 59
FIGURA 5.9 – Testando primeira chamada a métodos 60
FIGURA 6.1 – Arquitetura do cenário 64
FIGURA 6.2 - Arquitetura da solução 1 66
FIGURA 6.3 – Utilização de interceptação de mensagens para prospecção de threads. 67
FIGURA 6.4 – Primeiro método handle da meta-classe 67
FIGURA 6.5 – Segundo método handle da meta-classe 67
FIGURA 6.6 - Arquitetura da solução 2 68
FIGURA 6.7 – Exemplo da primeira solução 69
FIGURA 6.8 – Exemplo da segunda solução 70
FIGURA 6.9 – Arquitetura do cenário 72
FIGURA 6.10 – Emprego de métodos sincronizados 73
FIGURA 6.11 – Exemplo de uso de guardas em Kan 74
FIGURA 6.12 – Implementação de guardas com reflexão computacional 75
FIGURA 6.13 – Exemplo de pré e pós-condições em Kan 75

FIGURA 6.14 – Uso de threads com guardas 76
FIGURA 6.15 – Interface da classe MO 79
FIGURA 6.16 – Interface da classe TA (Transação Atômica) 79
FIGURA 6.17 – Interface da classe GerTran (Gerenciador de Transações) 80
FIGURA 6.18 – Condições inicial e final 81
FIGURA 6.19 – Acessos livres a um objeto 82
FIGURA 6.20 – Meta-Classe serializadora 82
FIGURA 6.21 – Ordenação de chamadas 83
FIGURA 6.22 – Grupos de meta-objetos 86
FIGURA 6.23 – Metaconfigurações múltiplas 87
FIGURA 6.24 – Arquitetura do meta-nível 89
FIGURA 6.25 – Redundância ou diversidade de projeto usando um mecanismo de votação 90
FIGURA 7.1 – Uso de guardas e pré-condições em Kan 93
FIGURA 7.2 – Ações atômicas em Arjuna 93
FIGURA 7.3 – Hierarquia de classes Arjuna [SHR 89] 94
FIGURA 7.4 – Arcabouço OO para suporte a atomicidade (adaptado de [TEK 94]) 95

Lista de Tabelas

TABELA 6.1 – Classes da primeira implementação do cenário 1 69
TABELA 6.2 – Classes da segunda implementação do cenário 1 70
TABELA 6.3 – Classes da segunda implementação do cenário 2 77

Resumo

Tolerância a falhas é um dos aspectos mais importantes a serem considerados no desenvolvimento de aplicações, especialmente com a participação cada vez maior de sistemas computacionais em áreas vitais da atividade humana. Dentro deste cenário, um dos fatores a serem considerados na persecução deste objetivo é o gerenciamento de atomicidade. Esta propriedade, por sua vez, apresenta duas vertentes principais: o controle de concorrência e a recuperação de estados. Considerando-se a tolerância a falhas e, particularmente, a atomicidade como requisitos com alto grau de recorrência em aplicações, verifica-se a importância de sua reutilização de forma simples e transparente e do estudo de meios de prover tal capacidade.

O presente trabalho procurou pesquisar e aplicar meios de produzir soluções reutilizáveis para implementação de programas tolerantes a falhas, mais especificamente de técnicas de controle de atomicidade, utilizando vários paradigmas computacionais. Neste intuito, foram pesquisados mecanismos de introdução de atomicidade em aplicações e suas respectivas demandas, para então extrair critérios de análise dos paradigmas a serem utilizados na implementações das soluções. Buscou-se suporte nestes paradigmas às demandas previamente pesquisadas nos mecanismos de gerenciamento de atomicidade e procurou-se chegar a soluções reutilizáveis mantendo simplicidade de uso, possibilidade de alteração dinâmica, transparência, adaptabilidade e velocidade de desenvolvimento.

Devido à existência de uma grande diversidade de situações que requerem diferentes implementações de atomicidade, alguns cenários típicos foram selecionados para aplicação e avaliação das técnicas aqui sugeridas, procurando abranger o maior número possível de possibilidades. Desta maneira, este trabalho comparou situações opostas quanto à concorrência pelos dados, implementando cenários onde ocorrem tanto acesso cooperativo quanto competitivo aos dados. Dentro de cada um dos cenários estudados, buscaram-se situações propícias ao emprego das características dos paradigmas e analisou-se o resultado de sua aplicação quanto aos critérios definidos anteriormente. Várias soluções foram analisadas e comparadas.

Além dos mecanismos de gerenciamento de atomicidade, também foram estudados vários paradigmas que pudessem ser empregados na implementação de soluções com alto grau de reutilização e adaptabilidade. As análises e sugestões posteriores às implementações serviram como substrato para conclusões e sugestões sobre a melhor maneira de empregar tais soluções nos cenários atômicos estudados. Com isso, foi possível relacionar características e capacidades de cada paradigma com a melhor situação de demanda de atomicidade na qual os mesmos são aplicáveis, moldando uma linha de soluções que favoreçam sua reutilização. Um dos objetivos mais importantes do trabalho foi, entretanto, observar o funcionamento conjunto destes paradigmas, estudando como os mesmos podem atuar de forma simbiótica e de que forma os conceitos de um paradigma podem complementar os de outro.

Palavras-Chave: tolerância a falhas, atomicidade, reflexão computacional, reutilização de *software*.

TITLE: "REUSABLE SOLUTIONS FOR IMPLEMENTING ATOMICITY CONTROL MECHANISMS IN FAULT TOLERANT PROGRAMS"

Abstract

Fault tolerance is one of the most important aspects to be considered when it comes to developing applications, especially when regarding the increasing role computational systems have been playing in vital areas of human knowledge. Having this in mind, one of the main factors to be taken into consideration is atomicity management. Such characteristic has two inherent concerns: concurrency control and state recovery. Considering fault tolerance and, particularly, atomicity as highly recurrent requirements in applications, one can notice the importance of its reuse in a simple and transparent way and the study of means to provide such feature.

The present paper aimed to search for and apply means for producing reusable solutions to implement fault tolerant applications, more specifically techniques of atomicity management, using several computational paradigms. In order to achieve such goal, atomicity mechanisms were studied, as well as their respective demands. From this study, criterions to be used to analyze the paradigms were found. The latter ones were analyzed in order to get to conclusions about the viability of using them in the implementation of reusable atomicity solutions, regarding simplicity, possibility of dynamic reconfiguration, transparency, adaptability and development speed.

Due to the great diversity of situations that demand different atomicity implementations, some typical cenarios were picked up to apply and amalyze the studied techniques, looking for covering the widest range of possibilities. To do so, this paper compared opposite situations regarding data concurrency, implementing cenarios where there were cooperative and as well as competitive data access. In each one of the regarded cenarios, possible impacts of the studied paradigms characteristics were tested and analyzed.

Several paradigms were studied and tested. Analysis and suggestions taken from such tests were used to get to conclusions about the best way of using them to implent the desired solutions. It was then possible to relate characteristics and capabilities of each paradigm to cenarios where they would be most suitable. One of the most important goals of this paper was to observe the selected paradigms working together, studying how they can be employed in a simbiotic way, and how their concepts can complement each other.

Keywords: fault tolerance, atomicity, computational reflection, software reuse.

1 Introdução

À medida em que aumenta a participação e conseqüente importância dos sistemas computacionais em todos os ramos do conhecimento humano, torna-se imprescindível que estes executem as tarefas para as quais foram programados de maneira correta. Partindo-se do princípio de que não existem sistemas completamente imunes a falhas, procura-se então produzi-los de forma a embutir nos mesmos mecanismos que os possibilitem detectar e tratar eventuais problemas advindos de uma execução problemática. Sendo os fatores causadores de falhas extremamente numerosos e distintos, vislumbra-se uma situação preocupante na qual a tolerância a falhas passa a ter importância vital no desenvolvimento de sistemas e aplicações.

Um fator primordial a ser considerado quando da garantia de confiabilidade de uma aplicação é o tipo de tratamento que seus dados recebem. Uma execução incompleta da mesma com conseqüente alteração parcial de dados pode ser muito mais maléfica que sua não execução. Por conseguinte, aplicações que não apresentem tratamento para tais eventualidades podem acarretar prejuízos extremamente sérios. Projetá-las com tais características deixa de ser apenas desejável, mas necessário. No domínio de tolerância a falhas, esta preocupação é abordada através de mecanismos de gerenciamento de ações atômicas, que buscam garantir a execução completa de um conjunto de ações correlacionadas ou garantir que, em caso de insucesso, ações parciais sejam desfeitas, sem resíduos. Sendo este um típico requisito não funcional amplamente desejável nos mais diferentes tipos de aplicações, vislumbra-se uma grande oportunidade de emprego de conceitos como reutilização de código, o que, por sua vez, traz à tona preocupações com outros fatores tais como transparência, facilidade de uso e velocidade de desenvolvimento, entre outros. Estes, não sendo o foco central da aplicação, não devem aumentar significativamente o tempo de desenvolvimento e a complexidade da mesma.

Métodos que possibilitem a inserção de aspectos comuns em aplicações de diferentes domínios sem que, concomitantemente, exijam esforço adicional podem facilitar a tarefa de desenvolvimento de sistemas mais confiáveis, têm sido a preocupação de diversos pesquisadores. Há, entretanto, várias demandas e modos de implementação de atomicidade em aplicações tolerantes a falhas. Isto torna a reutilização de soluções para estas situações uma das características mais importantes a serem buscadas durante sua implementação. O número de cenários, contextos e situações onde há demanda por atomicidade é enorme, ao mesmo tempo que as soluções existentes são muitas e algumas vezes específicas a um pequeno número de situações. Ampliar a possibilidade de reutilização deste trabalho e, mais ainda, retirar esta preocupação do desenvolvedor, tornaria o trabalho de introduzir atomicidade em uma aplicação mais simples, rápido e transparente.

Dentro deste contexto de grande número e variedade de cenários onde o gerenciamento de atomicidade faz-se necessário, percebe-se a importância da existência de soluções preexistentes que possam adaptar-se ao maior número possível de situações, ao mesmo tempo em que exijam menor esforço do desenvolvedor no sentido de incorporá-las ao código que lida com seus requisitos funcionais, liberando-o para preocupar-se apenas com estes.

1.1 Estrutura da dissertação

Este trabalho tem como principal objetivo implementar soluções reutilizáveis para gerenciamento de atomicidade utilizando diversos paradigmas. Para isso, inicia mostrando o porquê da importância desses mecanismos no capítulo 1 e estuda-os mais a fundo no capítulo 2, buscando extrair dos mesmos demandas que serão usadas como critério para avaliação das soluções a serem implementadas posteriormente. O resultado deste estudo e estas demandas serviram como parâmetros para aplicação e comparação dos paradigmas utilizados nas

implementações. O capítulo 3 estuda a aplicação de conceitos de reflexão computacional no gerenciamento de atomicidade e suas implicações. Examina vários protocolos de meta-objetos e verifica que características podem ser usadas no intuito deste trabalho. A contribuição do modelo orientado a objetos é motivo de estudo no capítulo 4. Este estudo não se limita apenas às características deste modelo que podem ser utilizadas em prol da reutilização de *software*, algo por demais conhecido. O capítulo 4 busca associar este modelo a outros conceitos no intuito de potencializar suas possibilidades, especialmente quando associado à reflexão computacional. O capítulo 5 avança um pouco mais nesse sentido, incluindo a discussão dos conceitos de padrões de *software*. Esta busca inclui testes com soluções reutilizáveis que unam orientação a objetos, padrões de *software* e reflexão computacional. No capítulo 6 são desenvolvidos diversos cenários onde são aplicadas soluções vislumbradas nos capítulos anteriores. Procurou-se desenvolver situações onde as demandas por atomicidade fossem as mais diferentes possíveis, para que a abrangência do estudo fosse maior e mais significativa. O capítulo 7 analisa outros trabalhos correlatos. A seguir, o capítulo 8 mostra o que foi concluído a partir de todo este estudo e faz sugestões no sentido de melhorar o desempenho dos paradigmas no que tange a atomicidade. Durante o estudo e análise dos paradigmas e das soluções em toda a dissertação, também são feitas sugestões similares onde cabível.

A linguagem de programação utilizada no desenvolvimento dos cenários foi Java, pois a mesma oferece suporte aos diversos paradigmas estudados. A versão da máquina virtual utilizada foi Java Kaffe, executada sobre sistema operacional Linux. Para ampliar o estudo da utilização de reflexão computacional, utilizou-se o protocolo de meta-objetos Guaraná versão 1.6, que estende as capacidades da API nativa desta linguagem.

1.2 Mecanismos de gerenciamento de atomicidade

Existe um grande número de situações onde há necessidade de garantia de atomicidade dos dados de uma aplicação. Cada uma delas, todavia, requer uma solução apropriada a suas demandas específicas. Isso levou a uma quantidade equivalente de mecanismos desenvolvidos na tentativa de suprir tais necessidades [JAL 95][CEL 89][BED 99][ELM 91]. Para que seja possível estudar soluções mais abrangentes e reutilizáveis para o gerenciamento de atomicidade, faz-se mister analisar os mecanismos preexistentes. A partir de conclusões daí obtidas, pode-se estruturar soluções que satisfaçam um número maior de cenários e que possam adaptar-se a mudanças em tempo de execução. Inicialmente foi realizado um estudo de mecanismos de gerenciamento de atomicidade [FER 00], com o objetivo de detectar-se diferentes soluções existentes. Desta forma, foi possível elaborar uma lista de cenários que atingisse a maior quantidade possível de especificidades, buscando aumentar a abrangência deste trabalho.

Aplicações que utilizam um mesmo conjunto de dados são cada vez mais comuns no estágio atual dos sistemas de informação, podendo ser inclusive considerada uma das características atuais dos mesmos. Esta situação é encontrada não apenas em ambientes multi-programados, mas também dentro de uma aplicação isolada que contenha componentes executados de forma concorrente. Unidades concorrentes podem atuar sobre dados compartilhados de forma cooperativa ou competitiva. Na concorrência cooperativa, as unidades têm por objetivo processar de forma incremental o mesmo conjunto de dados. Na concorrência competitiva, as unidades necessitam acesso exclusivo aos dados compartilhados. A exclusão mútua é exigida e pode ser controlada através de bloqueio dos dados compartilhados. Um fator primordial para que seus dados possam ser usados de forma aceitável em termos de correção é a garantia de manutenção de sua consistência e integridade. Várias situações, entretanto, concorrem para que tal não se verifique. No caso de vários programas acessando dados armazenados em um banco de dados, a atomicidade deve ser providenciada pelo sistema de gerenciamento do banco de dados. Devido ao foco deste trabalho ser a garantia de atomicidade sob o ponto de vista de aplicações no modelo de objetos, enfatizaram-se o controle de acesso concorrente a dados e a recuperação de estados de objetos.

Em resumo, o acesso concorrente aos dados conduz a situações críticas que podem levar à perda de consistência e integridade destes. Isto é particularmente perigoso com aplicações que precisam executar na verdade uma série de ações sobre o conjunto de dados, embora estas devam ser vistas externamente como apenas uma. O maior problema é a potencialização de erros que ocorre nestas situações, pois uma eventual falha em uma única operação na verdade deve ser considerada como uma falha em todas as ações, ou seja, nenhuma deve ser finalizada com sucesso. Mais do que apenas detectar tal situação, é necessário corrigi-la, para que o estado final dos dados permaneça válido. O tratamento deste grupo de forma única é uma das preocupações do gerenciamento de atomicidade.

1.3 Falhas causadas pela ausência de atomicidade

Supondo a execução concorrente de dois objetos A e B que compartilham um dado X e sobre este dado ambos realizam uma operação de leitura e uma operação de gravação, as seguintes situações indesejáveis podem ocorrer [DAT 91] :

- ***Atualização perdida:*** mais de um objeto atualizando um mesmo dado, sendo que cada um não toma conhecimento das operações do outro (fig. 2.1):

Objeto A	Objeto B
Ler (x)	
	Ler (x)
Gravar (x+100)	
	Gravar (x+100)

FIGURA 2.1 – Atualização perdida

- *Recuperação inconsistente:* um objeto lê um dado imediatamente antes do mesmo ser atualizado (fig. 2.2):

Objeto A	Objeto B
Ler (x)	
	Ler (x)
Gravar (x+100)	
	Imprimir (x)

FIGURA 2.2 – Recuperação inconsistente

- *Gravações prematuras:* um objeto grava um valor e a seguir aborta, deixando-o inconsistente (fig. 2.3):

Transação A	Transação B
Ler (x)	
Gravar (x+10)	
	Ler (x)
	Gravar (x+20)
	Abortar

FIGURA 2.3 – Gravações prematuras

- *Leitura prematura:* um objeto lê dados atualizados por outro objeto em uma operação não concluída (fig. 2.4):

Objeto A	Objeto B
Ler (x)	
Gravar (x+10)	
	Ler (x)
	Gravar (x+20)
Abortar	

FIGURA 2.4 – Leitura prematura

2 Atomicidade, transações e ações atômicas

Neste capítulo são apresentados os conceitos básicos utilizados no gerenciamento de atomicidade em aplicações tolerantes a falhas. Várias técnicas e diferenças entre conceitos são explicitados. Estes servirão de base para a seleção de cenários e desenvolvimento de soluções reutilizáveis para os mesmos.

2.1 Transações atômicas

O conceito de transação atômica, ou simplesmente transação, tornou-se o principal paradigma no desenvolvimento de sistemas que manipulam dados de forma compartilhada [CAM 95]. Autores como Shrivastava [SHR 90], Little [LIT 97], Wheater [WHE 90] e Parrington [PAR 88] não fazem distinção entre ação e transação atômica, sendo que alguns usam o termo "(trans)ação atômica". Jalote [JAL 95] também realiza esta correspondência, definindo transação como uma ação a ser executada atomicamente no contexto de banco de dados.

Date [DAT 91] define uma transação como uma unidade de trabalho lógica, devendo ser percebida como uma operação única, atômica, permitindo aos usuários concluir que seus programas são executados atomicamente como se não houvesse concorrência ou mesmo falhas. Entre suas propriedades, encontra-se a atomicidade [HAE 83][PRI 96], que garante execução completa ou aborto da mesma. O conceito de transação é importante dentro do contexto de tolerância a falhas, pois pode ser usado para garantir que a consistência dos dados seja preservada mesmo na presença de concorrência e falhas [SHR 93]. Sua aplicação, embora inicialmente encontrada apenas em sistemas específicos de bancos de dados, foi ampliada para outras áreas que manipulam dados, além de sistemas distribuídos tolerantes a falhas [BUZ 95]. Transações atômicas são usadas para tolerar falhas em sistemas concorrentes competitivos [BED 98].

Uma transação deve satisfazer às seguintes condições, conhecidas como propriedades ACID (*Atomicity, Consistency, Isolation, Durability*), de acordo com [HAE 83] e [PRI 96]:

- *Atomicidade*: as operações de uma transação devem ser tratadas como uma única, ou seja, todas são executadas ou abortadas.
- *Consistência*: requer que uma transação seja correta, ou seja, leve os dados entre dois estados consistentes. De acordo com [SIB 99], assegurar a permanência da consistência após uma transação é responsabilidade do programador da aplicação que codifica a transação.
- *Isolamento*: uma transação não pode ler os resultados intermediários de outra. Os resultados intermediários de uma transação são invisíveis. As operações não devem ser intercaladas de forma inconveniente, resultando em um estado inconsistente.
- *Durabilidade*: os resultados de uma transação devem ser persistentes. Uma vez completada a transação com sucesso, todas as atualizações realizadas persistirão, mesmo que ocorra uma falha de sistema após a transação ter sido completada.

Tanenbaum [TAN 92] utiliza os termos seriabilidade e permanência ao invés de isolamento e durabilidade, não citando a consistência. Bernstein et al [BER 87] apenas mencionam seriabilidade e recuperabilidade. Buzato [BUZ 95], Lampson [LAM 79] e Boudol [BOU 90] usam os termos equivalência serial (isolamento), unicidade (atomicidade) e persistência (durabilidade).

A literatura apresenta vários procedimentos e estratégias de implementação de propriedades ACID. Todos, todavia, apresentam intrínsecas duas vertentes: o controle de concorrência, para que um grupo de operações sobre certo conjunto de dados não seja interrompido pela execução de outro grupo, e a recuperação de estados, para que uma interrupção devido a qualquer tipo de falha não viole o princípio de unicidade destas operações, em conformidade com Jalote [JAL 95], que afirma que os principais fatores a serem considerados na garantia de atomicidade são os acessos concorrentes e as falhas. Os protocolos que gerenciam o acesso sincronizado para suportar concorrência são chamados protocolos de controle de concorrência, enquanto que os que asseguram atomicidade em caso de falhas são os protocolos de recuperação [ELM 91]. Protocolos de conclusão atômica são protocolos que garantem a execução atômica em um ambiente distribuído [SAC 97].

2.2 Propriedades ACID: Técnicas de Implementação

Existem várias técnicas que podem ser aplicadas para satisfazer as propriedades ACID, tais como conversações, ações atômicas coordenadas e ações atômicas colorizadas.

2.2.1 Conversações

De acordo com Strigini [STR 91], conversações são um bloco de recuperação com vários processos atuando, cada um com um ponto de verificação inicial. Cada processo só pode terminar (realizar sua conclusão) por consenso. Se pelo menos um falhar, todos os outros devem voltar ao estado do início da conversação. Para tal, não podem comunicar-se com processos externos durante a conversação.

Segundo [BED 99], o emprego do conceito de conversações é a melhor maneira de implantar concorrência cooperativa em um sistema e implementar recuperação coordenada de estados. O fluxo de informações é restrito ao escopo da conversação, evitando assim que eventuais erros sejam disseminados.

O início da conversação é marcado pelo estabelecimento de uma linha de recuperação que servirá como referencial do ponto de partida de uma eventual recuperação de estados dos processos participantes. Ao final, uma linha de teste estabelece um limite no qual cada processo deverá passar pelo teste de aceitação. A conversação só terá sido realizada com sucesso se todos os processos passarem nestes testes. Se algum deles falhar, o procedimento de recuperação é então disparado, restaurando os estados dos processos correntemente às suas entradas na conversação. É visto então que esses processos devem cooperar na detecção de erros. Nenhuma informação pode passar para algum processo não envolvido na conversação.

Através deste mecanismo é possível implementar dois tipos de recuperação: por retrocesso (*backward recovery*) ou por avanço (*forward recovery*) do estado da computação. O primeiro tipo é transparente à aplicação, bastando que todos os participantes da conversação realizem um retrocesso ao último estado consistente de dados (*rollback)*. O segundo baseia-se em mecanismos de tratamento de exceção e pode demandar módulos extras para tal, especialmente se precisar lidar com múltiplas exceções, sendo por isso dependente da aplicação [BED 99].

No presente trabalho, o emprego de conversações pode ser visto e analisado no primeiro cenário atômico implementado no capítulo 6. Duas versões foram desenvolvidas e comparadas. Os resultados da análise de seu uso podem ser vistos na seção correspondente.

2.2.2 Ações atômicas coordenadas

Ações atômicas coordenadas (AAC) são um esquema unificado para coordenar atividades concorrentes e suportar recuperação entre vários componentes interagindo em um ambiente distribuído [ROM 97a]. São uma tentativa de integrar transações atômicas e conversações [BED 98]. Fornecem um arcabouço conceitual para lidar com diferentes tipos de concorrência e obter tolerância a falhas integrando transações e o conceito de conversações. Estas controlam a concorrência cooperativa e recuperação, enquanto que as transações são usadas para manter a consistência dos recursos frente a falhas e concorrência competitiva.

Este tipo de mecanismo permite a implementação de controle de concorrência cooperativo e competitivo. Esta tentativa de integração de transações e conversações busca dominar não apenas um, mas vários tipos de concorrência [BED 99]. Ações atômicas coordenadas possuem propriedades das transações atômicas e conversações.

Ações atômicas coordenadas suportam recuperação para trás através do uso de pontos de recuperação, nos quais estados dos objetos são armazenados, e recuperação para frente através da utilização de tratadores de exceção. Estas exceções podem ser sinalizadas por uma *thread* e recebidas pelas outras quando do término destas, ou de forma preemptiva, utilizando-se de alguma característica da linguagem de programação em uso para interromper todas as *threads* participantes. Quando uma operação falha, esta deve ser tratada localmente. Caso isto não seja possível, a exceção é propagada às outras operações envolvidas na ação atômica. Havendo mais de uma exceção ao mesmo tempo, um mecanismo de resolução de exceções torna-se necessário. Uma solução dentro do modelo de objetos é tipá-las e organizá-las em uma árvore hierárquica [BED 99]. Embora a entrada das operações em uma ação atômica coordenada não necessite ser feita de forma síncrona, sua saída deve ser sincronizada. Para mais detalhes, ver [RAN 98a] , [RAN 98b], [ROM 97b] e [ZOR 99].

2.2.3 Ações atômicas colorizadas

As ações colorizadas (*coloured actions*) são um mecanismo baseado na marcação das transações com "cores" (daí o termo “colorizadas”). Ações com a mesma marca (cor) possuem propriedades similares às das ações atômicas convencionais, mas não necessariamente em relação às com cores diferentes. Cada ação pode, além disso, possuir mais de uma cor [SHR 90].

Sejam duas transações aninhadas, com a interna T1 bloqueando dois conjuntos de dados (X1, X2) e a externa T2 esperando a liberação do conjunto X1. Marcando-se a transação interna com duas cores C1 e C2 (e seus conjuntos de dados X1 e X2 cada um com uma dessas cores respectivamente) e a transação externa com C1, por exemplo, faríamos com que T1, ao terminar, liberasse X2, mas mantivesse X1 bloqueado por T2. Neste caso, apenas para o conjunto de dados X1 as transações são aninhadas. Para X2, são duas transações de mesmo nível, o que aumenta o grau de concorrência. Caso T2 aborte, apenas os efeitos sobre X1 seriam desfeitos.

Sua principal diferença em relação às transações atômicas convencionais ocorre quanto a atomicidade de falha e permanência de dados, pois suas ações são atômicas apenas sobre os objetos acessados que possuam a mesma cor da ação. Também suas implementações de bloqueios são diferentes [SHR 90]. Outra diferença entre elas é que uma ação colorizada pode bloquear diferentes conjuntos de dados, usando para cada conjunto uma de suas cores, o que resulta em tratamentos diferentes para cada um deles individualmente. Outros dois mecanismos (*glued actions* e *top level independent actions)* também são implementados usando este mecanismo de marcação e passagem de controle de bloqueios entre transações [WHE 90].

2.3 Controle de atomicidade

A seguir, são discutidas sucintamente as duas mais importantes demandas de atomicidade em aplicações: a concorrência e a recuperação. Para maiores detalhes, ver [FER 00]. Logo após, estas mesmas demandas são apresentadas no contexto das implementações dos cenários, mostrando-se como as mesmas foram desenvolvidas e que dificuldades foram encontradas.

2.3.1 O aspecto da concorrência

Além da já citada característica de uma transação necessitar ser executada completamente ou abortada, também existe a possibilidade de execução de mais de uma transação ao mesmo tempo. Isto leva a duas demandas: recuperação em caso de impossibilidade de término da mesma e isolamento entre as mesmas, para que não haja interferência de uma sobre o resultado de outra(s). Uma execução de diversas transações sem que uma interfira em outra(s) é chamada serial [PAP 86]. Uma execução é dita serializável se produzir o mesmo efeito nos dados que a execução serial das mesmas transações [BER 87]. Uma maneira simples de descobrir execuções serializáveis é a construção de grafos de serialização. Esses grafos mostram setas entre transações. As setas são as precedências de execução. Uma execução é serializável se o grafo for acíclico [BER 87].

O problema na execução serial de transações é a eliminação da concorrência o que, por sua vez, diminuiria o desempenho das mesmas. Se duas transações não acessam os mesmos dados, sua execução concorrente não acarretaria na violação de integridade ou de consistência dos dados. Segundo Lampson [LAM 79], duas transações podem ser executadas concorrentemente se a entrada da segunda for disjunta da saída da primeira. Uma solução inicial seria permitir a execução concorrente de transações que não acessem os mesmos dados e realizar execução serial caso contrário. Esta alternativa não é plenamente satisfatória, pois nem sempre é possível descobrir os dados a serem acessados por uma transação, a não ser dinamicamente. Além disso, o acesso aos mesmos dados pode ocorrer em diminutas frações de tempo, o que não justificaria a serialização das transações. O problema da interferência entre transações pode ser contornado através do uso de algoritmos de controle de concorrência. Para realizar este controle, é útil transformar cada uma das operações sobre os dados sendo realizadas simultaneamente em acessos atômicos. [STR 95a].

Duas operações são conflitantes se tentarem ler/escrever nos mesmos dados. O único caso aceitável ocorre quando as duas operações são de leitura [CEL 89]. Shrivastava et al [SHR 88] descrevem programas que possuem esta propriedade como atômicos quanto à concorrência e chamam as computações que invocam esses programas de ações atômicas.

Protocolos de controle de concorrência têm sido propostos com base na teoria da serialização, como o bloqueio em duas fases [ESW 76], *timestamping* [BER 87] e o controle de concorrência otimista [KUN 81], a seguir brevemente caracterizados.

2.3.2 Técnicas de controle de concorrência

Há vários esquemas para controle de concorrência, porém todos trabalham atrasando uma operação ou abortando a transação que emitiu tal operação [SIB 99]. Entre as técnicas mais utilizadas estão os protocolos de bloqueio, a ordenação através do uso de *timestamps*, técnicas de validação e esquemas de multiversão.

Protocolos de bloqueio são conjuntos de regras que uma transação deve obedecer para bloquear e desbloquear dados. Um destes protocolos é o protocolo de bloqueio em duas fases, que garante serialização mas não está livre de *deadlocks*. É necessário e suficiente quando da

ausência de informações sobre o tipo de acesso a ser feito [SIB 99]. A técnica de *timestamping* associa marcas de tempo às transações e esta ordem determina o modo pelo qual a serialização será executada. O esquema de validação é mais adequado para transações cujas operações são predominantemente de leitura de dados, com conseqüente baixa taxa de conflitos entre elas. Cada transação possui apenas um *timestamp*, que determina sua ordem. Ao final, ela deve passar por um teste de validação para completar-se. Este teste verifica se as alterações que foram realizadas em variáveis locais temporárias podem ser gravadas sem causar violação da serialização. Se não passar pelo teste, deve ser desfeita. O controle de concorrência multiversão baseia-se na criação de novas versões de cada item de dado para cada transação que grava no item. Quando uma operação de leitura é feita, o sistema seleciona uma das versões para efetivar essa leitura. O esquema de controle de concorrência deve garantir que a versão a ser lida seja selecionada de forma a assegurar a serialização por *timestamps* [SIB 99].

2.3.3 Bloqueio em duas fases

Existem vários algoritmos de controle de acesso concorrente a dados que se baseiam no bloqueio dos mesmos. Cada operação de leitura ou escrita deve obter o bloqueio dos dados alvo para assegurar a inacessibilidade dos mesmos a outras operações durante o período no qual estão inconsistentes. Date [DAT 91] define dois tipos de bloqueio (leitura e escrita) e mostra a incompatibilidade entre eles, exceto no caso de dois ou mais bloqueios de leitura, os quais podem ocorrer nos mesmos dados ao mesmo tempo.

O algoritmo de controle de concorrência usando bloqueios mais usado é o de bloqueio em duas fases (*two-phase locking protocol*) [ESW 76]. Esse algoritmo é assim denominado por apresentar duas fases distintas: a fase de crescimento (*growing phase*), na qual todos os bloqueios necessários à operação devem ser obtidos, e a fase de encolhimento (*shrinking phase*), na qual os bloqueios são liberados. Caso a transação não consiga obter algum dos bloqueios necessários durante a primeira fase, é forçada a esperar, o que pode levar a situações de *deadlock*. Há dois procedimentos a adotar quando um bloqueio não pôde ser obtido: desfazer a transação, o que logicamente reduz o desempenho, ou então suspendê-la até que consiga todos os bloqueios necessários, o que pode resultar em *deadlocks*, que são evitados com grafos de espera, de implementação complexa e lenta em ambientes distribuídos, ou por mecanismos de *time-out*, cujo inconveniente é a difícil determinação do período de espera.

Um dos problemas a serem evitados pelas técnicas de controle de concorrência são os abortos em cascata, quando o aborto de uma operação leva a uma seqüência de outros. Para evitar isso, o algoritmo de bloqueio em duas fases libera todos seus bloqueios simultaneamente durante a fase de conclusão da transação. Se os bloqueios forem liberados gradualmente, a transação pode ser abortada após haver liberado alguns dos bloqueios que possuía. Neste caso, esses objetos podem já estar bloqueados por outras transações, e estas transações também deveriam ser forçadas a abortar, pois estariam usando dados inconsistentes [SHR 90].

Quanto ao problema da resolução de *deadlocks*, há três categorias de mecanismos que lidam com este problema [TEK 94]:

- *Time out*: é a maneira mais simples de eliminar possíveis *deadlocks*, levada a cabo através da especificação de um tempo limite máximo de espera, após o qual a transação será abortada. O problema é a possibilidade de aborto de uma transação que não se encontre em *deadlock*, mas apenas sofrendo atrasos de comunicação na rede
- *Prevenção de deadlock*: no momento da solicitação de bloqueio por parte de uma transação é feito um teste para detectar potenciais *deadlocks*. Usam-se *timestamps* que favorecem as transações mais antigas (teoricamente seriam mais trabalhosas de abortar e recomeçar) [CEL 89]

- *Detecção de deadlock*: utiliza o grafo supra citado para detectar *deadlocks*. Exige mais comunicação entre os objetos de dados e nem sempre é fácil de ser executado em ambientes distribuídos

2.3.4 Timestamps

Outro mecanismo de controle de concorrência é o mecanismo de *timestamping*, o qual é considerado não bloqueante. O mecanismo de uso de *timestamps* aumenta o grau de concorrência, pois mais de uma transação pode ser executada ao mesmo tempo, desde que sua ordem não seja violada. Baseia-se no recebimento de números distintos baseados em tempo, chamados *timestamps*, por parte de cada transação. Um gerenciador de transações é responsável pela não repetição desses números. O mecanismo pode então ordenar pedidos de transações de acordo com os números (*timestamps*) destas. Uma transação é abortada se tentar executar uma operação fora de ordem, como por exemplo ler/escrever em um dado que foi alterado por uma outra transação de *timestamp* posterior ao seu. Cada transação abortada, ao ser executada em nova tentativa, receberá outro *timestamp*, desta vez um número maior, o que aumentará suas chances de ser executada com sucesso. Como nenhum dado é bloqueado, não há risco de *deadlock*. De forma resumida, um nodo ao receber duas solicitações conflitantes, executa primeiro a de menor *timestamp*. Se uma solicitação chegar tarde demais (após uma de menor *timestamp*) é rejeitada. Uma crítica que este mecanismo recebe é o fato do mesmo poder gerar um grande número de reinícios, devido a seu favorecimento de transações de curta duração (as mais longas, consequentemente mais antigas, têm menor prioridade). Quanto ao número grande de reinicio, ocorrem quando as aplicações usam os mesmos dados [CAM 95].

Outros mecanismos não bloqueantes utilizam a obtenção tardia e a liberação precoce de bloqueios. O primeiro bloqueia os objetos apenas no momento em que os mesmos são necessários. Isto torna possível a ocorrência de *deadlocks*. Já a liberação precoce de bloqueios procura terminá-los assim que o objeto não seja mais necessário. O problema deste mecanismo é favorecer a ocorrência de abortos em cascata. Esta situação pode ser gerada quando uma ação atômica libera o bloqueio sobre um determinado dado mas é abortada. Ela deve, portanto, acessar esse dado e desfazer suas alterações. O dado, entretanto, pode já estar bloqueado por outra ação. Desta forma, quando for desbloqueado, não apenas a ação atômica que foi abortada deve desfazer suas ações sobre esses dados, mas também deve abortar todas as ações posteriores que tiverem alterado os mesmos, gerando assim um aborto em cascata. Além disto, as outras ações, embora corretas e bem sucedidas, serão abortadas, o que contraria a definição de atomicidade [WHE 90].

2.3.5 Controle de concorrência otimista

O controle de concorrência otimista parte do princípio que os conflitos de acesso a dados ocorrem raramente, já que na maior parte do tempo suas execuções são corretas e não requerem controle de concorrência. Assim, a validação de acesso a dados é feita somente no ponto de conclusão da transação. Neste caso, a checagem é feita realizando-se a intersecção entre os conjuntos de dados a serem lidos e gravados pela transação corrente e as outras transações ativas. As transações são processadas sem preocupação com serialização. Apenas antes da conclusão há checagem de conflitos. Caso estes ocorram, a transação é abortada. O problema maior deste protocolo é a necessidade de manter-se um gráfico de conflitos. Outro problema é a possibilidade de haver um grande número de conflitos e conseqüentes abortos. Embora não haja motivo especial para este algoritmo gerar um grande número de conflitos, sua eventual ocorrência é muito mais danosa que nos anteriores, pois todas as operações da transação já terão sido executadas e terão de ser desfeitas.

2.3.6 Outras técnicas

Algumas metodologias foram propostas na tentativa de aumentar o grau de concorrência entre transações concomitantes. Entre elas encontram-se as ações atômicas aninhadas sem hierarquia (*nested top-level atomic actions)* [LIS 94] que permitem o aninhamento de ações atômicas sem que haja relação entre abortos (ou términos) e manutenção de bloqueios entre elas. Outra técnica são as transações divididas (*split-transactions*) [PU 88], as quais procuram dividir uma transação em outras menores, de acordo com os conjuntos de dados a serem acessados por estas, sendo assim possível executá-las concorrentemente. Sha [SHA 88] apresenta a serialização de conjuntos de dados (*setwise serialisability*), que coloca restrições de atomicidade sobre conjuntos de dados (*atomic data sets*), as quais podem ser satisfeitas independentemente. A estas técnicas juntam-se as já discutidas realização de bloqueios tardios e liberação precoce de bloqueios.

2.3.7 O aspecto da recuperação

A recuperação de estados dos dados permite, ao levá-los a de volta a seus estados originais, garantir sua consistência. Há uma grande número de problemas e falhas que podem ocorrer durante a execução de uma programa, levando este a um término prematuro. Todos os dados que porventura tiverem sido alterados por estes programas não terminados devem retornar a seus valores anteriores. Evitar os estados inconsistentes gerados por tais situações é chamado de problema de recuperação [BER 87].

Há basicamente dois tipos de procedimentos usados na recuperação de dados após uma falha [PRI 96]: o uso de arquivos de registros de ocorrências (arquivos de *log*) ou a criação e gerenciamento de cópias de dados (*shadows*).

O uso de arquivos de *log* é uma técnica que utiliza arquivos para gravar informações necessárias à volta dos dados a seus estados originais em caso de falhas. Estas informações podem ser os valores dos dados anteriores e posteriores à transação, chamadas informações físicas, ou registros das operações realizadas sobre os dados, chamadas informações lógicas. Em caso de falha, é necessário realizar-se as operações inversas correspondentes. Isto diminui o tamanho do arquivo mas, ao mesmo tempo, aumenta a complexidade de recuperação. Outras informações que podem ser incluídas nos arquivos de *log* são listas de transações ativas, terminadas ou abortadas. Com isso, é possível decidir o tipo de recuperação a ser executada em caso de falha. Uma técnica semelhante, denominada sombreamento, grava os novos valores dos dados em uma outra localização da memória estável, ao invés de utilizar os próprios dados. Quando do término da operação com sucesso, tais valores são gravados sobre os dados originais. Uma variação deste mecanismo é o uso de replicação de objetos [LIT 90].

Reed [REE 83] propôs um mecanismo de histórico de objetos. Este não utiliza arquivos de registros, mas versões dos objetos através do tempo. Cada alteração realizada sobre um objeto cria uma nova versão do mesmo. A recuperação é feita através da seleção da versão apropriada do objeto. Cada operação de leitura/gravação recebe uma marca de tempo (chamada de *pseudotime*). A ordem na qual as operações podem ser executadas sobre um objeto é a ordem de seus *pseudotimes*. Cada objeto, ao ser lido ou gravado, recebe um *timestamping*. Se uma tentativa de gravação possuir um *timestamping* menor que o do objeto, é rejeitada. Caso contrário, é criada uma nova versão do objeto.

Tanto o mecanismo de gravação em registro de ocorrências quanto sombreamento são apenas parte do problema recuperar o estado consistente dos dados. Após uma falha, é necessário que este eles voltem a esta condição, o que é realizado através de algoritmo de *redo* (refazer) ou *undo* (desfazer) operações.

Um algoritmo de *redo* deve ser utilizado quando uma transação foi concluída e, antes que pudesse gravar os dados, sofreu uma falha. Neste caso, como a confirmação já havia sido dada, os dados devem ser efetivamente alterados. Já o algoritmo de *undo* deve ser utilizado quando uma transação que não foi concluída alterou dados. Desta forma, como a transação não foi confirmada, suas alterações devem ser desfeitas. Uma combinação desses algoritmos, *redo/undo*, também pode ser empregada [CAM 95]. A maneira pela qual a aplicação grava seus dados (antes ou depois da confirmação) define o algoritmo de recuperação a ser usado.

A execução de *undo* ou *redo* junto com arquivos de registro de ocorrências é feita através da leitura deste. Dependendo do mecanismo a ser usado, diferentes informações devem ser gravadas no arquivo. No caso de sombreamento, para desfazer (*undo*) alguma operação, basta não atualizar os dados, já que nenhuma operação foi efetivamente realizada sobre os dados originais. Para refazer (*redo*), basta gravar de novo os dados alterados sobre os originais.

Uma observação importante a ser feita sobre as técnicas de recuperação é que todas devem necessariamente ser idempotentes, ou seja, seu resultado será sempre o mesmo independentemente do número de vezes que forem executadas. Isto é condição *si ne qua non* para a aplicação de qualquer técnica pois, durante a recuperação dos dados, pode ocorrer nova falha que exigirá seu recomeço.

2.4 Resumo do capítulo

Este capítulo mostrou a importância do gerenciamento de atomicidade em aplicações e alguns dos problemas mais comuns decorrentes de sua ausência. Para que seja possível desenvolver soluções reutilizáveis para a implantação de tal controle, é necessário conhecer as técnicas utilizadas na sua implementação e as principais demandas que as mesmas apresentam, para que possa-se então buscar nos paradigmas a serem estudados formas de suprir tais necessidades. Várias técnicas foram descritas e serão utilizadas como substrato para o desenvolvimento dos capítulos seguintes.

3 Emprego de reflexão computacional na introdução de requisitos não funcionais

3.1 Apresentação

O conceito de reflexão computacional foi primeiramente introduzido por Smith em 1982 [SMI 82] e difundido por Maes [MAE 87]. De acordo com esta, reflexão computacional é um mecanismo que permite a um sistema manipular ou modificar seu comportamento devido à incorporação de estruturas que representam seu próprio estado. Cazzola et al [CAZ 99] caracterizam reflexão computacional como a capacidade de um programa manipular como dados algo que represente seu estado durante sua execução.

Reflexão computacional corresponde à capacidade de um sistema de observar suas próprias estruturas internas e seus respectivos estados correntes, assim como uma representação de sua execução e, a partir deste conhecimento, tomar decisões sobre seu comportamento. Este pode ser alterado em tempo de execução pelo próprio sistema [BLA 2000]. O sistema é conectado causalmente, ou seja, as alterações promovidas sobre sua representação refletem-se sobre seu estado corrente e comportamento, e vice-versa [COU 00]. De acordo com Maes, as estruturas internas e o domínio que representam são ligados de forma que se a alteração de uma delas afeta a outra.

O uso de reflexão pode aumentar a reutilização de uma aplicação, pois esconde os detalhes de implementação do programador. Isto faz com que algumas decisões devam ser tomadas no lugar da aplicação [BLA 99]. Para que isto seja possível, é necessário obter informações sobre a mesma em tempo de execução.

O paradigma reflexivo possui várias capacidades que podem ser exploradas para tornar a introdução de atomicidade em uma aplicação mais simples, rápida, transparente e, principalmente, reutilizável. Embora seus conceitos sejam bem estabelecidos, existem várias implementações de protocolos reflexivos, cada um com diferentes características e graus de suporte a esses conceitos [FER 01b]. Desta forma, é necessário conhecer a abrangência e modo de funcionamento de cada um e compará-los com os objetivos desejados antes de realizar qualquer seleção.

Além do suporte fornecido pela API (*Application Programing Interface*) nativa da linguagem de programação Java, conhecida como JavaCore [SUN 99], há outros protocolos de meta-objetos desenvolvidos para esta linguagem. Estes buscam ocupar uma lacuna deixada por esta API, que não faz distinção entre o meta-nível e o nível base nem suporta interceptação de mensagens.

Um protocolo de meta-objetos (PMO) é uma coleção de métodos em uma meta-interface [BLA 99]. O acesso ao interpretador é fornecido através deste protocolo, que define os serviços disponíveis no meta-nível [COS 99].

Um dos benefícios do uso de reflexão computacional é a diminuição no acoplamento (vinculação entre módulos) e aumento de sua coesão, obtidos pelo uso de interceptação e reificação. A modelagem de requisitos não funcionais através de um protocolo de meta-objetos resulta em um desenho mais flexível.

3.2 Reutilização de suporte a requisitos não funcionais

Aplicações precisam focar não apenas os objetivos para os quais foram desenvolvidas, mas também outras exigências necessárias para que eles possam ser atingidos: os requisitos não funcionais. Estes são assim chamados por adicionarem funcionalidades outras que não as diretamente requeridas pela aplicação [STR95a] e são tipicamente processamento distribuído, persistência de dados, controle de concorrência e tolerância a falhas, entre outros. Por serem comuns a diferentes aplicações, requisitos não funcionais se tornam excelentes candidatos a serem isolados em algum tipo de estrutura ou mecanismo que permita seu reaproveitamento. Um grande empecilho a este procedimento provém da dificuldade em obter-se altos níveis de generalização de código sem emprego de técnicas de introspecção de programas em tempo de execução. Estas técnicas podem não apenas ser providas pela utilização de reflexão computacional, como esta também vai além, permitindo *feedback* dos resultados obtidas após tal introspecção. A combinação das propriedades fornecidas pela reflexão computacional com a possibilidade de adaptação dinâmica de componentes representados como classes no modelo orientado a objetos é um excelente campo de testes com vista à reutilização transparente de requisitos não funcionais a serem implementados como aspectos separados de aplicações específicas. Esta visão de tolerância a falhas torna ideal a separação de aspectos [KIC 97] através de técnicas de implementação baseadas em reflexão.

3.3 Uso da arquitetura reflexiva para implementação de atomicidade

A reflexão computacional é um mecanismo que possibilita o monitoramento da execução do código de uma aplicação. Sua arquitetura apresenta níveis de separação entre o código que executa os requisitos funcionais da aplicação e o que lida com aspectos que devem ser transparentes à esta. Assim sendo, a introdução implícita de requisitos não funcionais como controle de concorrência e recuperação pode ser lograda através da utilização do meta-nível existente nesta arquitetura. No modelo de meta-objetos, mensagens aos objetos no nível base podem ser interceptadas e enviadas de forma transparente ao respectivo meta-objeto. O objeto que a enviou não tem como saber se sua mensagem será processada no meta-nível antes de chegar a seu destino [ANC 95]. A reflexão também permite a total separação entre o código da aplicação e o código da sincronização e recuperação necessário para garantir a atomicidade dos dados. Enquanto que o código da aplicação pode ser implementado normalmente no nível base, o código da sincronização e recuperação é colocado na forma de meta-objetos ou meta-classes no meta-nível. Desta forma, a meta-programação pode modificar o efeito da aplicação no que diz respeito a seu comportamento, possibilitando a adoção de diferentes esquemas de suporte à atomicidade através da introdução de instâncias de novas meta-classes associadas às instâncias das classe da aplicação, sem exigir alteração no código destas [HAE 98].

A reflexão computacional pode auxiliar na execução de uma ação atômica através da adição de características necessárias à recuperação em casos de falhas, levando a computação à situação existente antes do início de sua execução. Isso ocorre através dos mecanismos de monitoração existentes no meta-nível, que permitem disparar ações como conseqüência de eventuais falhas, possibilitando o retorno ao estado original.

3.4 Reflexão computacional associada a orientação a objetos

É reconhecida a importância do modelo orientado a objetos e a reflexão computacional como técnicas promissoras para resolver problemas concernentes à tolerância a falhas [LIS 95b][BUZ 97][WU 97a]. A orientação a objetos facilita o reuso, enquanto que a reflexão computacional aumenta o reuso e a adaptabilidade a diferentes contextos, além de implementar transparência e separação de requisitos não funcionais [COR 96].

Em um ambiente orientado a objetos, a reflexão computacional pôde ser implementada na forma de meta-objetos que representassem informações internas ao sistema. As interfaces desses meta-objetos são chamadas protocolos de meta-objetos (*MOP – Metaobjects protocols*), pois permitem a comunicação entre objetos da aplicação e os meta-objetos [WU 97a]. A combinação dos destes dois paradigmas, ou seja, o uso de um arquitetura reflexiva orientada a objetos no desenvolvimento de especificações não funcionais como tolerância a falhas permitiu a potencialização das vantagens de ambos. Obtêm-se um *software* modularizado e de fácil reutilização. A modularização do software ocorre intra e inter níveis. Dentro do meta-nível, diversas estratégias podem ser implementadas no formato de classes e utilizadas de acordo com as necessidades de cada aplicação, de forma não intrusiva [BUZ 97] [LIS 96].

Wu et al [WU 97a] também realizam esta associação, sugerindo o uso de reflexão computacional através de um protocolo de meta-objetos, o que permite a adição desses requisitos não funcionais mantendo suas características de flexibilidade e customização, o que possibilita a aplicação selecionar os componentes mais apropriados para determinada aplicação e alterá-los caso ocorram mudanças dinâmicas no ambiente. Destacam que requisitos dependentes da aplicação, como processamento distribuído, controle de concorrência e persistência de dados, são especialmente propícios a serem implementados via reflexão computacional.

A orientação a objetos possibilita a uma aplicação suportar atomicidade de modo modular e encapsulado [STR 95a] [TEK 94]. Avançando-se neste conceito, tem-se a utilização de *frameworks* na implementação de mecanismos mais flexíveis de disponibilização de tais propriedades. A reflexão computacional tem sido utilizada não só em conjunto com a orientação a objetos como também na construção de *frameworks* que auxiliam o desenvolvimento de tipos específicos de aplicações, como as tolerantes a falhas [LIS 95a]. A implementação de requisitos não funcionais como meta-classes organizadas em um *framework* torna o sistema flexível, transparente e adaptável, aumenta a reutilização de código e consequentemente a produtividade e ainda permite a inclusão de novas propriedades sem necessidade de alteração no código da aplicação [QUA 99]. Neste contexto, a camada de tolerância a falhas, como sincronismo e recuperação no caso de transações atômicas, fica no meta-nível. Os aspectos funcionais da aplicação são implementados no nível base [LIS 97b] [STR 95a] [KAS 99]. Lisbôa [LIS 98] mostra a utilização da reflexão computacional em especificações não funcionais, especificando entre estas exigências de confiabilidade, segurança e adaptabilidade.

A orientação a objetos permite o desenvolvimento de software a partir de uma ótica na qual objetos cooperam entre si solicitando ou prestando serviços. Desta forma, seu emprego na implementação de atomicidade e outros serviços de tolerância a falhas pode ser feito através do suporte fornecido a partir de objetos fornecedores deste serviço, ao invés dos objetos da aplicação em si [LIS 95a], conforme a figura 3.1. Desenvolvendo este conceito, percebe-se que a reflexão computacional evita que aplicações que necessitem implementar requisitos não funcionais o façam de forma monolítica, criando código de difícil manutenção e pouca reutilização. A estruturação fornecida pela orientação a objetos complementa este cenário, permitindo uma melhor exploração do potencial e das características deste paradigma. Esta separação calcada nos paradigmas em questão é defendida por Stroud et al [STR 95b], que afirmam que a clara separação entre os dois tipos de requisitos permitida pela reflexão computacional torna os programas menos complexos, mais flexíveis e de manutenção mais fácil. Cada tipo de aplicação apresenta seus próprios requisitos não funcionais. Não é possível que um único mecanismo satisfaça a todas as aplicações. De acordo com Wu et al [WU 97a], aplicações têm diferentes requisitos, os quais não podem ser abrangidos pelo paradigma "tamanho único" *(one size fits all)* usado nos *middlewares*, os quais estão obsoletos. Desta forma, os programadores devem adaptar a construção dos requisitos não funcionais às necessidades de seus programas, porém isto seria muito mais complexo se os dois tipos de código estivessem juntos, ou seja, se a aplicação fosse monolítica.

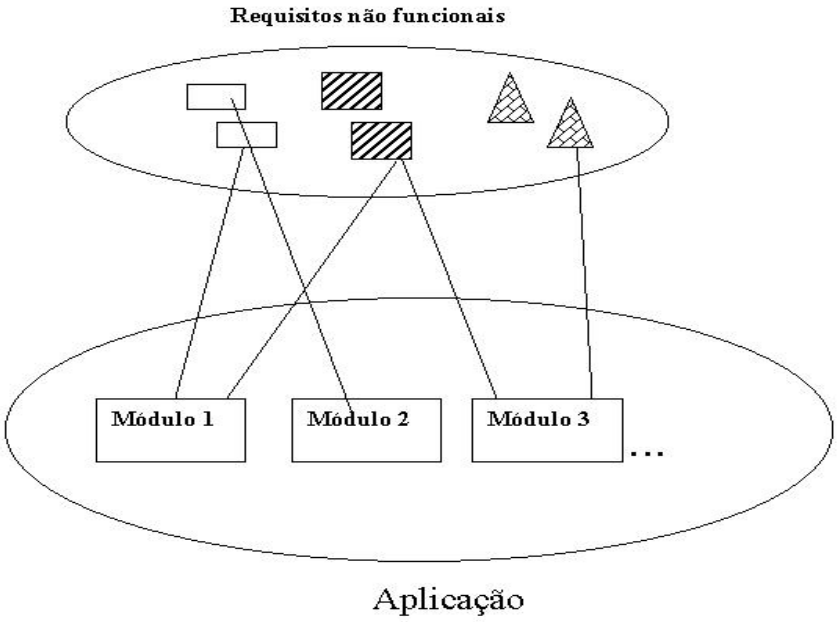

FIGURA 3.1 - Requisitos não funcionais no meta-nível (adaptado de [LIS 97a])

3.5 Objetivos do gerenciamento reflexivo de atomicidade

A associação de reflexão computacional ao modelo de objetos foi realizada de forma a chegar-se a soluções para a introdução de atomicidade em aplicações com as seguintes características:

- Simplicidade: evitar que o desenvolvedor precise se envolver com detalhes dissociados do objetivo de sua aplicação;
- Reutilização: permitir que as mesmas classes reflexivas possam ser utilizadas em várias aplicações sem que para isso seja necessária alteração alguma;
- Velocidade de desenvolvimento: com os requisitos não funcionais já previamente modularizados em classes reflexivas, basta ao desenvolvedor realizar as associações entre suas classes e as meta-classes que implementam os requisitos desejados;
- Facilidade de alteração: caso seja necessário ou conveniente alterar os procedimentos das meta-classes para que estas executem melhor suas funções ou se adaptem a novas realidades, basta otimizá-las sem necessidade de alteração do código da aplicação;
- Modularidade: cada meta-classe é responsável por determinada característica, bastando ao desenvolvedor selecioná-las de acordo com suas necessidades;
- Flexibilidade: como é possível, em tempo de execução, descobrir toda a estrutura de um determinado objeto, pode-se chegar a um código totalmente genérico, que não exija conhecimento prévio da classe do objeto a ser tratado. Desta maneira, o mesmo código pode atuar com qualquer classe. Uma meta-classe que implemente um determinado tipo de recuperação de estados não ficaria vinculada a uma única classe, pois seu código descobriria a estrutura da classe e, a partir daí, executaria a recuperação. Uma única solução seria utilizável para qualquer classe.

3.6 Utilização das características do protocolo JavaCore

3.6.1 Introdução

A linguagem Java padrão implementa mecanismos de reflexão computacional através do protocolo de meta-objetos JavaCore, que utiliza o modelo de meta-classes [BES 99]. Cada vez que novas classes ou interfaces são carregadas, o *Java Runtime Environment (JRE)* cria instâncias da classe *Class*. O mecanismo reflexivo tem sido ampliado desde a versão 1.0, com o aumento de capacidades embutidas na linguagem. A partir da versão 1.2 pode-se acessar qualquer campo de um objeto, inclusive do tipo *private*, devido ao acréscimo de novas classes ao pacote *java.lang.reflect* que corresponde à API reflexiva Java.

O meta-protocolo JavaCore rege a interação inter-níveis, suportando reflexão estrutural e parcial, respeitando os requisitos de segurança da linguagem. Neste protocolo não há distinção entre meta-nível e nível base, o que dificulta a separação de requisitos em diferentes camadas.

3.6.2 Características

Foi implementado no formato de uma API, a *Reflection*, o pacote *java.lang.reflect*. Deve-se notar, todavia, que o acesso desta API aos dados é controlado pelo gerenciador de segurança, de modo que fatores como a localidade da aplicação influenciarão na possibilidade de utilização de todas as capacidades do protocolo.

Há uma instância da meta-classe Class para cada classe carregada durante a execução da aplicação [MCC 97a] [MCC 97b]. Objetos da classe Class, as meta-classes, ficam no meta-nível, associados aos seus objetos referentes no nível base. A API de reflexão em Java é simétrica [MCC 97a], ou seja, a partir de uma classe é possível chegar-se em seus membros e vice-versa. A movimentação acontece em ambos sentidos. Este aspecto da API permite-nos descobrir as interdependências entre uma classe e o resto do sistema.

O objetivo deste trabalho não é estudar as características desta API, apenas seus aspectos que podem facilitar ou dificultar a implementação de atomicidade. Para maiores detalhes sobre o mesmo, ver [SUN 99].

3.6.3 JavaCore e atomicidade

O perfil apresentado na implementação da versão 1.2 deste protocolo não facilita o trabalho de desenvolvimento de aplicações com extensas demandas de transparência. O estudo deste protocolo mostrou que o mesmo fornece apenas capacidades reflexivas básicas, não favorecendo implementações de tarefas mais complexas. Seu uso foi testado em [FER 00], durante o qual foi desenvolvido um protótipo de arcabouço reflexivo baseado neste protocolo. Além de exigir chamadas diretas às meta-classes, não permite a alteração do comportamento de um método, como pode ser feito utilizando-se outros protocolos reflexivos. Com isso, a execução da aplicação é bem menos flexível e pode encampar menor número de possibilidades.

Além disso, associações entre um objeto do nível base e vários meta-objetos devem ser feitas uma a uma, pois não há um mecanismo que, a partir do meta-nível, possa reunir mais de um meta-objeto e associá-los ao nível base, delegando a execução dos métodos de acordo com a localização da característica desejada em cada meta-objeto. Tal possibilidade é importante na simplificação do desenvolvimento de aplicações, pois permite que vários meta-objetos, cada um implementando, por exemplo, um determinado requisito, sejam associados de uma só vez a um único objeto base, fornecendo a este uma série de funcionalidades de acordo com suas

necessidades. Para o desenvolvedor, bastaria selecionar as características que seus objetos necessitassem e fornecê-las aos mesmos de forma simples e rápida.

3.6.4 Problemas

Na criação de aplicações com um grau mínimo de suporte à atomicidade através de reflexão, observou-se , na versão 1.2 algumas dificuldades e problemas a serem contornados com a utilização de outras técnicas complementares:

- Não possui mecanismo de interceptação de mensagens1
- Não pode substituir o comportamento de um método por outro2
- A criação de objetos (método *CreateObject*) de uma classe predefinida qualquer retorna não um objeto desta classe, mas um objeto da classe Object, o que pode exigir o uso de *casting* e dificultar a utilização dos métodos daquela
- Necessita chamada explícita ao meta-nível
- Não possui mecanismo de associação múltipla entre os níveis

3.7 Aplicação de características do protocolo Guaraná na introdução de atomicidade

3.7.1 Introdução

Guaraná é um conjunto de diretrizes, um protocolo de meta-objetos e uma implementação. Busca ser independente de linguagem de programação, desvinculando-o dos aspectos inerentes a alguma linguagem de programação [SEN 01]. O protocolo de meta-objetos (MOP) Guaraná usa uma arquitetura reflexiva que procura ampliar a flexibilidade, segurança e reutilização de código [OLI 98a]. Baseia-se na interceptação de operações e reconfiguração dinâmica de comportamento através de modificação do nível base.

Não é necessário realizar qualquer alteração no modo de programação em Java, pois suas modificações estão embutidas diretamente na máquina virtual. A versão utilizada deste meta-protocolo foi a 1.6, baseada em modificações sobre uma JVM Kaffe de domínio público [OLI 99a][OLI 98c][SEN 01]. A versão 1.7 foi recentemente disponibilizada, porém as alterações embutidas na mesma não são significativas para o objetivo deste trabalho. Embora este protocolo tenha sido projetado para ser independente de linguagem, até o presente momento só existe implementação baseada em Java.

Entre suas principais preocupações está a segurança. Para isso, faz com que as interações entre os níveis reflexivos sejam mediadas por uma entidade independente. Estas entidades constituem o núcleo de execução do Guaraná [SEN 01][OLI 98d]. Uma das formas de atuação destas entidades intermediárias pode ser melhor compreendida através do funcionamento de uma instância da classe *OperationFactory*, responsável pela criação de operações no meta-nível. Um meta-objeto só pode construir operações sobre os objetos do nível base através de uma instância desta classe. Da mesma maneira, a operação não pode ser passada diretamente pelo meta-objeto para o nível base. Aquele precisa utilizar o núcleo do Guaraná para tal. Para garantir maior segurança, o Guaraná não permite que meta-informações sobre componentes

[1] Obsoleto: É possível fazer esta interceptação através de um proxy, a partir da versão 1.3
[2] Obsoleto: O proxy recebe a invocação original e os argumentos de chamada, tornando possível realizar a execução de outors métodos e executar ou não o método originalmente chamado

críticos de seu núcleo seja obtida, assim como interceptações ou alterações sejam realizadas sobre os mesmos [SEN 01].

Dois fatores positivos quanto ao critério reusabilidade são a estrutura hierárquica dos meta-objetos do Guaraná e o desacoplamento entre os níveis reflexivos proporcionado por ele. Como também é possível aplicar mais de um meta-objeto sobre o mesmo objeto-base, este torna-se o somatório daqueles [SEN 01], o que permite um grande número de combinações e recombinações, dinâmicas ou não, ampliando o grau de reusabilidade do protocolo.

O Guaraná fornece suporte a reflexão estrutural e comportamental. Quanto à sua utilização para o propósito deste trabalho, permite isolar os requisitos funcionais dos não funcionais de uma aplicação. Também suporta reflexão em tempo de execução, que apresenta melhores resultados na produção de soluções reutilizáveis, já que meta-protocolos baseados em tempo de compilação, embora mais eficientes em termos de tempo de execução, perdem muito em flexibilidade [SEN 01].

Pode-se obter mais eficiência em meta-protocolos baseados em tempo de execução através de técnicas como *partial evaluation* e *just-in-time compilation*. Alguns pesquisadores sustentam que algumas técnicas de reflexão computacional pode ser usada para diminuir sua sobrecarga [ITO 95]. Proponentes de arquiteturas reflexivas sustentam que os benefícios advindos de implementações mais abertas suplantam o problema da sobrecarga [BLA 99] [COS 2000a]. O método handleOperation permite tratar a informação reificada antes que esta chegue ao nível base. O tratador handleResult permite o tratamento da computação produzida no nível base. Esta volta do fluxo ao meta-nível para posterior tratamento depende, porém, do tratamento da operação no primeiro método handleOperation, ou seja, na interceptação da mensagem. Se uma operação não interessa ao meta-nível, este pode evitar sua volta ao meta-nível. Esta estratégia é importante para diminuir a sobrecarga e aumentar o desempenho da aplicação.

A preocupação com a segurança fez com que limites fossem impostos aos mecanismos deste meta-protocolo. Deve existir uma entidade entre o meta-nível e o nível base responsável pela integridade desta associação. Estas entidades, que devem ser incorruptíveis, estão presentes no núcleo do meta-protocolo. Pôde-se observar a atuação de tais entidades em vários cenários implementados neste trabalho. A única classe que pode criar operações no meta-nível é OperationFactory. Uma instância desta classe é repassada ao meta-objeto durante sua associação com um objeto do nível base (como parâmetro do método Initialize). Caso o objeto não armazene tal instância neste momento, não poderá mais criar operações no meta-nível (fig. 3.2).

```
public Class ... extends MetaObject
{
OperationFactory opf;

void initialize (OperationFactory opf_, Object o)
{
opf = opf_ ;
}
```

FIGURA 3.2 – Instanciação de OperationFactory

Outro fator que corrobora para a segurança deste meta-protocolo é o fato da operação nunca ser entregue diretamente pelo meta-objeto a seu objeto no nível base, mas através do núcleo (*kernel*) do protocolo. Isto evita que uma operação inexistente seja enviada ao objeto base. Também por motivos de segurança, algumas classes críticas deste protocolo, como Operation, Result e Guaraná, não podem sofrer reflexão. Também por motivos de segurança, é impossível vincular um objeto reflexivo a uma nova configuração sem o consentimento desta. Pode-se, desta forma, criar-se meta-configurações que não permitem ser substituídas. Assim como nem tudo pode ser descoberto através do uso de reflexão na linguagem Java, também com

o meta-protocolo esta afirmação é verdadeira. Esta restrição de acesso é importante para garantir a integridade da aplicação e da própria linguagem.

O suporte a reflexão em tempo de execução fornecido pelo Guaraná aumenta seu grau de flexibilidade, o que, por sua vez, aumenta a reutilização de suas implementações. Outros fatores importantes a serem considerados na adoção deste meta-protocolo para as implementações dos cenários foram o alto grau de transparência que pode ser obtido através de seu uso, relacionamento inter-níveis transparente e desacoplamento entre os mesmos (não há como determinar se um objeto está presente em alguma meta-configuração).

Todos as meta-classes construídas em Guaraná devem estender a classe MetaObject. Com isso, herdam alguns métodos importantes, como Initialize e Release, que podem ser confundidos com métodos construtores e destrutores. O método Initialize notifica o meta-objeto de sua vinculação com um objeto do nível base. Este método tem como um dos parâmetros uma instância de OperationFactory que, confirme visto, deve ser armazenada se o meta-objeto tem intenção de criar operações no meta-nível. Quando o objeto base é criado, o meta-objeto é avisado por uma chamada ao método Handle. Quando da desvinculação, o meta-objeto é notificado pelo método Release.

3.7.2 Características do Guaraná utilizadas na implementação de atomicidade

Como o objetivo deste trabalho não foi o estudo de um protocolo de meta-objetos em si, mas sua utilização na introdução de atomicidade em aplicações, apenas características relevantes quanto a este aspecto que foram implementadas e analisadas serão aqui discutidas. Para maiores detalhes, ver [OLI 98b] e [SEN 01].

A entidade central do Guaraná é a classe *MetaObject*. Todos os meta-objetos devem estender esta meta-classe. As classes *Operation* e *Result* respectivamente processam as operações e seus resultados. Operações e resultados no meta-nível são representados por instâncias destas meta-classes. A classe *HashWrapper* atua como uma blindagem no processo de reificação, impedindo recursão infinita durante esse processo. As classes *Composer* e *SequencialComposer* permitem a construção de meta-configurações envolvendo vários meta-objetos, sendo alguns meramente delegadores. Já a classe *Guaraná* é a interface com o núcleo do meta-protocolo, não permitindo a criação de instâncias [SEN 01].

Outra forma de evitar recursão infinita é substituir a operação do nível base por uma criada no meta-nível. Tais operações não estão sujeitas ao “gancho reflexivo”, pois são entregues ao nível base na forma de resultado [SEN 01].

Para que um meta-objeto possa interceptar mensagens enviadas a um ou mais objetos no nível base, é preciso que seja estabelecida uma conexão entre esses dois níveis. O protocolo reflexivo Guaraná permite que um meta-objeto seja associado a um ou mais objetos no nível base. Esta associação deve ser feita de forma explícita, através do método *reconfigure*. Também é possível criar uma matriz de meta-objetos e associá-la a um objeto no nível base. Com isso, este objeto passa a ser monitorado pelo grupo de meta-objetos. Cada um desses elementos pode acrescentar uma característica específica. Isto implica maior grau de modularidade e reutilização. Pode-se associar a um objeto os meta-objetos que apresentem certas características previamente selecionadas. Além disso, novas associações entre objetos e meta-objetos podem ser criadas ou destruídas em tempo de execução.

A associação de mais de um meta-objeto a um mesmo objeto-base ou vice-versa (fig. 3.3) deve ser feita através de uma instância da classe *Composer*, que possibilita o desenvolvimento de meta-configurações múltiplas a partir de um meta-objeto central, instância desta classe, que

delega o objeto-base a outros meta-objetos, estes sim responsáveis pela implementação de alguma tarefa mais específica. Este formato de arquitetura é importante tanto na separação de requisitos como na reutilização de código. Meta objetos independentes e responsáveis por diferentes características ou requisitos podem ser incorporados a um objeto base através de um *Composer*. Ao mesmo tempo, criam-se diversos meta-objetos independentes entre si mas consistentes internamente quanto à sua função, e estabelecem-se conexões com os objetos no nível base de acordo com as demandas destes.

Uma característica importante no que concerne à configuração e reconfiguração entre os níveis é o fato de uma meta-configuração associada a um objeto base ser consultada antes que nova vinculação seja efetivada. A meta-configuração prévia pode aceitar, recusar ou substituir a meta-configuração a ser instalada. Na versão atual do Guaraná, a vinculação é implementada através da adição de um campo oculto em cada objeto. Um valor não nulo nestes campos significa que há uma meta-configuração ativa, pois significará uma referência a um meta-objeto.

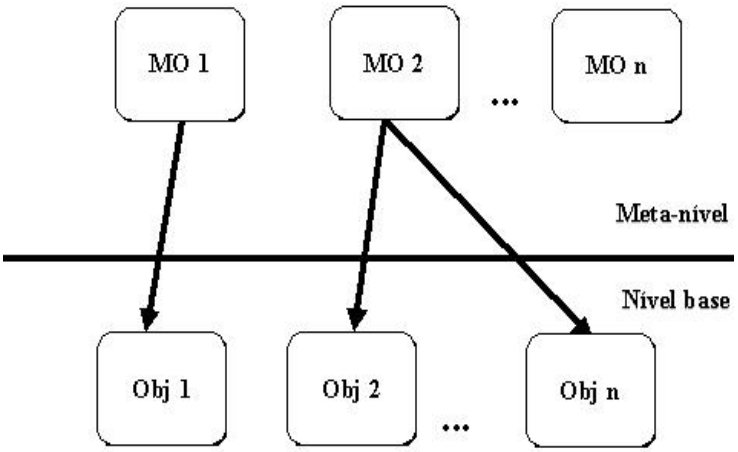

FIGURA 3.3 – Associação de objetos entre os níveis

A seguir, são descritas algumas característica do protocolo que fornecem suporte à introdução de atomicidade em aplicações.

3.7.3 Utilização de reflexão para salvamento de estados de objetos atômicos

Esta foi uma das características exploradas nos cenários atômicos implementados para teste. Uma das grandes vantagens oferecidas pelo uso de reflexão computacional é a possibilidade de interceptação de mensagens para salvamento de estados anteriores à operação iminente sobre um objeto e sua posterior recuperação. Um acesso realizado a partir do meta-nível sobre um objeto no nível base pode levar a um estado de recursão infinita. Conforme visto, há várias maneiras de evitar tal fato. Na versão atual do protocolo Guaraná, uma das maneiras mais simples é criar operações de gravação e leitura no próprio meta-nível através de uma instância da classe responsável por isso, a *OperationFactory*. Estas operações devem então ser submetidas à execução. Para que não causem recursão infinita, todavia, deve-se utilizar uma estrutura como uma *hashTable* e colocar nesta estrutura todas as operações criadas no meta-nível. Assim, quando uma operação é reificada, o meta-nível procurará a mesma na *hashTable* e, caso a encontre, retornará o controle ao nível base, evitando assim a recursão infinita. O enxerto de código a seguir é uma sugestão recebida diretamente do criador do Guaraná, Alexandre Oliva. A arquitetura desta solução está na figura 3.4.

```
public Class ... extends MetaObject{
OperationFactory opf;
HashTable pending = new HashTable( );
```

```
void initialize (OperationFactory opf_, Object o) { opf =
opf_ );

//Obs.: Se o MO não armazenar a instância de
OperationFactory quando do disparo de initialize, //não
poderá mais submeter operações no objeto do nível base,
embora ainda possa ser possível //interceptar e alterar as
operações recebidas pelo mesmo.

Result handle (Operation op, Object o) {
If (pending.containsKey(op))
        return null;
... getField (someField); ...
}

Object getField (Field f) {
Operation op = opf.read(f);
pending.put (op,op);
Object res = op.perform( );
pending.remove(op);
return res;
}
}
```

FIGURA 3.4 – Evitando recursão infinita com Guaraná

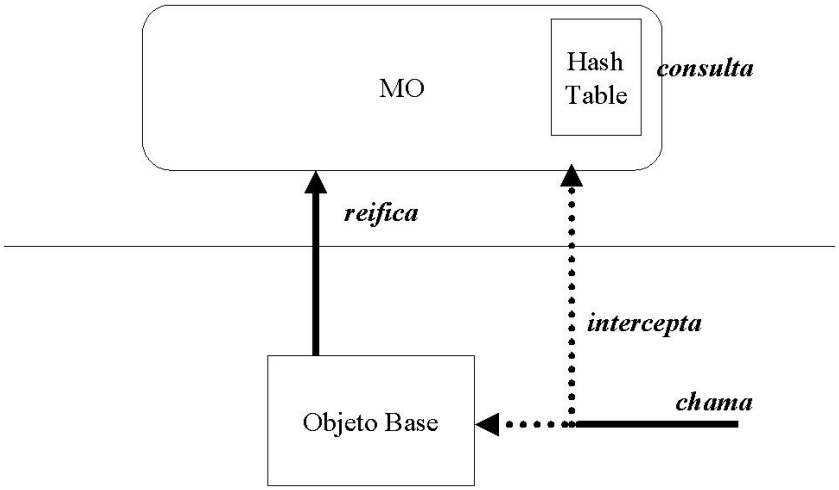

FIGURA 3.5 – Acesso ao nível base sem recursão infinita

Esta solução na verdade não impede a ocorrência de reinterceptação, apenas retorna o fluxo do programa ao meta-nível sem que o mesmo prossiga em situação de *loop*. O método existente no meta-nível que não deseja ser reinterceptado apenas coloca sua marca em uma *hashTable* que será consultada sempre que houver interceptação. A existência de uma marca do método entre seus elementos significa que o mesmo não deve ser interceptado (fig. 3.5), o que é obtido através de um comando *return null*. Este estratagema é utilizado pelos métodos que devem ler ou escrever nos objetos do nível base sem gerar nova interceptacão. No exemplo anterior, o método *getField* procedia desta maneira. Inserindo a si próprio na *hashTable*, impedia que suas operações sobre o objeto base fossem novamente interceptadas, gerando *loop* infinito. Após a execução deste método, entretanto, o mesmo deve excluir-se da *hashTable*, o que foi feito com a instrução “pending.remove(op);” .

Uma outra solução que pode ser utilizada em algumas situações específicas, como em relacionamentos 1:1 entre objetos no nível base e meta-objetos, é momentaneamente desligar a associação entre o meta-objeto e seu objeto base, realizar as alterações sobre este último e, imediatamente a seguir, refazer a associação. Tal procedimento só é recomendável em casos em que o objeto base está associado a um único meta-objeto. Se o meta-objeto for membro de uma meta-configuração composta por vários meta-objetos, estes outros meta-objetos não saberão deste desligamento. Também poderão impedir que este meta-objeto retorne à meta-configuração, ou ainda, no caso de alteração no meta-objeto de nível mais alto na meta-configuração enquanto este meta-objeto está temporariamente desligado da mesma, suas operações serão canceladas e todos os outros meta-objetos dessa mesma meta-configuração reinicializados. Em uma associação com um único meta-objeto, não haveria este problema.

Uma outra maneira oferecida pelo Guaraná é através de mensagens entre meta-objetos. A classe Guaraná possui um método *broadcast* que permite esta comunicação via meta-nível, tornando possível a um meta-objeto avisar aos outros que poderão receber uma mensagem proveniente de seu objeto base. Com isso, é possível descobrir que objeto enviou a mensagem. Nas implementações usadas para teste, como todos os objetos eram subclasses da classe *Thread*, foi possível utilizar outra possibilidade fornecida pelo Guaraná: o método *getThread*, que fornece a *thread* solicitante. Caso este não seja uma *thread*, entretanto, deve-se usar uma das alternativas anteriores, já que este protocolo não utiliza reificação ativa, embora um mecanismo de notificação de saída esteja em estudo pelo autor do protocolo para posterior adição ao mesmo.

Soluções não reflexivas precisam salvar os objetos explicitamente, ou seja, devem ter conhecimento prévio dos dados a serem manipulados pelos programas. Além desta condição reduzir drasticamente o grau de reutilização de código, também elimina a transparência do mesmo, pois é tarefa do programador desenvolver código para tal. Usando reflexão, não apenas os objetos não precisam ser conhecidos *a priori*, como o código fica isolado no meta-nível, não sendo responsabilidade do programador desenvolvê-lo ou alterá-lo.

O procedimento mostrado na figura 3.6 foi usado em uma das implementações teste desenvolvidas. O procedimento recebe um objeto de qualquer classe, descobre seus campos e valores e, neste exemplo, os guarda em uma estrutura interna desenhada para este fim específico.

```
public void reifica (Object origem)
{
Class c = origem.getClass( );
Field[ ] campos = c.getDeclaredFields( );
Object destino[ ] = new Object[campos.length];
for (int i =0; i<campos.length;i++)
{
try{
destino[i] = campos[i].get(origem);
}catch (IllegalAccessException e){ }
}
int pos = achaObjeto(new Integer(origem.hashCode( )));
reserva[pos] = destino;
}
```

FIGURA 3.6 – Reificação de instâncias de classes desconhecidas a priori

Outra forma de evitar recursão infinita entre os dois níveis reflexivos é utilizar a classe *HashWrapper* do Guaraná, que atua como uma espécie de blindagem para seus objetos internos, evitando que sejam interceptados indefinidamente.

Soluções não reflexivas podem usar serialização de objetos como forma de alcançar um maior grau de transparência (fig. 3.7). Devem, porém, resolver onde e como gravar o objeto, além de apresentar menor desempenho por acessar memória em disco. Uma das implementações testemunha desenvolvidas como comparação neste trabalho utilizou o seguinte procedimento não reflexivo para salvar o estado anterior de um objeto:

```
private void salvar_objeto(Thread tr, int i)
{
arquivo = tr.getClass( ).getName( ) + "_"+i;
try{
        FileOutputStream fos = new
FileOutputStream(arquivo);
        ObjectOutputStream s = new
ObjectOutputStream(fos);
        s.writeObject(tr);
        s.flush( );
         }catch(Exception e){}
 }
```

FIGURA 3.7 – Gravação do estado do objeto em disco

3.7.4 Flexibilização via introspecção de métodos/atributos

A introspecção de métodos e atributos é uma das características mais importantes fornecidas pela reflexão computacional em relação à reutilização. A possibilidade de manipular dados cujos tipos não são conhecidos de antemão faz com que seja obtido maior grau reutilização da aplicação. A introspecção de métodos e atributos, assim como seus valores correntes, permite a geração de código genérico que lide com instâncias de classes desconhecidas *a priori*. Para maior detalhes sobre prospecção de métodos, atributos e valores, ver [SUN 99].

3.7.5 Implementação da técnica de histórico de objetos

Algumas técnicas de recuperação de dados utilizam sombreamento ou ainda versões de objetos. É necessário criar cópias dos dados em seus estados anteriores. Para isso, pode-se utilizar mecanismos reflexivos de criação dinâmica de instâncias. Estes mecanismos permitem criar instâncias de classes desconhecidas em tempo de execução, o que torna o código mais genérico e, como conseqüência, mais reutilizável. Em Java é possível executar a carga dinâmica de classes em tempo de execução, utilizando-se o método *newInstance* e a classe *Class:*

```
Class classe  = Class.forName( ... variável string com o nome da classe a criar...);
Object x      = classe.newInstance();
```

Alguns dos cenários implementados neste estudo utilizaram uma solução menos elegante, porém mais simples e rápida: o emprego de uma matriz de objetos da classe Field e um objeto da classe Vector, o qual é preenchido, com o uso de reflexão computacional, com os valores correntes de cada um destes campos. Com isso, é simples implementar a recuperação de dados via versões de objetos. Para criar uma matriz com os campos de uma determinada classe, basta o código a seguir:

```
Field[ ] campos = c.getDeclaredFields();
```

Esta mesma matriz é utilizada tanto para salvar quanto para restaurar os valores nos campos desta classe desconhecida no objeto da classe Vector, sendo usada para salvar valores, como na figura 3.8:

```
valores.removeAllElements();
for (int i =0; i<campos.length;i++)
{
        try{
                Operation op = opf.read(campos[i]);   //
matriz previamente preenchida
        pending.put(op,op);
                Object valor =
op.perform().getObjectValue();
                valores.addElement(valor);
                pending.remove(op);
}catch (IllegalAccessException e){}
}
```

FIGURA 3.8 – Operações de leitura no meta-nível

Uma solução não reflexiva poderia utilizar-se da clonagem de objetos, o que traz muitos inconvenientes, pois nem toda classe é clonável, além deste método não contribuir para simplificar a implementação, pois exige não apenas a implementação da interface *Cloneable* como também da sobrecarga do método *Clone* original e sua chamada de dentro do método criado. Esta solução é bem mais complexa e nem sempre eficaz. Todos os campos de um objeto são copiados bit a bit. Se este objeto possui ponteiros (outros objetos), então tal cópia terá cópias exatas dos campos de ponteiro, fazendo com que o objeto original e o clonado compartilhem os mesmos dados (a clonagem não será uma cópia totalmente nova) [CON 98]. Isto não ocorre quando um novo objeto é criado através do uso de reflexão computacional.

3.7.6 Recuperação de estados de objetos atômicos

Caso seja necessário ativar o mecanismo de recuperação de dados, seja qual for o escolhido para implementação, deve-se restaurar o estado original de cada objeto envolvido na transação que tenha sido modificado. A solução implementada utilizando-se de reflexão é similar ao salvamento de estados de objetos atômicos, apenas executando em sentido oposto. O procedimento reflexivo que realiza tal tarefa recebe um objeto e, procurando-o na estrutura de gravação utilizada pelo método de gravação, retorna seu estado original (fig. 3.9).

```
public void  restaura (Object origem)
{
int pos = achaObjeto(new Integer(origem.hashCode()));
Class c = origem.getClass();
Field[] campo = c.getDeclaredFields();
for (int i = 0 ; i < campo.length; i++)
{
        try{
                try{

campo[i].set(origem,reserva[pos][i]);
                }catch (NoSuchFieldException e){}
        }catch (IllegalAccessException e){}
}
}
```

FIGURA 3.9 - Restauração de estados de objetos com operações criadas no meta-nível

A versão não reflexiva que utiliza serialização precisa antes descobrir que tipo de objeto está restaurando (fig. 3.10) :

```
public void restaurar_estado_original()
{
for (int i=0;i<matriz_local.length;i++)
        restaurar_objeto(matriz_local[i],i);
}


private void restaurar_objeto(Thread tr, int i)
{
String arquivo = tr.getClass().getName()+"_"+ i;

try{
        FileInputStream fos = new
FileInputStream(arquivo);
        ObjectInputStream s = new ObjectInputStream(fos);
        tr = (Thread)s.readObject();
}catch(Exception e){System.out.println (e);}
}
```

FIGURA 3.10 – Restauração de estados via serialização

3.7.7 Interceptação de mensagens

Uma importante característica do Guaraná é sua possibilidade de monitorar o acesso a objetos no nível base (fig. 3.11). Tal característica permite alto grau de transparência em implementações de atomicidade. Este é um dos principais fatores a recomendar sua utilização no intuito de incrementar a reutilização de soluções. Meta objetos associados aos objetos a serem monitorados interceptam todas as mensagens por estes recebidas. Isto permite o posterior tratamento das mensagens, sendo possível então decidir no meta-nível que procedimento realizar. É possível especificar no valor de retorno do primeiro método handleOperation se os resultados produzidos pela computação no nível base devem ser ignorados, inspecionados ou modificados. É possível também simular um resultado, sem que o objeto base esteja ciente sequer da chamada feita a ele. Entre as possibilidades oferecidas por este protocolo estão:

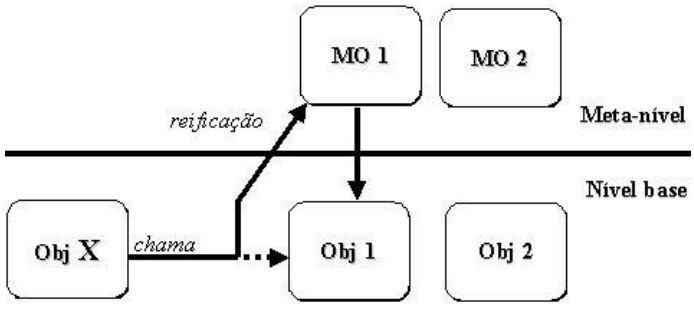

FIGURA 3.11 – Interceptação de mensagens

- ***Retornar o controle da execução do programa do nível base:*** caso o meta-objeto decida não realizar nenhuma operação sobre o objeto base, aquele pode

simplesmente devolver o controle para o objeto no nível base, tornando-se assim transparente ao fluxo do programa (fig. 3.12).

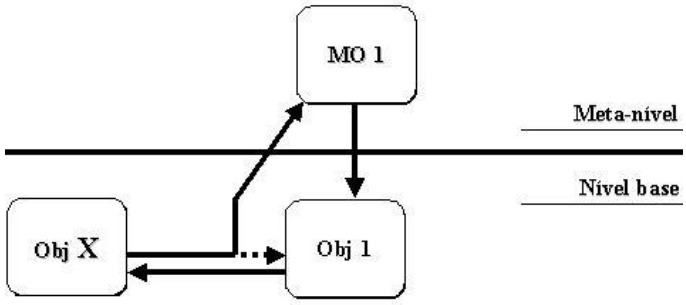

FIGURA 3.12 - Possibilidade 1. Devolver a execução ao nível base

- ***Substituir a execução no nível base por outra no meta-nível:*** se a chamada a um método do objeto localizado no nível base não deve ser executada, esta pode ser substituída por uma chamada a outro método qualquer. O objeto que a enviou receberá o resultado sem saber da substituição. Do mesmo modo, o objeto que contém o método original não saberá desta execução alternativa (fig. 3.13).

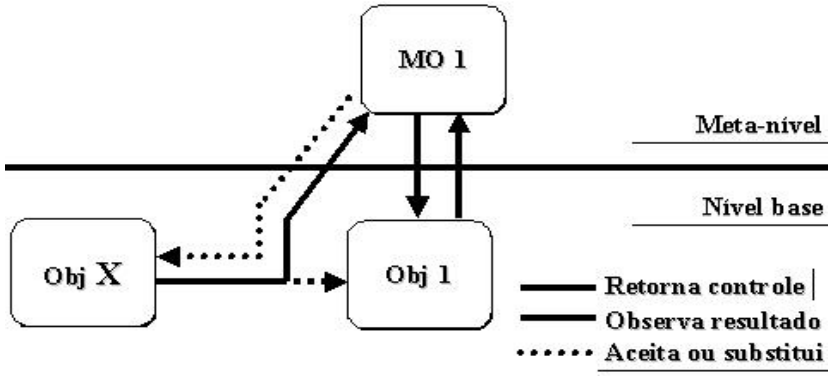

FIGURA 3.13 - Possibilidade 2. Retornar execução ao nível base e testar o resultado

- ***Substituir o resultado da execução do método:*** de forma semelhante à anterior, esta opção permite simplesmente retornar ao objeto solicitante um resultado qualquer, sem executar o método no objeto base. Com isso, nenhuma operação é executada, seja no meta-nível ou no nível base. Todavia, o objeto base que disparou a mensagem receberá sua resposta (fig. 3.14).

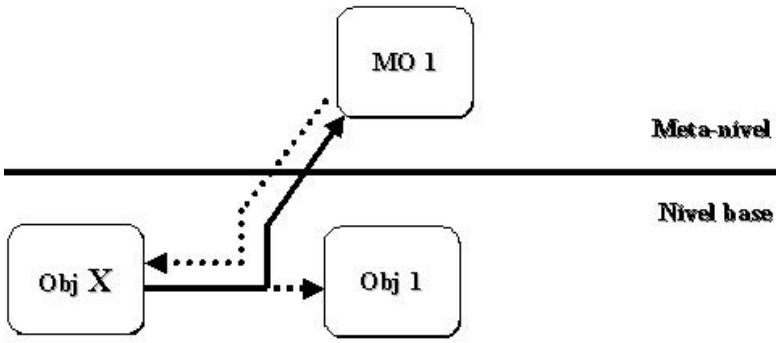

FIGURA 3.14 - Possibilidade 3. Substituir a execução no nível base por outra no meta-nível

- ***Devolver a execução ao nível base e interceptar seu resultado***: uma das características mais poderosas do Guaraná é a possibilidade de retornar o fluxo da

execução ao objeto no nível base, deixá-lo executar o método chamado e, após essa execução, tomar para si novamente o controle do fluxo e, examinando o resultado da execução no nível base, decidir se a mesma deva ser mantida ou substituída. Desta forma, mesmo que o objeto no nível base execute seu método, ainda é possível desfazê-lo. Este mecanismo pode ser visto na descrição da arquitetura da interceptação de mensagens do protocolo Guaraná (fig. 3.15).

Para auxiliar o meta-nível a decidir entre as possibilidades acima, o protocolo Guaraná fornece ao meta-objeto, no momento da interceptação da mensagem, dados como a identificação da mensagem chamada, além do objeto sendo acessado. Esta última informação é necessária em situações onde há mais de um objeto base associado a um meta-objeto, ou quando este precisa realizar operações sobre o objeto base como, por exemplo, salvar os valores correntes de seus atributos antes de permitir a execução da mensagem sobre o mesmo.

A interceptação de mensagens permite o descobrimento do momento exato em que um objeto será acessado. Desta forma, qualquer objeto atômico pode ser associado a um meta-objeto que será responsável pela manutenção de sua atomicidade atuando em conjunto com outros meta-objetos no meta-nível. Todos esses meta-objetos são transparentes à aplicação e ao programador, que não precisa codificar procedimentos para obter tal resultado. Além de permitir alto grau de reutilização, transparência e simplicidade, também permite que vários outros requisitos não funcionais, e até funcionais caso necessário, sejam associados simultaneamente aos objetos do nível base. Supondo-se a existência de vários meta-objetos responsáveis por diferentes requisitos não funcionais, basta selecionar para cada objeto do nível base quais serão úteis ou necessários e associá-los a eles. Devido à existência de classes como *Composer* e *SequencialComposer* que são basicamente classes delegadoras, é possível implementar-se configurações múltiplas de meta-objetos. Além disso, também é possível criar ou eliminar associações inter-níveis em tempo de execução.

Soluções não reflexivas, embora possam utilizar conceitos de orientação a objetos para encapsular módulos responsáveis por requisitos não funcionais, não têm como realizar a comunicação implícita de contextos permitida através do uso da arquitetura reflexiva multi-níveis. A interceptação de mensagens atua como importante mecanismo de garantia de transparência, pois não interfere no código da aplicação, além de muito simplificar o uso pelo programador, pois o mesmo não precisa ficar ciente da implementação do meta-nível.

3.7.8 Arquitetura do mecanismo de interceptação de mensagens

O mecanismo de interceptação de mensagens do protocolo Guaraná (fig. 3.14) funciona através de dois procedimentos de interceptação que ocorrem em momentos distintos: imediatamente antes à chegada da mensagem ao objeto base e imediatamente após sua saída. Desta forma, é possível não apenas desviar sua execução, mas também permiti-la e examiná-la sem que este fato signifique aceitar seus resultados. Existem dois métodos implementando tal mecanismo:

public Result handle (final Operation op, final Object ob)

public Result handle (Result res, Object o)

Os dois procedimentos *handle* acima são disparados respectivamente antes e depois da chegada da mensagem ao método chamado. Dentro dos mesmos é colocado o código a ser executado nestas ocasiões. Percebe-se a existência de classes da hierarquia Guaraná, como *Result*, *Object* e *Operation*. Estas classes especiais realizam funções primordiais dentro deste modelo de reflexão, pois representam no meta-nível, respectivamente, o resultado da operação, o objeto que a sofreu e a operação em si. Um objeto da classe *Operation* permite descobrir informações sobre a operação executada, como seu nome, seus parâmetros e valor de retorno,

entre outras. Operações podem ser reificadas ou criadas no próprio meta-nível através de um objeto da classe *OperationFactory*.

O objeto da classe *Result* retornado pelo primeiro *handle* possui vários métodos que definem o comportamento da segunda interceptação, no segundo evento *handle*. Dependendo de como pretende-se proceder, seleciona-se um destes métodos:

- *inspectResult:* observar mas não modificar o resultado
- *modifyResult:* observar e modificar
- *noResult:* não quer observar

Implementações não reflexivas não têm como identificar o momento em que um objeto atômico é chamado e/ou alterado, portanto todo o desenvolvimento de técnicas de recuperação deve ser codificado como parte da própria aplicação, impedindo a separação de contextos criada com o emprego de reflexão computacional. Esta separação, como pôde ser observado, não apenas simplifica a implementação como também aumenta sua reutilização, transparência e velocidade de desenvolvimento.

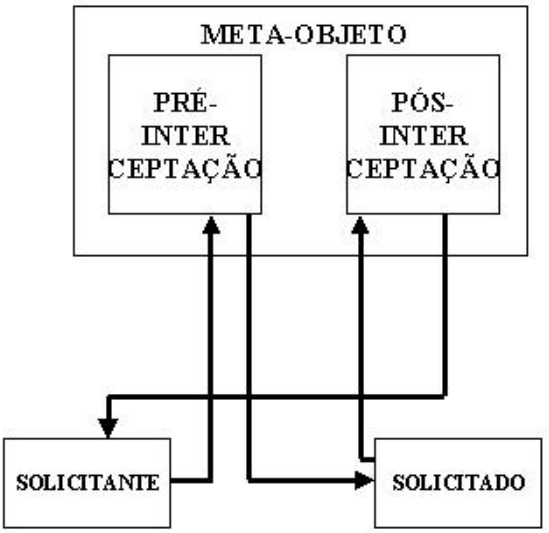

FIGURA 3.15 - Arquitetura de interceptação de mensagens do MOP Guaraná

3.7.9 Interceptação com substituição de métodos ou alteração de resultados

Duas das possibilidades mais poderosas do paradigma reflexivo são implementadas pelo protocolo Guaraná: a substituição de métodos e a alteração de resultados. A substituição de métodos permite que determinados métodos não sejam executados sobre certos objetos do nível base, mas sim substituídos por outros. Da mesma forma, através de reflexão podemos permitir a execução de uma operação sobre um objeto no nível base, interceptar novamente a chamada ao término dessa operação e então decidir sobre a permanência ou não de seus resultados. Esta possibilidade é um pouco diferente da substituição de métodos, pois permite a execução dos mesmos no nível base para só então permitir sua conclusão com sucesso ou alterar o resultado obtido. Esta técnica pode ser usada em casos onde seja necessário manter a consistência dos dados acessados, não permitindo aos mesmos possuir determinados valores. Neste caso, a ação seria desfeita e os resultados retornados a um estado anterior aceitável.

O exemplo da figura 3.16 mostra como executar uma operação no nível base alterando-se a seguir seu resultado. Há duas classes: a classe NivelBase e a classe MetaNivel, cujos nomes já identificam sua localização. A classe NivelBase apresenta um método *strlen* que apenas retorna

o número de letras de uma *string* recebida como parâmetro de entrada. A classe MetaNivel intercepta este método, deixa-o ser executado e, após essa execução, intercepta-o novamente e modifica seu resultado. Para fins de exemplo, o que foi feito foi apenas duplicar o valor encontrado pelo método, de modo que este vai sempre resultar no dobro do número de letras que realmente existem.

```
public class MetaNivel extends MetaObject {

public Result handle(final Operation op, final Object ob) //
método executado antes do método
      // strlen do objeto NivelBase
{
     // testa se o método é o strlen
     if ((op.getMethod( ).toString( )).equals("java.lang.Long
NivelBase.strlen     (java.lang.String)"))
// caso seja, executa-o informando ao segundo evento handle
que deve interceptá-lo      //novamente
    return Result.modifyResult;

   //se não for o método strlen, retorna o controle ao nível
base.
     return Result.noResult;
   }

   public Result handle(final Result res, final Object ob)
// método executado após o método
      // strlen do objeto NivelBase
   {
//se for o método strlen
 if (((res.getOperation( ).getMethod( )).toString(
)).equals("java.lang.Long NivelBase.strlen
(java.lang.String)"))

// retornar o dobro do resultado de sua operação
return Result.returnObject (new
Long(2*((Long)res.getObjectValue( )).longValue( )),
res.getOperation( ));

else
     return null;
    }
}
```

FIGURA 3.16 – Interceptação de um método e substituição de seu resultado

O segundo método *handle* tem parâmetros diferentes do primeiro. Isso ocorre porque este evento ocorre após a execução do método no nível base. Enquanto que o primeiro evento *handle* recebe informações sobre a operação interceptada (no objeto op da classe *Operation*) e o objeto sobre o qual a mesma ocorre (no objeto ob da classe *Object)*, o segundo evento *handle* recebe o resultado da operação (no objeto res da classe *Result*) e o estado final do objeto (no objeto ob da classe *Objec*t). Existem atributos e métodos específicos para cada uma destas classes que podem auxiliar nas decisões a tomar. No exemplo acima usou-se o método *getOperation* sobre o objeto da classe *Operation* para descobrir que método foi invocado sobre o mesmo. Também usou-se o método *returnObject* da classe *Result* para alterar o resultado após a execução do método. As classes do Guaraná, vistas mais adiante em sua hierarquia, fornecem métodos e atributos úteis na definição das ações a serem tomadas pelos meta-objetos.

O exemplo a seguir usará uma classe MOB para ilustrar como substituir a execução de um método qualquer por outro. Esta classe possui dois métodos simples (metodoBase1 e metodoBase2) que apenas imprimem um aviso na tela, além de dois outros métodos (duplica e triplica) que recebem um número e retornam seu dobro ou triplo. A substituição de métodos é ligeiramente diferente caso o método apresente parâmetro, os quais devem ser descobertos em tempo de execução, motivo pelo qual todos esses métodos serão usados no exemplo da figura 3.17:

```
public Result handle(final Operation op, final Object ob)
   {
if (op.isMethodInvocation( ))
              try {
          final java.lang.reflect.Method m = op.getMethod( );

                  // se o método é o “metodoBase1”
                 if ( m.getName( ).equals("metodoBase1") )
                 {
             Operation newOp = opf.invoke(metaMetodo2,
new Object[0],o
                    //retorna seu resultado
                   return Result.operation(newOp,
Result.inspectResultMode);
                  }

          // se o método é o “duplica”
                 if ( m.getName( ).equals("duplica") )
                 {

           // obtém seus parâmetros
                    Object b [ ] = op.getArguments( );

           // executa  “triplica”
          Operation newOp = opf.invoke(metaTriplica, b ,
op);

            // retorna seu resultado
                  return Result.operation(newOp,
Result.inspectResultMode); }

  }catch (IllegalAccessException e) { }
return Result.inspectResult;
   }
```

FIGURA 3.17 – Substituição da execução de um método por outro

As capacidades inseridas na arquitetura de interceptação do MOP Guaraná abrem uma vasta gama de possibilidades de incremento do grau de reutilização de aplicações. Operações podem ser desfeitas mesmo após sua execução caso alguma falha seja detectada. Operações não permitidas em determinados objetos podem ser vetadas em tempo de execução. O objeto base recebe um grau de proteção muito maior do meta-nível e as aplicações não precisam ser alteradas, pois o meta-nível pode cuidar de substituir chamadas a métodos não permitidas a instâncias de certas classes. Com isso, obtêm-se maior reutilização de código.

3.7.10 Problemas encontrados e sugestões

Vários problemas foram encontrados durante o estudo e a aplicação do protocolo Guaraná sobre cenários com demandas de atomicidade. Além da dificuldade gerada pela existência de pouco material disponível sobre seu funcionamento, em algumas situações, tais como a de evitar a laços recursivos infinitos entre os dois níveis da arquitetura recursiva não foi tarefa trivial. O excesso de interceptações também pode levar a sobrecargas prejudiciais ao bom desempenho da aplicação no quesito velocidade.

Durante este trabalho, várias formas de evitar a recursão são descritas, desde as mais simples e utilizáveis apenas em situações particulares ou de menor risco, até o uso de estruturas mais complexas deste protocolo. Esta discussão é complementada com a mostra de um mecanismo criado no meta-nível com o objetivo de evitar recursão infinita em casos onde seja necessário utilizar no próprio meta-nível o objeto do nível base que está sendo interceptado (fig. 3.4). Este fato normalmente levaria a um laço infinito entre os níveis pois cada tentativa de acessar o objeto a partir do meta-nível geraria uma nova interceptação a qual faria o fluxo voltar ao meta-nível e assim sucessivamente *ad infinitum.*

Embora este não pareça ser o caminho a ser adotado nas próximas versões do Guaraná, sugere-se a criação de meios de diminuir esta interceptação forçada de chamadas, desde que bem controlada pelo desenvolvedor do meta-nível. Conforme já foi discutido aqui, um mecanismo de notificação possivelmente venha a ser incorporado brevemente, de acordo com seu autor, o que poderia resolver parcialmente este problema.

A solução mais simples encontrada para evitar o problema da recursão infinita foi o desligamento temporário da conexão entre os níveis:

```
Guarana.reconfigure(dados,null,null);
restaura_dados(para_objeto);
Guarana.reconfigure(dados,null,this);
```

Esta solução é aplicável apenas em casos onde a configuração inter-níveis existente entre os objetos for 1:1. Em outros casos, foi necessário utilizar objetos da classe hashTable.

Outra dificuldade encontrada durante a implementação dos cenários com o uso deste protocolo foi usar os parâmetros passados nos métodos *handle*, após cada interceptação de mensagens. Apenas os métodos Guarana.toString(obj), Guarana.getClass(obj), Guarana.getClassName(obj) e Guarana.hashCode(obj) puderam ser executados sobre os parâmetros recebidos, pois garantem não interceptação pelo Kernel. Qualquer outra operação executada sobre os mesmos resulta em nova chamada a este método.

public Result handle (final Operation op, final Object ob)

public Result handle (Result res, Object o)

O parâmetro op do primeiro método *handle* retorna a operação que foi interceptada. Já o parâmetro ob retorna o objeto alvo de tal operação. Para que tentativas de utilizar tais parâmetros não resultem em novas chamadas a este método, gerando assim nova recursão infinita, foi necessário usar objetos da classe HashWrapper, que serve como uma “blindagem” contra recursão. A outra solução foi o já citado uso de hashTables para “rebater” as reinterceptações no seu início.

Algumas capacidades que poderiam facilitar o uso deste protocolo na tarefa específica de gerenciamento de atomicidade e que tiveram de ser contornadas de alguma outra forma são listadas a seguir:

- Possibilidade de interceptação de mensagens apenas na chamada e na saída do método (atualmente, caso um método possua dez comandos, cada um destes comandos internos gerará uma nova interceptação, além das já citadas interceptações ao início a ao término do mesmo);

- Possibilidade de interceptação de saída (especialmente útil para desenvolvimento de atomicidade, permite que se descubra para qual objeto o objeto monitorado está enviando uma mensagem). A interceptação de saída, também chamada de interceptação ativa, é útil em situações onde objetos atômicos alterem outros objetos e assim sucessivamente. Através de recursividade e interceptação ativa poderia ser possível mapear a hierarquia de chamadas e suportar atomicidade em todos os níveis. De acordo com o autor, por motivos de segurança, o máximo que poderá aparecer em novas versões são mecanismos de notificação;

- Possibilidade de existência de interceptações não reentrantes (interceptações cujas chamadas ao meta-nível não gerassem novas interceptações);

- Maior facilidade de tratamento dos parâmetros de entrada dos métodos Handle dos meta objetos.

3.8 Outros protocolos de meta-objetos

Além dos já citados JavaCore e Guaraná, outros meta-protocolos reflexivos foram pesquisados e analisados no intuito de selecionar o mais apropriado para o desenvolvimento dos cenários. Foram examinadas implementações dos protocolos OpenJava [TAT 99], MetaJava [GOL 97] e Reflective Java [WU 97a]. As principais diferenças entre estas implementações ocorrem não apenas em nível de características e possibilidades do protocolo em si, mas na forma como o qual deve ser utilizado. Tais diferenças quantitativas e qualitativas foram estudadas e analisadas e decidiu-se pela utilização do protocolo Guaraná pois o mesmo, apesar das dificuldades, mostrou maior potencial quando confrontado com os parâmetros definidos como base para introdução e gerenciamento de atomicidade.

A utilização de diferentes meta-protocolos ou protocolos de meta-objetos pode demandar do desenvolvedor diferentes procedimentos. Uma diferença básica ocorre na ativação do meta-nível. Conforme já descrito, uma implementação de suporte a atomicidade foi realizada em [FER 00], porém o PMO utilizado não permitia ativação implícita, o que trouxe severos prejuízos à mesma nos aspectos que estavam sendo utilizados como baliza de implementação. A ativação implícita do meta-nível permite a obtenção de um grau muito maior de transparência, simplicidade de desenvolvimento e manutenção do código de nível base.

Outra forma de emprego de PMO encontrada foi através do uso de diferentes arquivos com código fonte e necessidade de posterior aplicação de pré-processadores, como em [TAT 99]. Além do programa propriamente dito, o desenvolvedor precisa também produzir outro arquivo com informações a respeito do procedimento a ser tomado pelo pré-processador para transformar seu código em um apto a ser então compilado pelo compilador padrão Java. Wu [WU 97a] demonstra a criação de uma meta-classe que implementa o controle de concorrência via bloqueio em duas fases (*two-phase locking*), porém a arquitetura de utilização deste MOP torna o trabalho de utilizá-la mais complexo que as demais. Guaraná permite que o código seja desenvolvido normalmente, como se este fosse parte integrante da linguagem de programação Java, pois baseia-se em alteração na máquina virtual para que esta lhe forneça suporte. Isto facilita bastante o trabalho de planejamento e codificação da aplicação, embora torne-a dependente da existência de versões para a máquina virtual Java em uso.

Um protocolo de meta-objetos pode ser codificado dentro de um compilador ou pré-processador da linguagem alvo, ou enxertado no ambiente e execução. Protocolos com reflexão implícita aumentam a transparência e permite a criação de código no nível base independente do meta-nível. Isto ocorre porque o próprio protocolo se encarrega de monitorar os acessos aos objetos base, independentemente de instruções explícitas do programador. Quanto menor o número de alterações no nível base necessárias para que esta conexão com o meta-nível seja efetivada, maior a transparência da aplicação.

Há protocolos que exigem que arquivos fontes sejam escritos em uma linguagem de vinculação, utilizando um pré-processador, como, por exemplo, Reflective Java [WU 97b]. Outros apresentam um pré-processador de macros, suportando reflexão em tempo de compilação, como, por exemplo, OpenJava [TAT 99].

3.9 Resumo do capítulo

Este capítulo procurou mostrar algumas implementações existentes de protocolos de meta objetos, definindo-se por um para implementar os cenários estudados neste trabalho. Algumas características relacionadas com possibilidades de reutilização, particularmente quanto a atomicidade, foram mostradas e analisadas. Por fim, dificuldades encontradas durante sua utilização foram discutidas, assim como algumas sugestões no sentido de melhorar seu suporte às características aqui enfocadas.

4 Soluções reflexivas no modelo orientado a objetos

4.1 Introdução

O objetivo deste capítulo é estudar a associação do modelo orientado a objetos com conceitos de reflexão computacional, buscando vislumbrar novas possibilidades em sua aplicação conjunta. Os conceitos fundamentais do modelo orientado a objetos já foram foco de detalhados estudos e sua capacidade de aumentar o grau de reutilização de *software* produzido seguindo esta orientação já foi por demais documentada. Um grande problema identificado na literatura diz respeito ao tempo de retorno de tal capacidade em termos de aumento na velocidade de desenvolvimento de *software*. Concorda-se que apenas após a construção de um número mínimo de classes pode-se começar a obter algum retorno em termos de reutilização e conseqüente incremento na velocidade de desenvolvimento de novas aplicações a partir dos "blocos" preexistentes. Pode-se, entretanto, obter reutilização imediata, com o emprego de outros paradigmas neste trabalho.

Este capítulo pretende incluir um nova variável para análise neste estudo, ou seja, testar a associação dos conceitos orientados a objetos com reflexão computacional e, mais adiante, com padrões de *software*, no intuito de tentar contornar esta situação. Com este objetivo, este capítulo mostra inicialmente como tal associação pode ser levada a cabo mantendo o foco sobre o objetivo deste trabalho, ou seja, de forma a obter-se um maior grau de reusabilidade. A seguir, mostra como esta associação tem sido implementada na forma de objetos atômicos, e como a reflexão computacional pode ser utilizada para aumentar a flexibilidade e transparência obtidas pelo uso exclusivo de mecanismos de herança como forma de conexão entre classes que implementam diferentes requisitos.

4.2 Simbiose de conceitos

O modelo orientado a objetos tem como uma de suas premissas a reutilização de soluções, seja no processo de codificação ou ainda durante as fases anteriores do desenvolvimento de *software*. Entre as propriedades básicas pertinentes ao modelo orientado a objetos que podem ser aplicadas com este intuito estão a modularização e o encapsulamento, que também facilitam o trabalho de manutenção das aplicações. Estas estão por demais claras quanto ao quesito reutilização. O objetivo deste trabalho, portanto, não é simplesmente versar sobre estas características da orientação a objetos, mas mostrar como algumas delas podem ser usadas no estudo proposto, especificamente quando associadas a características de outros paradigmas. Neste intuito, estudou-se a associação deste modelo com reflexão computacional e padrões de *software*, entre outros.

A associação orientação a objetos/reflexão computacional dá-se de forma inequívoca na existência de protocolos de meta-objetos, materialização mais visível desta simbiose. Simbiose porque não apenas a reflexão computacional beneficia-se desta associação, mas também o contrário é verdadeiro. A orientação a objetos se beneficia das facilidades reflexivas para diminuir o acoplamento e aumentar a transparência. Já a reflexão computacional vale-se da estruturação permitida pelos objetos e do conseqüente melhor gerenciamento de complexidade no meta-nível. A fusão destes conceito é o meta-objeto.

Kiczales [KIC 91] propõe que o princípio da incrementalidade, que diz que a adição de uma nova funcionalidade seja feita nem necessidade de nova implementação de funcionalidades preexistentes, e o princípio da robustez, que defende que o alcance de eventuais erros provenientes do meta-nível devem afetar o mínimo possível o nível base. Tais princípios na verdade podem ser implementados utilizando a modularidade e o encapsulamento permitidos pela orientação a objetos para incluir novos módulos (princípio da incrementabilidade) e manter

um mínimo de acoplamento entre eles (princípio da robustez). Afirma ainda que tal associação entre reflexão computacional e orientação a objetos. Aquelas tornam possível abrir a implementação sem revelar detalhes desnecessários ou comprometer sua portabilidade, enquanto que as últimas permitem que o ajuste local e incremental da implementação.

Conceitos provenientes de cada um dos membros desta associação também são usados para potencializar características do outro. Por exemplo, o conceito de encapsulamento da orientação a objetos pode ser incrementado pela utilização de reflexão computacional para esconder do programador detalhes de implementação [BLA 99]. Já a decomposição de funcionalidades em módulos torna-os mais reutilizáveis, pois apenas a funcionalidade requerida está presente em cada módulo [ROM 2000]. Este modelo é especialmente apropriado para organizar grupos de componentes [HAE 96], sendo uma das mais importantes características provenientes da junção destes conceitos.

4.3 Modularização, encapsulamento, reutilização e manutenção

O encapsulamento de conjuntos bem definidos de características em diferentes classes específicas permite a modularização das aplicações resultando em maior facilidade de uso e manutenção. Cada uma destas classes pode ser tratada e utilizada de forma isolada e independente. Sua utilização também poderá acontecer em muitos outros sistemas com necessidades similares [RUM 94]. Esta possibilidade torna-se extremamente importante no incremento da velocidade de desenvolvimento de aplicações [FER 98a], pois permite ao programador incorporar tais características a seus sistema de forma rápida, simples e eficiente. Basta selecionar as classes que fornecem suporte às características desejadas no sistema e introduzi-las no mesmo, sem necessidade de recodificação ou mesmo de entendimento de seu funcionamento, pois o encapsulamento exige do programador conhecimento apenas sobre as chamadas a realizar sobre os métodos das classes em questão (*interface*). Além disso, a manutenção de tais programas torna-se muito mais simples, pois pode ser feita de modo estanque apenas sobre as partes que necessitem alteração. Eventuais melhorias nas implementações dos requisitos não funcionais podem ser feitos através de simples substituições de classes, sem que haja necessidade de alteração no código da aplicação.

4.4 Orientação a objetos, reflexão computacional e atomicidade

De acordo com Ancona et al [ANC 95], a tolerância a falhas pode ser implementada pelo próprio sistema, contudo há perda de flexibilidade. Se for suportada por bibliotecas, não serão obtidas nem transparência nem separação de diferentes contextos. Em sistemas orientados a objetos, a herança favorece a obtenção dessa separação, porém a transparência fica prejudicada. A reflexão, desta forma, pode ser associada a este paradigma na implementação da tolerância a falhas.

4.4.1 Dados atômicos

Uma das implementações realizadas neste trabalho utilizou conceitos de dados atômicos, procurando mostrar como é possível implementar classes responsáveis pelo seu próprio monitoramento. Herlihy, Liskov e Weihl [HER 88] [WEI 85] introduziram o conceito de tipos de dados atômicos. Estes são tipos abstratos de dados que possuem sincronização e recuperação. Os objetos atômicos (instâncias de dados atômicos) são responsáveis pela garantia destas propriedades. Stroud e Wu [STR 95a] defendem a implementação de sincronização e recuperação implicitamente, através da utilização de meta-objetos. Wu [WU 97a] vale-se de reflexão computacional em Java para aumentar a flexibilidade de programas durante sua

execução, podendo servir inclusive para introduzir estas propriedades. Um dos cenários desenvolvidos compreende uma meta-classe que pode ser usada em contextos mais amplos, ou seja, não é por si só solução para o problema de suporte a atomicidade. Seu desenvolvimento, conforme será visto, foi feito com dois intuitos: mostrar a possibilidade de modularização de componentes reflexivos e implementar uma classe baseada nos conceitos de dados atômicos vistos anteriormente. A própria classe fornece os meios para garantia de sua recuperação, o que é feito de modo explícito pelo desenvolvedor.

Dados atômicos necessitam, além de suas operações normais, da implementação de sincronização e recuperação. Existem três maneiras fundamentais de implementação dessas características [STR 95a]: implicitamente, explicitamente ou de modo híbrido. Esta classificação baseia-se no responsável pela implantação destes dois mecanismos. No primeiro tipo, o sistema é o responsável, já no segundo o programador deve prover código para isso. O problema associado às implementações explícita e híbrida é o fato de não haver uma separação clara entre o código que lida com a sincronização e recuperação e o código da aplicação propriamente dita. Além disso, é impossível alterar esses mecanismos sem realizar nova implementação [STR 95a]. Muitas vezes, um objeto que implemente mecanismos de concorrência poderia aumentar seu desempenho através da adoção de novo esquema de sincronização, entretanto isto acaba tornando-se impossível. A adoção de uma implementação implícita desses mecanismos poderia não apenas retirar do programador o trabalho de desenvolvimento de código para tal, mas também fornecer meios para alteração do comportamento do objeto atômico durante sua execução. À medida em que novos esquemas fossem necessários, seu código poderia ser carregado. Seria necessário, entretanto, a existência de monitoramento a partir de um nível fora da aplicação, para que tais momentos fossem interceptados. Este formato difere do sugerido por Stroud et al [STR 95b]. Estes mostram um modelo no qual meta-objetos que implementem atomicidade são ligados aos objetos da aplicação no nível base, tornando este objeto atômico, ou seja, provendo a ele controle de concorrência e recuperação, mecanismos estes que estão implementados no meta-nível. Assim, esses requisitos não funcionais são acrescentados à aplicação de forma transparente e não intrusiva. Os objetos atômicos são implementados na forma de meta-objetos, fornecendo métodos para garantir sua atomicidade. Objetos do nível base que necessitem desta característica devem ser associados a eles. Todas as operações chamadas sobre estes objetos são interceptadas pelo meta-objeto, o qual é responsável pela garantia de atomicidade.

A classe de meta-objeto sugerida por Stroud [STR 95a] apresenta a seguinte interface (fig. 4.1):

```
Class AtomicMetaObject : public MetaObject
{
public:
AtomicMetaObject ();
virtual int Meta_BeginComponent (Tid id);
virtual int Meta_CommitComponent (Tid id);
virtual int Meta_AbortComponent (Tid id);
virtual void Meta_InvokeOperation (Tid id, Id mId, Args
args, Rslt rslt);
}
```

FIGURA 4.1 – Interface da classe AtomicMetaObject

Um objeto do nível base pode então ser associado a um meta-objeto desta classe:

Class Minha_classe : **metaobject class** AtomicMetaObject ;

O meta-objeto intercepta as chamadas ao objeto e gerencia a atomicidade. Para flexibilizar sua aplicação, Stroud ainda sugere o desenvolvimento de outras meta-classes, cada uma implementando um algoritmo de concorrência e recuperação. Essas classes são subclasses de AtomicMetaObject, sobrecarregando seus métodos para implementar cada um dos diferentes algoritmos.

Cada vez que um método do objeto é chamado, ocorre uma interceptação do mesmo e o método *Meta_InvokeOperation* de seu meta-objeto associado é executado. Caso seja uma operação de gravação, por exemplo, então o meta-objeto realiza todos os bloqueios necessários antes de repassar o controle ao objeto do nível base.

4.4.2 Objetos como dados atômicos

Objetos atômicos (definidos por Stroud [STR 95a] como instâncias de dados atômicos) que utilizem a implementação implícita requerem pouco trabalho de codificação por parte dos programadores da aplicação. Estes necessitam apenas desenvolver seu código da mesma forma que fariam em um ambiente seqüencial e confiável. Stroud [STR 95a] acrescenta a esta codificação apenas a obrigatoriedade de definição de uma especificação dos conflitos que poderiam existir entre as operações nos objetos. Definem conflito de operações como diferenças de resultados que poderiam acontecer caso a ordem de duas operações fosse alterada.

4.4.3 Implementação explícita com uso de herança

Stroud e Wu [STR 95a] mostram exemplos de definição de conflitos de operações em PC++ [WU 95], nos quais utilizam pré-processamento para lidar com os tipos de dados atômicos, adicionando código necessário ao controle de concorrência e recuperação. Parrington e Shrivastava [PAR 88] mostram o uso da orientação a objetos na definição de mecanismos de tolerância a falhas e seu refinamento em subclasses. Stroud [STR 95a] introduz o uso da reflexão computacional, como será visto na próxima seção, no intuito de aumentar a flexibilidade de tais sistemas, permitindo a alteração dinâmica de seu comportamento, variando os mecanismos de tolerância a falhas.

O uso de pré-processadores e herança de classes com funcionalidades extras possui o inconveniente de apresentar, entretanto, uma grande dificuldade: a perda de flexibilidade. Além disso, novos esquemas e especificações só podem ser incorporados através de alterações no pré-processador. Tal fato pôde ser verificado na implementação realizada em [FER 00] , onde a herança de classes precisou ser utilizada para substituir a interceptação implícita de métodos devido à inexistência de tal mecanismo no protocolo de meta-objetos empregado na implementação (JavaCore).

4.4.4 Implementação implícita com uso de meta-objetos

A reflexão computacional leva para o meta-nível o trabalho de prover as características não funcionais desejadas, função desempenhada pelas classes incorporadas e pelo mecanismo de herança no modelo anterior. A grande vantagem do mecanismo reflexivo é a possibilidade de ajuste no comportamento de um objeto sem necessidade de modificações em sua implementação, mas simplesmente mudando seu meta-objeto. Além disto, obtém-se uma clara separação entre os dois tipos de código (requisitos funcionais e não funcionais).

O protocolo de meta-objetos permite a implementação de operações atômicas pelos programadores no nível base da mesma forma como o fariam em um outro ambiente qualquer. O trabalho necessário para que tal aconteça pode ser feito separada e transparentemente no

meta-nível. Meta-classes que implementem diferentes protocolos de controle de concorrência ou recuperação de falhas podem ser derivadas de uma superclasse, sendo chamadas quando um objeto associado no nível base é criado. Métodos sobre estes objetos podem então ser interceptados e tratados no meta-nível. No meta-nível está o código responsável pela garantia da atomicidade pretendida.

Com a utilização da reflexão na implementação implícita de requisitos não funcionais, obtém-se a possibilidade adicional de alteração dinâmica do protocolo utilizado pelo objeto, dependendo das necessidades da aplicação. Para tal, basta ocorrer uma associação entre o mesmo e o meta-componente que implemente o protocolo desejado. Stroud e Wu [STR 95a] apresentam duas meta-classes em PC++ que implementam dois protocolos de controle de concorrência: otimista e pessimista.

Apesar da necessidade de garantia de consistência ser inerente aos sistemas concorrentes, a mistura do código da aplicação com o código necessário para lograr-se tal característica torna difícil sua execução. O uso de reflexão computacional com sua separação de códigos enquadra-se neste contexto, isolando os aspectos relativos à atomicidade, os quais permanecem transparentes à aplicação. Aspectos não funcionais de atomicidade, como concorrência e recuperação, são implementados em meta-classes no meta-nível, enquanto que operações sobre os objetos, necessárias à execução do programa, são implementadas no nível base. Alterações em cada um dos lados deste conjunto não são vistas e tampouco afetam o outro lado. Novas características não funcionais também poderão ser facilmente incorporadas ao sistema na medida em que se fizerem necessárias. Todas essas características são transparentes e não intrusivas no que concerne à aplicação.

A ligação entre o objeto da aplicação e um meta-objeto pode ser alterada dinamicamente de acordo com a necessidade daquela. No caso do controle de concorrência, vários meta-objetos poderiam implementar diferentes protocolos, os quais poderiam ser ativados ou desativados sem alterar a aplicação [WU 97a].

As chamadas aos métodos dos objetos da aplicação podem ser interceptados e repassados aos meta-objetos associado a eles. O meta-objeto, então, pode executar sua função e, após, devolver o controle ao objeto originalmente chamado. É importante ter-se, então, uma maneira de realizar esta interceptação de forma transparente. Wu et al [WU 97a] propõem o uso de herança de classe. A classe que implementa a reflexão seria uma subclasse da uma classe da aplicação. Os método da classe da aplicação seriam sobrecarregado nessa subclasse reflexiva de forma que uma chamada a eles seria repassada para o meta-objeto. A classe reflexiva teria uma variável “meta-objeto” que referenciaria uma instância da classe. Os programadores criariam suas aplicações a partir das classes reflexivas ao invés das classes originais da aplicação. As chamadas aos métodos dos objetos seriam interceptadas e manipuladas por um meta-objeto.

Os meta-objetos devem ter acesso a informações sobre a aplicação, para que possam melhor decidir a respeito da estratégia a ser usada. No caso de controle de concorrência, um meta-objeto deve decidir que tipo de bloqueio a realizar. Se não receber nenhuma informação adicional, deverá executar bloqueio exclusivo, diminuindo assim o grau de concorrência. Por outro lado, informações em excesso podem tornar o meta-objeto dependente da aplicação, ou seja, inútil para outras aplicações que porventura queiram usá-lo.

4.4.5 Comparação

Através da análise das implementações acima, concluiu-se que o uso de herança para associar os diferentes níveis da arquitetura reflexiva, embora possa ser utilizado em soluções que não demandem maiores graus de flexibilidade e transparência, não obteve o mesmo desempenho em relação a estes fatores quando comparado com o uso de meta-objetos com

interceptação implícita. Devido à importância destes aspectos para a implementação objetivo deste trabalho, esta solução foi deixada de lado em favor da ligação implícita inter-níveis. A falta de um mecanismo de interceptação de mensagens na API nativa Java já havia sido percebida em [FER 00]. Para contornar tal problema, foi desenvolvido um conjunto de classes para suporte a concorrência e recuperação de objetos, utilizando várias técnicas diferentes e mecanismos decisórios e de substituição dinâmica em tempo de execução. A transparência desta solução, todavia, não pôde ser lograda em seu grau máximo, pois foi necessário fazer com que o objeto localizado no nível base fosse subclasse da classe de entrada no meta-nível. Tal solução é mais prejudicial à medida em que a linguagem de programação Java não dá suporte a herança múltipla, o que deve ser contornado através da implementação de *interfaces*, exigindo que eventuais heranças preexistentes fossem readaptadas a esta nova realidade e implementadas sob este formato. A hierarquia de classes desenvolvida em [FER 00] é mostrada na figura 4.2.

A classe do objeto do nível base que recebe suporte deve ser subclasse de "ObjetoAtomico" que, por sua vez, está conectado no meta-nível com uma classe responsável pela manutenção das transações sobre este objeto, utilizando duas classes gerenciadoras de concorrência e recuperação, cada qual com seu mecanismo próprio de decisão sobre a estratégia a executar. Estas estratégias estão implementadas em classes independentes e podem ser selecionadas dinamicamente pelo já citado mecanismo de decisão. Esta implementação, baseada fundamentalmente sobre mecanismos de herança e de prospecção de atributos, mostrou-se menos transparente que as que utilizaram interceptação e prospecção, embora fosse razoavelmente equivalente quanto a sua capacidade de adaptação a novos contextos.

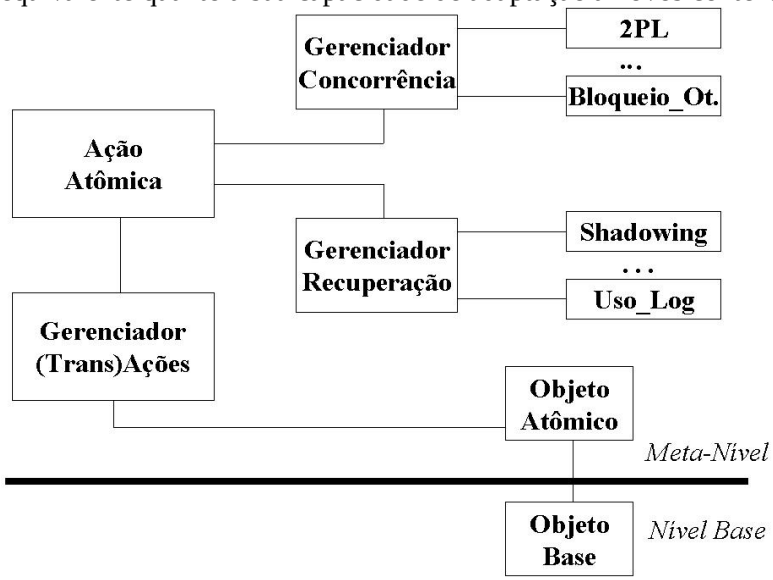

FIGURA 4.2 – Framework reflexivo para implementação de transações atômicas [FER 00]

4.5 Resumo do capítulo

Este capítulo discutiu algumas características do modelo orientado a objetos que favorecem reutilização. Como o próprio modelo foi criado com esta preocupação em mente, conclui-se que pode ser amplamente empregado no desenvolvimento de soluções reutilizáveis. O ponto crucial discutido foi, contudo, a melhor estratégia de utilização destas características com características de outros paradigmas, no intuito de utilizá-las não apenas em conjunto, mas de forma a fazer com que a utilização das características de um paradigma potencialize a ação das características do outro. Uma implementação desta associação foi estudada e, a seguir, feita uma comparação entre o uso de interceptação de mensagens e a aplicação do mecanismo de herança, analisando-se as vantagens e desvantagens de cada via.

5 Utilização de padrões de software para construção de modelos atômicos

5.1 Introdução

Padrões de projeto foi um dos paradigmas estudados e analisados com o objetivo de observar sua efetividade na construção de soluções reutilizáveis. Outra meta foi verificar sua compatibilidade quando aplicado juntamente com outros paradigmas estudados com o mesmo intuito, como o modelo orientado a objetos e a reflexão computacional.

Um modo natural de obter-se reutilização é através do emprego de experiências anteriores bem sucedidas. A aplicação de padrões de projeto já foi por demais demonstrada na literatura existente, não sendo portanto o objetivo deste trabalho analisá-lo de forma isolada. Mais que isto, procurou-se expandir sua aplicabilidade, empregando-o juntamente com reflexão computacional. Foi realizada a implementação de um padrão de projeto voltado a recuperação de estados de objetos [SIL 96a] lançando-se mão de conceitos reflexivos. O objetivo foi observar-se os ganhos ou eventuais desvantagens provenientes desta solução.

Um padrão de projeto é uma regra de três partes, que expressa a relação entre um determinado contexto, um certo sistema de forças que ocorrem repetidamente nesse contexto, e uma configuração de *software* que permite a estas resolverem seus conflitos. Permitem codificar soluções e seus relacionamentos para que possam ser reutilizados em contextos semelhantes [GAB 96]. Documentam um problema, suas características e como solucioná-lo, moldando o relacionamento de forças recorrentes em um contexto específico e uma configuração para resolvê-las [RIE 96].

5.2 Padrões e reutilização

Documentar padrões é uma forma de reutilização e compartilhamento de informações [APP 2000]. A descrição da solução tenta capturar o cerne do problema, de modo que outros possam aprender com ela, além de reutilizá-la em situações similares. Padrões carregam a essência da solução provada do problema dentro de um contexto de forças interagentes. O desafio é tornar estas soluções o mais abrangentes possível, de modo a permitir que sejam mais reutilizáveis. Neste sentido, buscou-se associar conceitos deste paradigma com outros que acenem com a perspectiva de acréscimo de produtividade. Percebeu-se que, embora voltados para a reutilização de soluções, padrões de projetos apresentam certas limitações que não são inerentes à sua definição. Para que uma solução possa ser reutilizada de forma mais abrangente e em um número maior de situações, pode-se realizar sua implementação empregando técnicas que concorram para um menor acoplamento do código a classes e situações específicas.

5.3 Modelos de padrões

Em [APP 2000], definem-se três modelos de padrão, de acordo com seu nível de abstração e detalhe: o modelo conceitual do domínio da aplicação, que utilizam padrões conceituais sobre a mesma, o modelo tradicional do desenho da aplicação, que emprega padrões de desenho, e o modelo de implementação, que usa padrões de programação. Uma classificação semelhante é feita pelo *Group of Five (GoF)* [GAM 95], que define padrões de arquitetura (conjunto de subsistemas predefinidos com responsabilidades e regras para organizar seus relacionamentos), padrões de desenho (descrevem estruturas recorrentes de comunicação de

componentes que resolvem um problema genérico dentro de um contexto específico) e idiomas (padrões de baixo nível, específicos de uma linguagem de programação, que descrevem como implementar aspectos específicos de componentes ou relacionamentos usando as facilidades de uma determinada linguagem de programação). Já [RIE 96] faz distinções semelhantes, porém dividem os padrões em padrões de análise, desenho e implementação.

5.4 Apresentação e organização de padrões

A literatura sobre padrões traz uma forma de apresentação que consiste em três seções: problema, contexto e solução. A primeira descreve concisamente o problema, a segunda as situações onde os problemas ocorrem assim como as forças e restrições para uma possível solução, enquanto que a última mostra como resolver as forças dentro deste contexto. Outra forma de apresentação consiste em várias seções, cada qual descrevendo um certo aspecto do padrão. A estrutura do padrão, seus aspectos estáticos e dinâmicos e os vários modos de usá-lo são os principais focos das seções. As seções que tratam da estrutura e sua dinâmica são chamadas de forma do padrão, enquanto as seções sobre a motivação, aplicabilidade e suas conseqüências mostram o contexto do problema. [RIE 96] sugere apenas duas seções: contexto e padrão. O contexto descreve, além do próprio, as restrições e forças que deram origem ao padrão. A segunda seção descreve a forma do padrão no contexto.

Há várias maneiras de organização de padrões. [GAM 95] utilizam duas dimensões, uma baseada no seu propósito (criacional, estrutural e comportamental) e outra em seu escopo (baseado em classes ou objetos). Buschmann et al [BUS 96][BUS 96] organizam padrões em dimensões de granularidade (arquitetura, desenho, implementação) e funcionalidade (criação, comunicação, acesso, organização).

5.5 Padrões e *frameworks*

Um conjunto de padrões relacionados pode ser utilizado na construção de *frameworks*, os quais realizam uma conexão entre conceitos de padrões e orientação a objetos. Um *framework* é uma mini-arquitetura reutilizável que provê uma estrutura genérica e comportamento para um conjunto de abstrações de software, assim como um contexto de metáforas que especificam sua colaboração e uso dentro de um dado domínio. A definição de [GAM 95] caracteriza-os como conjuntos de classes cooperantes que compõem uma estrutura reutilizável para uma classe específica de *software*. Um desenvolvedor adapta um *framework* para uma aplicação específica através de subclasses e composição de instâncias das classes do mesmo. A diferença entre um *framework* e uma biblioteca comum é que aquele emprega um fluxo de controle invertido entre si e seus clientes. Ao usar um *framework*, normalmente apenas implementa-se algumas funções de chamada ou especializam-se algumas classes, e então invoca-se um único método ou procedimento. A partir daí, o *framework* realiza o resto do trabalho.

Padrões podem ser utilizados no desenho e na documentação de um *framework* [APP 2000] [BED 98]. Um único *framework* tipicamente possui vários padrões. Na verdade, ele pode ser visto como a implementação de um sistema de padrões. Há, entretanto, dois fatores a diferenciá-los: um *framework* é um software executável, enquanto que padrões representam conhecimento e experiência sobre software; além disso, *frameworks* são a realização física de um ou mais padrões de soluções. Estes são instruções sobre como implementar essas soluções. Outras diferenças ressaltadas em [GAM 95] são quanto ao grau de abstração (maior nos padrões), granularidade arquitetural (menor nos padrões) e especialização (padrões são menos especializados).

5.6 Implementação de um padrão de recuperação de estados utilizando técnicas reflexivas

5.6.1 Introdução

O conceito de padrões tem sido empregado dentro do contexto de reutilização de software, pois oferece soluções testadas para determinados problemas recorrentes, de forma a facilitar o desenvolvimento de aplicações mais confiáveis cada vez mais rapidamente [RIE 96]. Padrões de software auxiliam a reduzir a complexidade das aplicações descrevendo formatos de estruturas, comportamento dinâmico e contexto de sua utilização. No domínio da tolerância a falhas, padrões podem ser usados para fornecer soluções de implementação de requisitos não funcionais como ações atômicas, políticas de replicação e exceções [LIS 98].

Já que tanto reflexão computacional quanto padrões de software possuem importantes propriedades que podem ser utilizadas no aumento do grau de reutilização de software, sua combinação pode potencializar tal característica. Padrões mostram soluções reutilizáveis, enquanto que reflexão torna a implementação destas soluções também mais reutilizáveis.

Este cenário mostra como a implementação de um padrão de software sugerido por Silva [SIL 96b] pode ser implementada empregando reflexão computacional para torná-la mais simples e adaptável. A solução apresentada pelo padrão é reutilizável, entretanto o uso de reflexão estendeu esta propriedade também à implementação.

5.6.2 Descrição do cenário

A possibilidade de um objeto ser acessado concorrentemente por vários outros traz à tona a necessidade de evitar que tais situações levem tal objeto a um estado inconsistente. Dados alterados parcialmente devem retornar a seu estado original, ou então as operações inacabadas devem ser refeitas [BER 87].

Um objeto pode receber chamadas a um dos métodos de sua interface que disparem uma série de operações. Estas operações alteram o estado de um objeto, levando de um estado inicial (**ei**) para um final (**ef**). Considera-se aqui estado de um objeto o conjunto corrente de valores de seus atributos. Se, durante a execução desta série de operações, ocorrer uma falha, o objeto externo (cliente) que disparou a execução do método pode ser notificado por uma exceção gerada pelo objeto que recebeu a chamada. Não há, contudo, garantia de que este encontra-se consistente. Portanto, garantir atomicidade a um objeto significa assegurar que seu estado inicial seja preservado como estado corrente caso a transição **ei->ef** não tenha sido completada com sucesso, ou refazer as operações não executadas para levar o objeto ao estado final esperado (fig. 5.1).

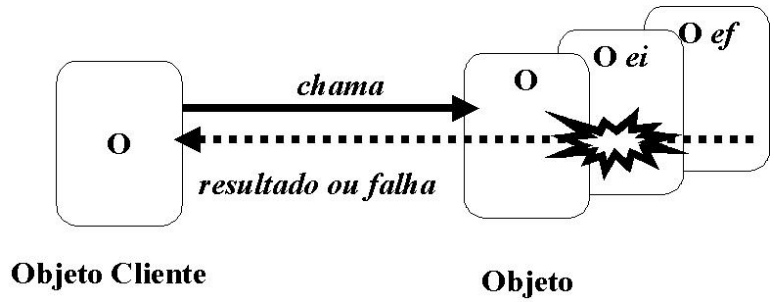

FIGURA 5.1 – Transição de estados do objeto

Lidar com restauração de estados pode ser tão simples quanto fazer uma cópia do objeto original ou tão complexo quanto memória cache de recuperação implementada em hardware [RUB 94]. Desta forma, o problema da recuperação demanda soluções especiais, o que sugere a oportunidade de criação de padrões de desenho de software para diferentes contextos de restauração de estados [LIS 98].

5.6.3 O padrão de recuperação de estados

O padrão de recuperação de estados de objetos [SIL 96a] permite que objetos de uma aplicação suportem sua própria recuperação. Também permite ao desenvolvedor escolher entre diversas políticas de recuperação, de acordo com o contexto da aplicação. Os autores sugerem uma implementação baseada em classes abstratas, conforme mostrada na figura 5.2.

A escolha de classes abstratas para implementar este padrão deixa para o desenvolvedor a tarefa de especializá-las. A implementação reflexiva aqui mostrada estende este padrão para que o mesmo possa oferecer uma conexão mais transparente entre a aplicação e a biblioteca de recuperação. Implementações reflexivas destas políticas são muito mais flexíveis, pois não demandam qualquer conhecimento prévio das classes dos objetos da aplicação [FER 01c].

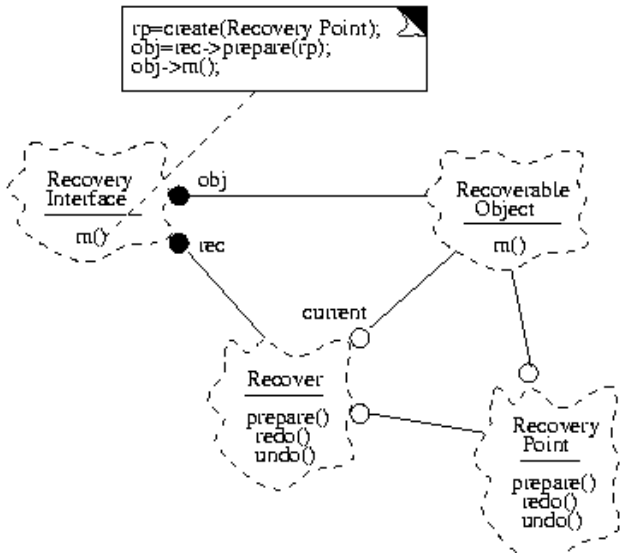

FIGURA 5.2 - O padrão de recuperação de estados [SIL 96a]

5.6.4 Implementação reflexiva do padrão

A meta-classe MC foi implementada para demonstrar como objetos podem se beneficiar com a adição de atomicidade de forma simples e rápida. Esta meta-classe implementa serviços genéricos de suporte a recuperação de estados de objetos. Apresenta métodos que servem como interface para os objetos da aplicação. Sua utilização pode ser feita em vários contextos, pois o objetivo é demonstrar a flexibilidade alcançada pelo uso de reflexão computacional em implementações de padrões.

Uma instância de MC intercepta chamadas para a instância de Recoverable Object (o objeto no nível base) a ela associada, conforme mostrado na figura 5.3. Então este meta-objeto realiza todo o trabalho de introspecção no nível base e salva o estado corrente do objeto para eventual posterior restauração. Para aumentar a flexibilidade e reduzir o *overhead* causado pela reflexão, apenas chamadas a métodos são interceptadas. Isto significa que as operações executadas internamente aos métodos não são tratadas.

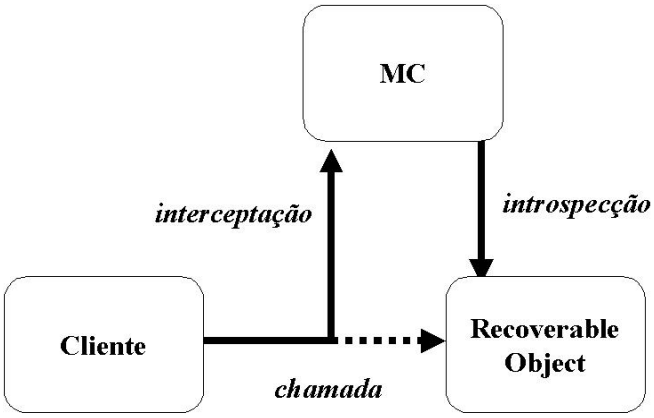

FIGURA 5.3 – Implementação da meta-classe

- A implementação reflexiva correspondente ao padrão de recuperação de estados de objetos (fig. 5.2) usa as classes sugeridas por este, porém enfatiza a transparência:

- *Recoverable Interface*: intercepta chamadas a métodos do objeto base, que corresponde à instância de Recoverable Object no padrão. Seus efeitos são recuperáveis. Este trabalho é realizado pela meta-classe MC. Esta meta-classe pode também atuar como ligação entre o objeto do nível base e as diferentes políticas de recuperação, implementadas no padrão como subclasses da classe Recover. Na implementação reflexiva, isto é feito transparentemente através de meta-configurações múltiplas.

- *Recoverable Object*: é o objeto que contém os dados que são manipulados e podem necessitar ser restaurados. Esta classe corresponde ao objeto do nível base que implementa recuperação.

- *Recover*: parte da política de recuperação independente do objeto. Apresenta as operações Prepare, Redo e Undo. Políticas particulares podem ser implementadas através do uso de herança. Métodos correspondentes a estes são implementados na meta-classe MC e usados no método Handle. Estas políticas podem ser facilmente implementadas simplesmente pela adição ou exclusão de novas meta-classes específicas em meta-configurações múltiplas. Implementações reflexivas substituem os mecanismos de herança através da adição/exclusão dinâmica de meta-classes, o que torna a tais implementações muito mais flexíveis e reutilizáveis, pois novas meta-classes não requerem conhecimento prévio das estruturas das classes preexistentes. Estas não precisam nem mesmo saber de sua existência. As novas classes não precisam ser declaradas como subclasses de Recover.

- *Recovery Point:* parte da política de recuperação específica do objeto. Esta classe pode necessitar conhecer a estrutura interna da instância de Recoverable Object em questão. O uso de reflexão computacional não demanda qualquer tipo de conhecimento prévio de estruturas de classes, pois estas podem ser obtidas em tempo de execução, através do emprego de mecanismos extremamente flexíveis presentes em tal paradigma. Desta forma, qualquer classe pode ser usada como Recoverable Object, já que o código não lida explicitamente com estruturas internas de qualquer classe, o que aumenta consideravelmente o grau de reutilização de tal implementação.

As operações Redo e Undo implementadas pela instância de Recovery Point podem usar diferentes políticas. Tais políticas dependem dos objetivos das operações. Se elas são feitas em cópias dos objetos, então Redo é executado através de sua atualização, caso contrário estes já estão atualizados. Operações de Undo seguem o mesmo padrão: no primeiro caso, o objeto não

sofreu seus efeitos e nada precisa ser feito. Entretanto, se as operações são feitas diretamente sobre os objetos, então estes devem ser restaurados.

Alguns dos atributos da meta-classe MC são usados para implementar as diferentes políticas de recuperação mencionadas. Ela suporta todas sem demandar qualquer alteração. Por exemplo, se as operações forem executadas diretamente sobre os objetos, então o atributo "copy" do tipo Object pode ser empregado. Se a política adotada realizar as alterações sobre cópias dos objetos, então o vetor "values" é usado juntamente com a matriz "fields" do tipo Field para armazenar cópias correntes ou anteriores do estado do objeto.

Novas meta-classes com diferentes políticas podem ser acrescentadas à meta-configuração e usar os atributos de MC de acordo com suas necessidades. Desta maneira, o grau de reutilização obtido é aumentado, pois não é necessário conhecimento prévio sobre as classes ou políticas envolvidas.

5.6.5 Estrutura da meta-classe

A meta-classe MC usa muita características reflexivas para obter importantes informações sobre o objeto base e decidir sobre a reificação de dados. Entre seus atributos há uma referência ao objeto do nível base, uma matriz que armazena os campos do objeto, um vetor com os valores destes campos e uma variável utilizada como *flag* na ativação/desativação da reificação. Também há uma *hashTable* e uma instância da classe *OperationFactory*. Esta classe é responsável pela criação de operações no meta-nível no protocolo de meta-objetos Guaraná. Um objeto de MC deve ser instanciado usando um construtor parametrizado (fig. 5.4):

```
public MC (Object ob_) {
copy = ob_;
Class c = copy.getClass( );
fields = c.getDeclaredFields( );
for (int i = 0; i<fields.length; i++)
        try{
                values.addElement(fields[i].get(copy));
        }catch (IllegalAccessException e) {}
Guarana.reconfigure(copy,null,this);
}
```

FIGURA 5.4 – Método construtor da meta-classe

O objeto do nível base deve ser enviado como parâmetro para o construtor da meta-classe:

```
Class c = new Class( );
MC mc = new MC(c);
```

O construtor de MC salva uma referência ao objeto do nível base, obtém informações sobre a estrutura e dados correntes da classe, como quantidade e tipos de campos, e salva seu estado inicial no meta-nível. Após, associa este objeto a si próprio, iniciando a interceptação de mensagens. Uma primeira vantagem advinda deste mecanismo é sua total transparência ao objeto base, além de poder ser reutilizado com instâncias de qualquer outra classe. Ele não depende da estrutura da classe, nem tampouco precisa conhecê-la *a priori*.

5.6.6 A interface de MC

A definição da meta-classe responsável pela interceptação de mensagens para o objeto no nível base da aplicação, salvando e restaurando seu estado, apresenta s seguintes atributos (fig. 5.5):

```
public class MC extends MetaObject {
private Object copy;                                    // a reference
to the base level object
private Vector values = new Vector( );                  // stores
reified values of the base level object
private Field[ ] fields;                                 // stores
the base level object fields
private Hashtable pending = new Hashtable( );  // used to
avoid infinite recursion
private OperationFactory opf;                            // creates
operations in the meta-level
private int on_off = 0;                                  // flag .
}
```

FIGURA 5.5 – Atributos da meta-classe MC

5.6.7 Gravação de estados dos objetos

Reificação é a chave para a inspeção e salvamento do estado do objeto. Os métodos desenvolvidos com este propósito foram:

- *Reifica:* reifica o objeto do nível base, criando operações de leitura no meta-nível. Tais operações são criadas por uma instância de *OperationFactory* (fig. 5.6).

```
Operation op = opf.read(fields[i]);
pending.put(op,op);
Object value = op.perform( ).getObjectValue( );
pending.remove(op);
```

FIGURA 5.6 – Reificação de estados de objetos desconhecidos

Estas operações criados no meta-nível são armazenadas em uma *hashTable* para evitar recursão infinita, um dos problemas encontrados ao lidar com interceptação de mensagens.

- *Salva:* restaura os valores armazenados no meta-nível de volta sobre o objeto do nível base. Também usa uma instância de *OperationFactory* para criar operações de escrita no meta-nível (fig. 5.7).

```
Operation op =opf.write(fields[i],values.elementAt(i));
pending.put(op,op);
Object value = op.perform( );
pending.remove(op);
```

FIGURA 5.7 – Restauração de estados de objetos desconhecidos

Assim como no método anterior, operações criadas no meta-nível são armazenadas em uma *hashTable* para evitar recursão infinita. Isto é logrado através da inclusão da operação na *hashTable* imediatamente antes de sua execução e removidas logo após seu término. O teste mostrado no método Handle faz o resto.

O estado do objeto é salvo apenas uma vez antes da execução do método. A função *isMethodInvocation* do protocolo de meta-objetos Guaraná, usada no método Handle no meta-

nível, permite descobrir tal situação. A meta-classe também fornece uma interface para seu objeto associado no nível base formada por quatro métodos:

- *Permanente:* desabilita interceptações para gravação, mantendo a última versão do objeto do nível base gravada no meta-nível.
- *Temporário:* habilita interceptações para gravação, de forma que o estado corrente do objeto possa ser salvo no meta-nível a cada nova chamada de método.
- *Restaura:* restaura o último estado salvo do objeto armazenado no meta-nível de volta sobre o objeto no nível base.
- *Checkpoint*: reifica o objeto do nível base e então desabilita posteriores interceptações para gravação.

O meta-objeto inicialmente explora o objeto a ele associado para descobrir seus nomes e tipos de campos, assim como seus valores correntes, e grava todas estas informações no meta-nível. Ele também possui uma referência para o objeto no nível base, para posteriores acessos. A seguir, o meta-objeto realiza a conexão entre si e seu objeto base. A partir deste momento, o comportamento do objeto base pode ser controlado a partir do meta-nível, de acordo com as necessidades do desenvolvedor, utilizando os métodos fornecidos pela interface do meta-objeto.

A figura 5.8 mostra um exemplo de como usar tal interface. O método “Permanente” desabilita interceptações para gravação, tornando todas as alterações sobre o objeto base permanentes. Quando o método “Checkpoint” é chamado, o estado corrente do objeto é salvo no meta-nível e posteriores gravações são desabilitadas. Operações executadas no objeto do nível base a partir deste momento podem ser desfeitas através do método “Restaura” ou confirmadas com o método “Checkpoint”. O método “Temporário” habilita as interceptações de novo e reinicia a gravação dos valores correntes do objeto do nível base a cada nova chamada a um de seus métodos.

```
mc.permanente( );
... chamadas a métodos sobre o objeto base...
mc.checkpoint( );
... chamadas a métodos sobre o objeto base...
mc.restaura( ); ou mc.checkpoint( );
```

FIGURA 5.8 – Exemplo de utilização da interface de MC

1. Permanent, temporary, restore e checkpoint: usam os métodos anteriores e a variável flag.

A capacidade de criar operações no meta-nível traz à reflexão computacional um novo leque de possibilidades. As operações podem não apenas ser interceptadas, como também ser criadas de acordo com o contexto. A meta-classe MC cria operações de leitura e escrita e as executa no meta-nível, sem o conhecimento do objeto do nível base.

- *Handle:* este método é chamado pelo kernel do protocolo de meta-objetos sempre que uma chamada é enviada ao objeto associado no nível base. Recursão infinita entre ambos os níveis é causada por uma operação no meta-nível que acesse um objeto do nível base que esteja associado a um meta-objeto. Isto leva a uma nova interceptação e assim por diante. Para evitar este problema, esta meta-classe emprega uma mecanismo de inclusão/exclusão e uma *hashTable*, que é usada para testar se a operação é uma nova ou apenas a mesma gerando recursão, conforme pode ser visto no código-exemplo a seguir:

```
if (pending.containsKey(op))
        return null;
```

Se a operação já estiver armazenada na *hashTable*, a recursão é evitada pela instrução "return null", que envia o fluxo de execução de volta para o nível base. Para diminuir o *overhead*, apenas chamadas a métodos são tratadas. Isto evita que cada instrução dentro do método gere nova gravação no meta-nível. Isto é feito apenas uma vez, no início do processo (fig. 5.9).

```
if ( op.isMethodInvocation( ) && on_off == 0 )
        return null; //não reifica

if ( op.isMethodInvocation( ) && on_ff == 1 )
        reifica( );   //reifica

if ( op.isMethodInvocation( ) && on_off == 2 )
        salva( );     //grava

if ( op.isMethodInvocation( ) && on_off == 3 )
{
        reifica( );      //reifica uma vez e desliga
        on_off = 0;
 }
```

FIGURA 5.9 – Testando primeira chamada a métodos

5.6.8 Composição de meta-classes

A meta-classe MC mostra como implementar em baixo nível de abstração atomicidade utilizando reflexão computacional. Permite atingir reutilização e transparência e diminuir o *overhead* que pode ser introduzido pela utilização da reflexão. O nível de abstração empregado também permite maior flexibilidade em sua utilização em diversos cenários, conforme já frisado. A partir de implementações deste tipo, pode-se chegar a *frameworks* que possuam mecanismos de decisão que permitam cobrir um número cada vez maior de cenários de suporte à atomicidade.

Uma das formas sugeridas de aglutinação de meta-classes são as classes *Composer* e *SequencialComposer*, do protocolo de meta-objetos Guaraná, que possibilitam a delegação de métodos a outras meta-classes, através de associações (configurações múltiplas de meta-nível) entre vários objetos e meta-objetos. Esta delegação possibilita reconfiguração dinâmica, o que não é obtido apenas pelo uso de mecanismos de herança.

5.6.9 Contribuição da classe e sugestões

Esta meta-classe permite que um objeto possa gerenciar sua própria recuperação de estados, evitando que o desenvolvedor precise implementar tal requisito. Objetos de qualquer classe podem valer-se de tal propriedade, pois a meta-classe utiliza reflexão computacional para descobrir em tempo de execução os atributos do objeto base. A recuperação pode ser ativada e desativada a qualquer momento, de acordo com as necessidades da aplicação. Além disso, a separação entre requisitos funcionais e não funcionais não apenas aumenta seu grau de reutilização, como também a velocidade de desenvolvimento. O trabalho do desenvolvedor é simplificado, pois o mesmo pode preocupar-se apenas com os aspectos para os quais a aplicação está sendo desenvolvida.

Requisitos não funcionais como tolerância a falhas, processamento distribuído, persistência de dados e controle de concorrência podem ser acrescentados de forma transparente às aplicações, utilizando-se técnicas aqui discutidas, com alto grau de transparência e

reutilização, além de incrementar a produtividade do desenvolvedor, com maior segurança e simplicidade.

A meta-classe MC aqui descrita permite que um objeto suporte sua própria recuperação de estado, liberando o desenvolvedor de tal tarefa. Objetos de qualquer classe podem se beneficiar da associação com esta meta-classe, pois mecanismos de reflexão computacional foram usados para criar uma meta-classe genérica. A recuperação de estados pode ser ativada ou desativada a qualquer momento, de acordo com as demandas dinâmicas da aplicação. Além disso, requisitos funcionais e não funcionais são separados, incrementando o reuso e a velocidade de desenvolvimento, entre vários outros ganhos. O trabalho do desenvolvedor é simplificado e este pode gastar seu tempo com as tarefas específicas da aplicação.

Implementações baseadas em padrões e reflexão computacional possibilitam não apenas desenvolvimento mais rápido de aplicações sem diminuir sua confiabilidade, como aumentam seu grau de reutilização. Requisitos não funcionais, como atomicidade, estão presentes em praticamente qualquer tipo de aplicação sendo, por isso, candidatos naturais a serem implementados de forma reutilizável. A reflexão computacional tem sido utilizada na implementação deste tipo de requisitos pela quantidade de características que apresenta neste sentido. Implementações de padrões de projeto utilizando características reflexivas são, portanto, uma associação facilmente vislumbrável quando tratamos de reutilização de *software*.

Meta-classes do protocolo Guaraná, como *Composer* e *SequencialComposer*, podem ser utilizadas na agregação de outras classes no meta-nível, cada uma das quais implementando um aspecto diferente de tolerância a falhas, como controle de concorrência, recuperação de dados, persistência, etc. Não apenas uma, mas várias políticas diferentes de cada um destes aspectos pode ser implementada. Mecanismos de decisão incorporados ao meta-nível podem decidir quando e quais políticas utilizar, simplesmente alterando a meta-configuração corrente. Por estarem encapsulados em classes, a conexão de tais mecanismos entre si e entre os objetos do nível base pode ser feita dinamicamente. Por utilizar mecanismos reflexivos, tais políticas e meta-classes não precisam ser previamente conhecidas.

Como cada uma destas instâncias implementa um requisito não funcional ou uma política alternativa de um, tudo que o desenvolvedor precisa fazer é conectar os objetos de sua aplicação aos meta-objetos cujas características deseja incorporar aos eles. A adoção ou alteração de políticas pode ser feita dinamicamente, sem que seja necessária sequer uma linha adicional de código, tornando as aplicações que utilizem estas técnicas mais simples, transparentes e eficazes.

5.6.10 Reutilização das técnicas empregadas

O emprego de um padrão previamente testado foi por si só prova da facilidade e eficácia da reutilização propiciada por este paradigma. Sua eficiência foi incrementada pela implementação utilizando reflexão computacional e o modelo orientado a objetos. O padrão prevê uma solução para um problema, porém a inserção de reflexão computacional leva a reutilização desta solução às últimas conseqüências, tornando a mesma implementação da solução reutilizável para qualquer classe, mesmo não conhecidas *a priori*.

5.6.11 Conclusões

A associação entre padrões e reflexão é uma das mais promissoras no campo de reutilização de *software*, especialmente por cobrirem partes distintas do processo de desenvolvimento de aplicações. Pôde-se perceber diferenças no grau de reutilização entre

implementações de um mesmo padrão utilizando reflexão computacional ou simplesmente aplicando métodos convencionais.

5.7 O Padrão Reflexão

A reflexão computacional pode ser vista como um padrão, pois apresenta uma solução bem definida para resolução de certos tipos de problemas. Este padrão divide a aplicação em duas partes, o meta-nível e o nível base. Sua associação é feita de modo transparente através de mecanismos de interceptação. Mensagens enviadas a objetos no nível base associados a meta-objetos são por estes interceptadas. De acordo com este padrão, deve ser possível projetar um sistema concentrando-se apenas em seus requisitos funcionais, integrando-se então os componentes não funcionais do mesmo, sem necessidade de alteração de sua arquitetura original.

5.8 Padrões e reflexão computacional

Padrões podem valer-se do modelo orientado a objeto para produzir soluções com alto grau de reusabilidade, melhor estruturação e menor custo de desenvolvimento. Também podem ser combinados na criação de *frameworks*, atingindo desta forma maior reusabilidade, porém é em sua aplicação associada à reflexão computacional que muitas novas possibilidades podem ser vislumbradas.

Padrões de software apresentam ótimas oportunidades de reuso, contudo sua abrangência muitas vezes restringe-se a certas situações e contextos específicos. Seu espectro de aplicação pode ser aumentado caso consiga-se embutir maior grau de adaptabilidade à sua implementação. Se uma solução desenhada por um padrão possuir capacidade de adaptar-se a diferentes contextos, obter-se-á maior grau de reuso. Isto implica em pensar na implementação de padrões utilizando ferramentas propícias ao suporte de tal característica.

Padrões mostram uma solução reutilizável, enquanto que reflexão computacional transforma esta solução reutilizável em uma implementação reutilizável. Ambos conceitos complementam-se simbioticamente de forma a promover maior adaptação à aplicação desenvolvida empregando tais paradigmas. As características provenientes do uso de reflexão permitem a implementação de um padrão de forma a torná-lo bem mais abrangente do que inicialmente pretendido. O emprego de reflexão computacional dá à implementação de um padrão de *software* maior grau de reutilização, flexibilidade e transparência.

Os motivos listados e as implementações analisadas levaram à conclusão que requisitos não funcionais, pelas suas demandas intrínsecas, são candidatos naturais a beneficiar-se da associação padrões de *software*/reflexão computacional. Implementações reflexivas de padrões de projetos ampliam sua abrangência e adaptabilidade, incrementando consequentemente sua reutilização [FER 01a] [FER 01c].

6 Cenários de implementações de atomicidade

6.1 Introdução

Existem diversos cenários nos quais deve-se garantir atomicidade aos dados de uma transação. Cada um destes cenários, entretanto, exige uma forma apropriada de ação, ou seja, uma estratégia particular de implementação. Aplicações distribuídas, por exemplo, apresentam requisitos adicionais aos existentes nas centralizadas. Também há diversidade em relação aos dados, que podem estar armazenados em mídia volátil ou necessitar gravação. Além disso, o acesso aos mesmos pode ser irrestrito, restrito a leituras, compartilhado ou exclusivo. Cada variável presente em uma determinada situação acrescenta novas exigências e problemas a observar. Não sendo possível elaborar uma estratégia que cubra todas as possibilidades existentes de maneira satisfatória, buscam-se soluções que possam ser adaptadas ao maior número possível de cenários, por meio de técnicas que não restrinjam sua utilização. A importância da reutilização provem do fato que esta é a chave para atingir este objetivo de cobrir um maior número de cenários.

Na busca por soluções reutilizáveis, vários aspectos foram levados em consideração. Em cada um dos cenários estudados, procurou-se desenvolver soluções que lidassem com códigos genéricos, ou seja, não estivessem atreladas a referências a classes, métodos ou atributos. Ao invés, buscou-se implementar mecanismos que pudessem prospectá-los dinamicamente. Técnicas que permitam à aplicação auto adaptar-se em tempo de execução, de acordo com o contexto no qual está inserida também são fatores a concorrer para o aumento do grau de reutilização. Além disso, almejou-se permitir alterações em seu comportamento sem que isto implicasse em necessidade de mudanças na aplicação. As soluções foram estruturadas de forma a isolar suas funções em módulos estanques.

Observando-se as demandas intrínsecas a cada tipo de aplicação e as características que espera-se que cada uma apresente, pode-se concluir pelo emprego de um dos vários mecanismos que permitem a introdução de garantias de atomicidade. Nem todo mecanismo pode ser utilizado para determinado contexto. É necessário que suas características sejam propícias ao cenário em estudo. Não se pode aplicar um único mecanismo a todos os cenários possíveis. Essa diversidade dificulta a obtenção de uma solução padrão mas, ao mesmo tempo, permite um maior grau de otimização, pois cada solução toma em consideração fatores mais específicos do ambiente de execução.

Para que fosse possível observar e estudar o desempenho de mecanismos de atomicidade em diferentes situações, cenários foram criados e diferentes soluções empregadas no intuito de comparar mecanismos e suas possibilidades de reutilização.

6.2 Objetivos

O objetivo das implementações de diversos cenários foi verificar a eficiência da utilização de características de diversos paradigmas, vislumbrando-se quando e como cada uma delas pode ser empregada nos mesmos. Os cenários selecionados procuraram abranger diferentes procedimentos de introdução e gerenciamento de atomicidade, permitindo assim uma exploração maior dos paradigmas estudados e, por conseguinte, possibilitando maior quantidade de parâmetros de comparação com diferentes implementações.

6.3 Cenário 1 - Aplicações concorrentes sem acesso competitivo aos dados

6.3.1 Descrição do cenário

O primeiro cenário desenvolvido mostra uma aplicação centralizada que realiza acessos concorrentes a dados disjuntos. Tais acessos podem acontecer aleatoriamente, ou seja, a aplicação possui um número indeterminado de processos utilizando tais dados de forma imprevisível. Estes processos, todavia, guardam estreita ligação entre si pois, caso um deles não seja completado com sucesso, não apenas seus efeitos mas também todos os efeitos de todos os outros processos concorrentes devem ser desfeitos. Os processos devem ser monitorados de forma que a percepção do fracasso de um deles dispare um mecanismo de desativação do efeito dos outros. A aplicação termina quando todos seus processos tiverem modificado seus dados com sucesso. O problema é perceber o término com sucesso de todos os processos ou descobrir que algum deles falhou e, neste caso, restaurar os estados originais dos objetos participantes.

As soluções desenvolvidas detectam a falha na execução de qualquer um dos processos participantes e, a partir deste ponto, desfazem todas as ações executadas não apenas pelo processo defeituoso, mas por todos os outros participantes da aplicação. Desta forma, garantem sua atomicidade e consistência. Os participantes são naturalmente isolados um dos outros, pois não acessam os mesmos dados. Quanto à durabilidade dos dados, estes estão presentes apenas em mídia volátil, podendo ser eventualmente passados para mídia de armazenamento permanente. Uma observação importante a ser feita sobre a garantia de atomicidade neste cenário é que esta lida apenas com o fator de recuperação de estados, pois a própria característica do cenário não demanda controle de concorrência no acesso aos dados, pelos motivos já explicitados. A recuperação de dados pode ser feita através de *rollback*, restaurando-se o estado original dos objetos, ou ainda através da nova execução dos procedimentos que falharam. As soluções implementadas utilizaram a primeira via.

A figura 6.1 descreve de forma genérica o cenário a ser implementado. Vários processos na forma de *threads* acessam conjuntos disjuntos de dados. É necessário que todos consigam terminar suas operações a contento, caso contrário todas as operações realizadas por todos os processos sobre todos os dados devem ser desfeitas. Os outros processos devem ter ciência do término com sucesso de todos os outros, ou seja, deve haver alguma forma de interação entre eles. Os dados precisam ter seus valores originais armazenados para possibilitar posterior recuperação.

As forma de interação entre os processos foi uma das maneiras de testar e comparar os paradigmas em estudo. Mais de uma solução foi estudada para este problema, utilizando diferentes técnicas. Uma solução utilizou uma *thread* monitora para executar tal tarefa, enquanto que outra empregou término explícito de *threads*.

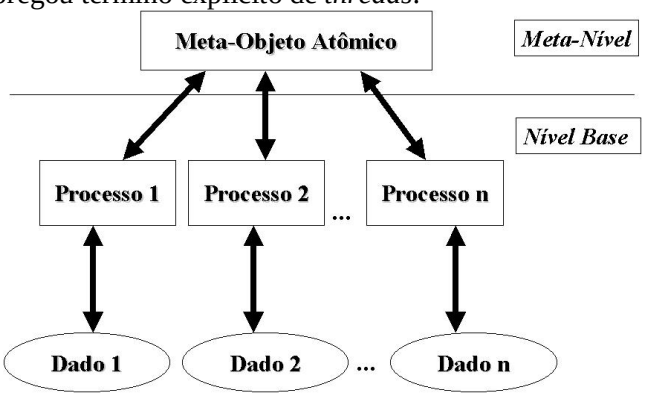

FIGURA 6.1 – Arquitetura do cenário

6.3.2 Utilização de thread monitora

A implementação do primeiro cenário mostra cada um dos processos agrupado em *threads* que devem ser executadas de forma atômica, ou seja, caso alguma dessas *threads* não seja concluída com sucesso, todas as outras devem ser desfeitas. Para torná-las uma unidade compacta, foram associadas em *ThreadGroups*, um mecanismo em nível de linguagem que agrupa diversas *threads* e executa determinadas operações sobre todas. O mecanismo de interação entre os processos foi implementado através de uma *thread* específica cuja função é apenas monitorar o estado de cada uma das outras *threads* componentes e, em caso de eventual falha, disparar o mecanismo de recuperação.

Java é uma linguagem orientada a objetos e, portanto, todas as suas soluções são feitas com base neste modelo. A criação de um grupo de *threads* é feito da seguinte maneira:

- Criar uma instância da classe *ThreadGroup*
- Utilizar o construtor da classe *Thread* que a associa a uma *ThreadGroup*: *Thread*(*ThreadGroup*, *String*).

O uso de *ThreadGroups* também permite a produção de código genérico, pois é possível descobrir o número de *threads* contidas no mesmo e acessá-las através de índices. Com isso, não é necessário fixar-se o número de processos participantes. Através de vários métodos disponíveis nesta classe, pode-se obter em tempo de execução várias informações, como:

- referências às *threads* participantes
- prioridade da *ThreadGroup*
- estado atual da *ThreadGroup*
- número de *threads* ativas

ThreadGroups também permitem a execução de ações conjuntas sobre suas *threads* componentes, como:

- parar o processamento das *threads*
- suspender o processamento das *threads*
- continuar o processamento, após suspensão
- destruir a *ThreadGroup*

Para que seja possível recuperar o estado original dos dados, os mesmos devem ser armazenados antes que as *threads* comecem a operar. Há dois mecanismos utilizáveis para tal: o emprego da API Java para serialização em mídia não volátil (gravação em disco rígido) ou utilização de reflexão computacional para descobrir os tipos e valores dos dados e criar cópias em memória volátil. A decisão a ser tomada na gravação em mídia persistente é a localização e formato dos dados. A utilização de componentes mecânicos leva a menores desempenhos. Embora a reflexão computacional traga embutido certo *overhead*, sua execução em mídia magnética faz com que seu desempenho seja melhor.

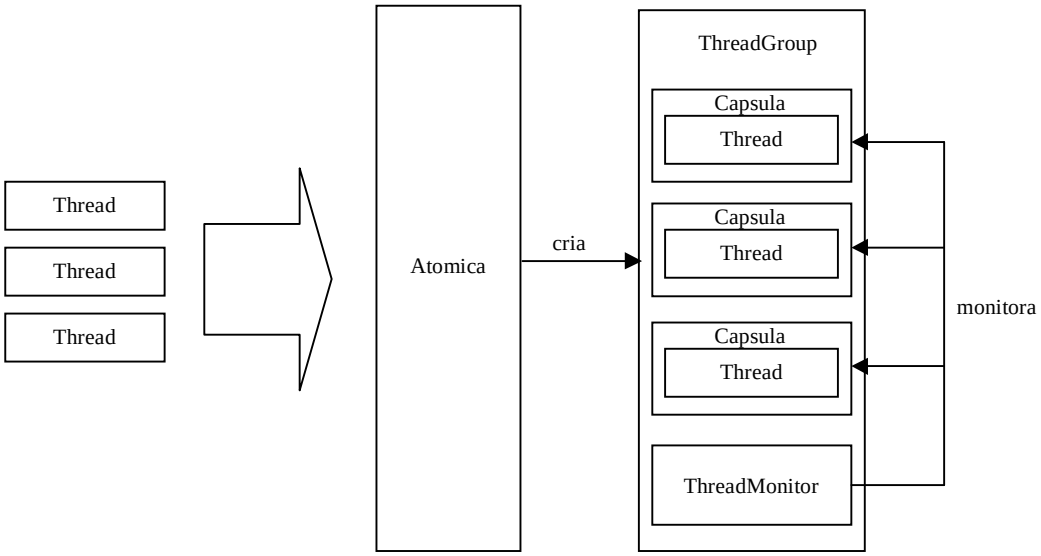

FIGURA 6.2 - Arquitetura da solução 1

Conforme pode ser verificado na figura 6.2, as *threads* são repassadas para um objeto da classe Atomica, ao qual cabe a tarefa de criar uma instância da classe ThreadMonitor e colocar cada uma das *threads* recebidas dentro de uma instância da classe Capsula. Esta última deve iniciar a execução da *thread* e testar seu término com sucesso ou não. Caso haja algum erro, a instância da classe ThreadMonitor, permanentemente monitorando este contexto, chama as funções do objeto Atomica para que as ações de recuperação de estados sejam executadas. Quando todas as *threads* terminam com sucesso, ThreadMonitor termina a execução do programa. Todas as instâncias de Capsula e a instância de ThreadMonitor são colocadas dentro de um mesmo *ThreadGroup* para que o monitoramento seja facilitado. Outra função das instâncias de Capsula é criar um meta-objeto para sua *thread* e realizar a associação entre os mesmos. O meta-objeto será responsável por descobrir eventuais exceções geradas pelas *threads* e setar uma *flag* na instância de Capsula que alertará a instância de ThreadMonitor para o fato, iniciando assim o processo de recuperação.

O objetivo desta solução foi torná-la o mais transparente possível para o programador. Desta forma, tudo que este precisaria fazer seria enviar uma matriz com as *threads* que devem ser executadas atomicamente. Não é necessário que seu código seja alterado ou ainda possua conhecimento de conceitos como *ThreadGroups*.

A implementação da primeira solução, apesar da arquitetura ser mais elegante e transparente que a segunda, encontrou um grande problema na implementação da linguagem de programação Java: as *threads* criadas nesta linguagem não retornam erros de execução, ou seja, não há como conhecer o resultado final de sua execução. Embora seja possível descobrir se uma *thread* está ativa ou não, o mesmo não é possível quanto ao motivo do fim de sua atividade. Desta forma, não há como uma classe externa monitorar o sucesso de execução de uma *thread*. É possível saber se a mesma terminou ou não, mas não o motivo deste término. O único modo é fazendo com que esta sinalize seu final, através da inclusão de código na mesma, o que reduziria drasticamente a transparência e facilidade de utilização da primeira solução proposta. Este problema acabou sendo contornado com o uso de técnicas reflexivas, as quais permitiram a manutenção dos benefícios desta primeira implementação.

A utilização de reflexão computacional também permitiu contornar situações nas quais dificuldades inerentes ao uso da linguagem de programação foram interpostas. A dificuldade de determinação do resultado final da execução de uma *thread* pôde ser contornada através de mecanismos reflexivos de interceptação e tratamento de mensagens no meta-nível. Por conseguinte, tornou-se possível a implementação da primeira solução deste cenário, sem que, para isso, fosse necessário diminuir o grau de simplicidade e reutilização, conforme poder-se-ia

estar propenso a concluir, considerando-se as dificuldades intrínsecas à linguagem de programação ora em uso.

O mecanismo de interceptação possibilita o gerenciamento de mensagens enviadas por um determinado objeto a outro. As *threads* em questão disparam mensagens sobre seus objetos compartilhados, porém a utilização do tratamento de erros *try/catch* de Java ou ainda os métodos como *isAlive( )* específicos da classe Thread não garantem a obtenção do resultado final da execução de sua execução. A estratégia adotada, como conseqüência deste fato, foi a criação de meta-objetos específicos para a prospecção deste tipo de informação, conforme mostrado na figura 6.3.

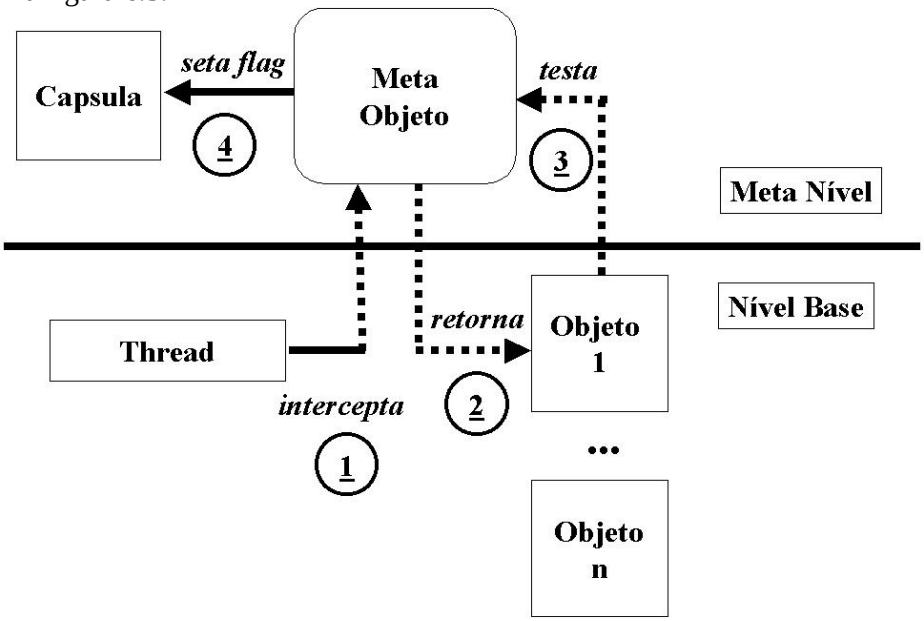

FIGURA 6.3 – Utilização de interceptação de mensagens para prospecção de threads

Esta arquitetura mostra um meta-objeto interceptando mensagens enviadas por uma objeto do tipo *thread* para um outro objeto alvo qualquer. O meta-objeto permite que a operação seja executada sobre o objeto no nível base porém, valendo-se de características do meta-protocolo Guaraná, obriga o retorno do fluxo ao meta-nível após a execução de tal operação (fig. 6.4). Com isso, pode testar se a mesma foi executada com sucesso. Em caso negativo, envia uma mensagem para a instância da classe Capsula associada à *thread* que causou a exceção. Isto torna possível à instância de ThreadMonitor verificar o fato e iniciar o processo de recuperação dos estados iniciais dos dados da aplicação.

```
public Result handle(final Operation op ,final Object ob)
   {
         return Result.inspectResult;
   }
```

FIGURA 6.4 – Primeiro método handle da meta-classe

Após a execução do método no nível base, o controle volta ao meta-nível, onde seu resultado pode ser testado, observando-se então se houve alguma exceção (fig. 6.5):

```
public Result handle(final Result res ,final Object ob)
   {
        if (res.isException())
          ... iniciar processo de recuperação...
        return null;
   }
```

FIGURA 6.5 – Segundo método handle da meta-classe

Os processos participantes são passados para uma instância da classe Atomica, que encapsula cada uma delas dentro de uma instância de *ThreadGroup* e inicia sua execução. Cada *thread* executa suas operações normalmente, sendo monitoradas pela *thread* extra da classe *ThreadMonitor*. Caso uma das *threads* participantes não consiga terminar seu conjunto de operações com sucesso, todas as operações de todas as *threads* são desfeitas, mantendo assim a atomicidade dos dados em questão. Esta arquitetura funciona com qualquer número de *threads* participantes, sem necessidade de alteração de código.

6.3.3 Threads com término explícito

Sem o uso das capacidades reflexivas fornecidas pelo meta-protocolo, é necessário que cada *thread* sinalize seu término, sendo a falta dessa sinalização a evidência do acontecimento de uma exceção. Esta solução foi desenvolvida no intuito de oferecer-se a possibilidade de comparação entre duas soluções com diferentes graus de transparência, simplicidade e sobrecarga devido ao emprego de técnicas distintas.

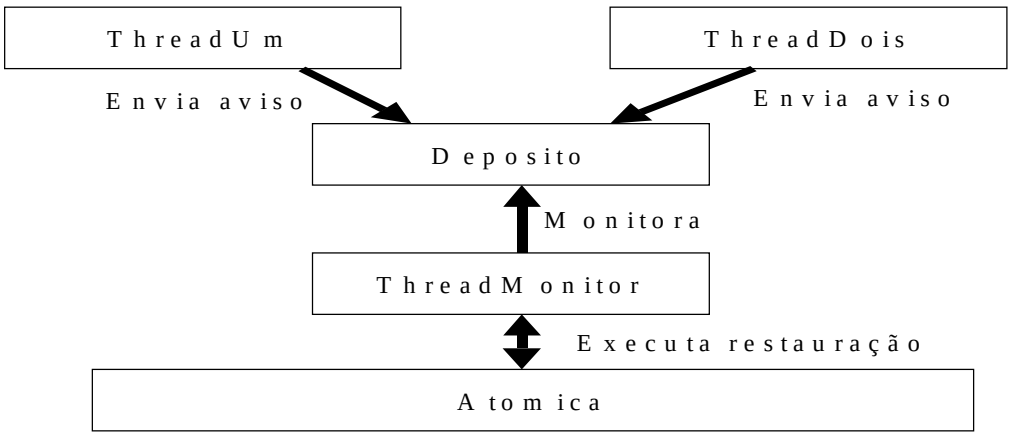

FIGURA 6.6 - Arquitetura da solução 2

A figura 6.6 mostra a segunda implementação ainda para o primeiro cenário. Esta foi feita para mostrar as dificuldades enfrentadas pela não utilização de reflexão computacional neste cenário. Se não for possível descobrir o estado de uma *thread* ao seu término, é necessário que a mesma sinalize tal fato. Isto demanda uma instrução final dentro da *thread* que seja interpretada como sinal. Pode-se usar a alteração do valor de uma variável, por exemplo, ou algum outro artifício. De qualquer forma, devido ao problema de escopo de variáveis em Java, o objeto cuja variável serviria de *flag* de finalização deveria ser passada como parâmetro para todas as *threads* envolvidas na conversação, tal como foi feito nesta segunda implementação. A classe Deposito contém esta variável. Uma mesma instância é criada e passada a todas as *threads*, inclusive a instância de ThreadMonitor, e seu valor corresponde ao sucesso ou falha das *threads*. Cada *thread* que termina incrementa seu valor. A ThreadMonitor conta o número de *threads* ativas na ThreadGroup e compara com o valor da variável. Quando não houver mais *threads* ativas, o valor da variável na instância de Deposito deve ser igual ao número total de *threads* participantes, ou seja, apenas a *thread* Monitor.

6.3.4 Implementação com thread monitora

Duas diferentes implementações de soluções foram desenvolvidas para o primeiro cenário. As classes utilizadas na primeira implementação encontram-se descritas na tabela 6.1:

TABELA 6.1 – Classes da primeira implementação do cenário 1

Nome da Classe	Função
TheadUm	*Thread* qualquer do programador a ser atomizada
TheadDois	*Thread* qualquer do programador a ser atomizada
ThreadTres	*Thread* qualquer do programador a ser atomizada
ThreadMonitor	Monitora o ciclo das *threads* do programador
Atomica	Recebe as *threads* do programador, encapsula cada uma em uma *thread* externa (Capsula), cria um *ThreadGroup* com essas threads, cria uma ThreadMonitor para elas dentro desse *ThreadGroup* e inicia a execução de todas as *threads*.
Capsula	Recebe uma *thread* do programador e a coloca dentro do ThreadGroup. Inicia a *thread* do programador.
MO	Meta-classe responsável por monitorar exceções nas *threads'*

O programa deve criar suas classes e passá-las à classe Atomica. A seguir, a classe Atomica coloca cada uma dessas *threads* dentro de uma *thread* Capsula. Isso foi feito para que o programador não precise tornar cada uma de suas classes subclasse de *Thread* nem precise colocá-las explicitamente dentro do *ThreadGroup*, o que demandaria alteração no construtor de todas as classes, já que elas precisam ser associadas ao *ThreadGroup* pelo seu construtor. Essa forma não exige criação de *ThreadGroup* pelo programador, o qual nem precisa conhecer *ThreadGroups* ou alteração de construtores. Antes de criar as cápsulas, o objeto da classe Atomica pode salvar o estado dos atributos das *threads* em disco ou, através de reflexão computacional, obter seus atributos e valores correntes.

A classe ThreadMonitor recebe mensagens de cada uma das capsulas e decide se deve abortar ou simplesmente fechar o programa. As funções de restauração estão localizadas na classe Atomica. ThreadMonitor tem apenas função de decisão e ativação de funções de acordo com as decisões tomadas. Embora esta solução tenha se mostrado conceitualmente correta, sua implementação esbarrou inicialmente na falta de instrumentos Java para executá-la a contento, sendo em seguida solucionada pela utilização de técnicas reflexivas. Um programa que possua três *threads* que precisem ser executadas atomicamente, utilizando este mecanismo, teria o formato mostrado na figura 6.7:

```
public class cenario1_solucao1 {
public static void main(String[ ] argv)
{
Atomica atomica;

Thread matriz[]=new Thread[3];
matriz[0] = new ThreadUm( );
matriz[1] = new ThreadDois( );
matriz[2] = new ThreadTres( );
atomica = new Atomica(matriz); //  basta passar uma matriz
de threads para a classe
    // Atomica
}
}
```

FIGURA 6.7 – Exemplo da primeira solução

Para que as *threads* fossem executadas atomicamente, bastaria criar uma matriz de qualquer tamanho com as mesmas e passar tal matriz como parâmetro para um objeto da classe *Atomica*.

6.3.5 Implementação com sinalização de término de thread

A segunda implementação, embora contorne todos os problemas supra citados e consiga lograr êxito em seu intento, tornou o código dependente da aplicação, diminuindo consideravelmente não apenas o grau de reutilização, mas também a modularidade e simplicidade.

A tabela 6.2 mostra as classes desenvolvidas nesta implementação.

TABELA 6.2 – Classes da segunda implementação do cenário 1

Nome da Classe	Função
TheadUm	Thread qualquer do programador a ser atomizada
TheadDois	Thread qualquer do programador a ser atomizada
ThreadMonitor	Monitora o ciclo das threads do programador
Atomica	
Deposito	Serve de comunicação entre as classes. Foi necessário criar essa classe porque a associação deveria ser feita em nível de construtor entre as classes ThreadMonitor, Atomica e as threads (ThreadUm, ThreadDois), o que geraria uma declaração circular.

O programa (cenario1_solucao2.java) cria as threads a serem atomizadas e as envia para a classe Atomica, responsável pela sua execução e controle. As *threads* tiveram que ser alteradas para poderem se comunicar. Assim, seu construtor recebe o parâmetro Deposito. Essa classe é a responsável pelo recebimento da confirmação do término da *thread*. A *thread* ThreadMonitor monitora essa classe para ver se houve erro. O problema é não criar referências circulares (colocar uma ThreadMonitor ou Atomica no construtor das *threads* ThreadUm ou ThreadDois, porque neste caso as classes ThreadMonitor ou Atomica deveriam ser criadas antes de ThreadUm e ThreadDois, portanto não as conheceriam de antemão nem poderiam recebê-las como parâmetro, como acontece agora.

Quando uma *thread* acaba, envia um aviso para Deposito. ThreadMonitor descobre que alguma thread abortou lendo essas mensagens. Se só houver uma *thread* ativa (ThreadMonitor) e nenhuma mensagem de erro, o programa pode terminar. A classe Atomica já gravou o estado anterior de cada uma delas e tem como restaurá-las.O mesmo programa da solução anterior seria escrito na forma mostrada pela figura 6.8:

```
public class cenario1_solucao2  extends Thread{
public static void main(String[ ] argv)
{
Deposito   d = new Deposito();
ThreadUm   a = new ThreadUm (d);
ThreadDois b = new ThreadDois (d);
Thread matriz[] = new Thread[2];

matriz[0] = a;
matriz[1] = b;
Atomica atomica = new Atomica (matriz, d); //  além de uma
matriz de threads, é
  //   necessário passar uma instância da
  // classe Deposito  para a classe
  // Atomica
}
}
```

FIGURA 6.8 – Exemplo da segunda solução

6.3.6 Comparação

O desenvolvimento da segunda versão mostrou a dificuldade gerada pela impossibilidade de contar com mecanismos auto-configuráveis e genéricos, como os providos pela reflexão computacional. A codificação explícita limita a reutilização deste código, além de tornar sua manutenção extremamente onerosa. A partir do momento em que mecanismos transparentes e genéricos passam a ocupar maior espaço na implementação da solução, mais simples e reutilizável a mesma se torna.

A segunda solução exige que o programador acrescente determinado código a cada uma de suas *threads*, o que só pode ser feito com conhecimento prévio da solução, o que elimina sua transparência e diminui seu grau de modularização, além de afetar negativamente sua reutilização e simplicidade de uso.

A maior desvantagem da primeira solução é a sobrecarga causada pelo uso de reflexão e seu mecanismo de interceptação de mensagens.

6.3.7 Reutilização das técnicas empregadas

Esta solução empregou uma classe monitora cuja função é verificar a execução das outras *threads* envolvidas no processo. Este monitor deve conhecer os dados alterados durante a execução das *threads*. O uso de reflexão computacional permitiu que a mesma classe monitora pudesse ser reutilizada com quaisquer outras classes, pois a mesma pode, através de seus mecanismos de introspecção, adaptar-se em tempo de execução. Esta classe pode monitorar qualquer tipo e qualquer quantidade de objetos. Como cada elemento parte da solução está confinado em uma classe independente, a solução inteira ou apenas pedaços dela podem ser reutilizados. Já a segunda solução também pode ser reutilizada, desde que a aplicação cumpra uma série de pré-requisitos que, se não impedem sua reutilização, dificultam-na sobremaneira.

6.3.8 Conclusões

Entre as muitas dificuldades encontradas na implementação das soluções para este cenário, estão a já citada perda de controle das *threads* por parte do programador Java e da inexistência de interceptação ativa de mensagens nos protocolos de reflexão computacional estudados. O emprego do meta-protocolo permitiu que a primeira dificuldade fosse contornada. Em relação ao segundo problema, estuda-se a inclusão de um mecanismo de notificação nas próximas versões do protocolo Guaraná, embora não esteja prevista sua adoção de interceptação ativa. Todos estes mecanismos, porém, cobram em desempenho o que disponibilizam em termos de adaptabilidade, simplicidade e reutilização. Como alguns mecanismos reflexivos são a única forma de tornar as soluções reutilizáveis, deve-se então procurar uma utilização parcimoniosa destes.

6.4 Cenário 2 - Aplicações concorrentes com acesso competitivo aos dados

6.4.1 Descrição do cenário

O primeiro cenário apresentou como característica o fato do conjunto de dados acessados pelas *threads* ser disjunto, ou seja, não ocorrem problemas de inconsistência de dados devido a acesso concorrente, o que restringe a equação da atomicidade à recuperação de estados. O segundo caso de estudo e aplicação de técnicas reutilizáveis confronta-se com outro tipo de acesso aos dados da aplicação. Neste cenário, estes podem ser acessados por mais de um método concorrentemente, o que implica em novas demandas de controle de atomicidade. Além do aspecto de recuperação dos dados, no presente cenário lida-se com o controle de acessos concorrentes aos mesmos. Tal situação pode ser vista na figura 6.9.

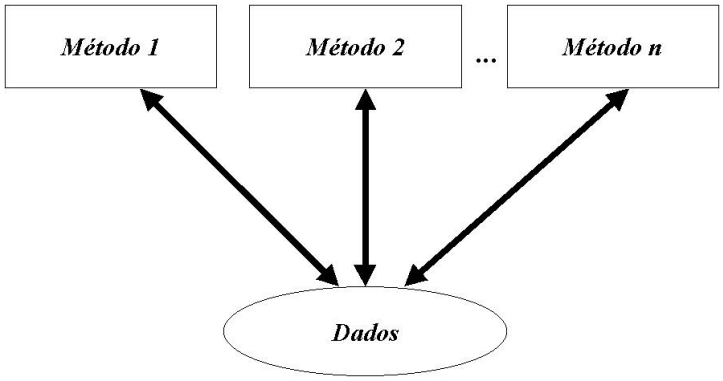

FIGURA 6.9 – Arquitetura do cenário

Neste caso, além de ter de perceber se todos os processos terminaram com sucesso ou não, também deve-se evitar a interferência entre os mesmos, garantindo seu isolamento, impedindo assim a ocorrência de estados inconsistentes nos objetos.

6.4.2 Soluções

O controle de concorrência empregado nesta solução utiliza bloqueio de acessos, ou seja, cria uma camada intermediária transparente à aplicação que se encarrega de responder por permissões e restrições aos pedidos de leitura/gravação. Para isso, empregam-se conceitos de transações atômicas. O conjunto de instruções colocado entre guardas é tratado atomicamente, ou seja, todas são executadas ou então a aplicação deve voltar à situação anterior. Há controle de concorrência e uma estratégia de recuperação de estados via *rollback* de dados.

Uma das preocupações quando da implementação de transações atômicas foi com o tempo de bloqueio dos dados e, mais do que isto, a granularidade deste bloqueio, que corrobora para este acréscimo. Embora tempos e granularidades maiores simplifiquem a implementação das soluções, diminuem consideravelmente seu desempenho e eficiência. Uma situação ideal apresenta curtos tempos de bloqueio de dados e pequena granularidade destes. Várias técnicas aqui descritas foram implementadas na busca de melhor desempenho quanto a estes dois fatores.

Para este cenário, duas formas de implementação de transações atômicas foram selecionadas no intuito de estudar sua eficiência e grau de reutilização: o emprego de métodos sincronizados e de guardas.

6.4.3 Transações atômicas com sincronização

Uma das maneiras de evitar que mais de um método acesse um determinado conjunto de dados ao mesmo tempo, comprometendo sua consistência, é a utilização de métodos sincronizados. Tais métodos garantem que, uma vez a *thread* tenha começado a acessar um dado compartilhado, poderá completar sua operação sem ser interrompida. Um método sincronizado em Java garante que este seja concluído antes que qualquer outro método sincronizado seja executado sobre o mesmo objeto. Há uma fila de *threads* esperando para serem executadas, com um escalonador decidindo sobre a próxima execução de acordo com as prioridades estabelecidas. Os métodos *wait( )* e *notify( )* (ou *notifyAll( )*) são utilizados neste mecanismo.

Quando duas *threads* têm acesso ao mesmo objeto e tentam modificar seu estado, pode haver interferência entre elas, com conseqüente inconsistência no estado do objeto, corrompendo-o. A solução é sincronizar as *threads*. O maior problema é que se esta sincronização não for realizada, as operações sobre os objetos não serão atômicas. Isto pode ocorrer quando o sistema operacional realiza *time-slicing* em *threads*. Como todos as situações devem estar previstas, deve-se considerar que as *threads* serão interrompidas freqüentemente. Para lidar com esta situação, Java apresenta monitores. Marcam-se operações compartilhadas por várias *threads* como *synchronized*. Quando uma *thread* entra em um método sincronizado, Java garante que esta poderá ser concluída antes que outra execute qualquer método sincronizado no mesmo objeto. Se uma classe possui métodos sincronizados, cada instância da mesma receberá uma fila que conterá todas as *threads* à espera para executar um dos métodos sincronizados [CON 98].

Esta primeira implementação do cenário 2 utilizou-se de *threads* sincronizadas. Tal solução cria diversas *threads* que acessam um mesmo conjunto de dados localizados em objetos. Para garantir que esses acessos ocorram atomicamente, é possível implementar os métodos responsáveis por alteração de dados como métodos sincronizados. Esta solução é eficaz em termos de controle de concorrência, porém pouco eficiente, pois apresenta elevada granularidade de dados, pois cada *Thread* pode bloquear todos os dados que necessita.

A figura 6.10 mostra a arquitetura da solução. Diversos métodos atuando de forma concorrente acessam os mesmos dados de uma aplicação. Pelo fato destes métodos serem declarados como sincronizados, eliminou-se a interferência entre os mesmos.

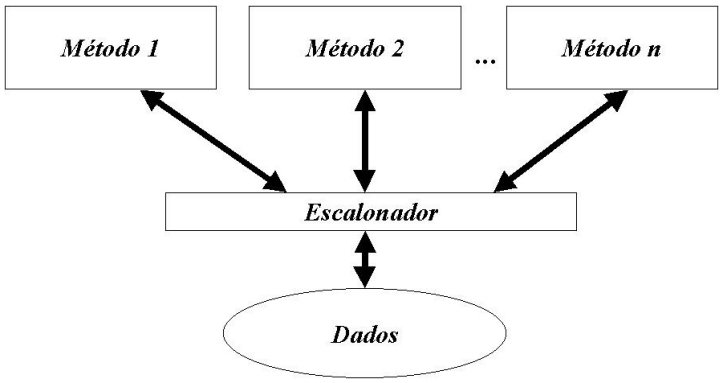

FIGURA 6.10 – Emprego de métodos sincronizados

6.4.4 Transações atômicas com guardas [3]

Esta técnica baseia-se em uma adaptação do sistema Kan [JAM 2000], que preconiza a utilização de estruturas chamadas de "guardas" para diminuir a granularidade e, consequentemente, o tempo de bloqueio dos dados. Ao invés do tempo de bloqueio ser a duração da *thread*, é possível isolar pequenos trechos de código para efeito de atomicidade. Desta maneira, vários "guardas" podem ser colocados dentro do código do procedimento a ser executado pela *thread*, sendo que apenas durante este período há necessidade de acesso exclusivo aos dados. Através da utilização de reflexão computacional, é possível também descobrir que dados estão sendo acessados pelas operações, diminuindo assim também a granularidade do bloqueio. Dentro de cada *thread* podem haver vários guardas independentes um dos outros.

A grande vantagem desta técnica sobre o uso simples de métodos sincronizados está não apenas no fato de possibilitar ao programador diminuir a granularidade de dados acessados, mas também permitir a existência de mais de um "guarda" no método, apenas durante os períodos críticos. Com isso, podem haver "janelas" de tempo nas quais o método em execução não está bloqueando nenhum dado, o que não aconteceria em um método sincronizado. A figura 6.11, adaptada de [JAM 2000], mostra um método qualquer que necessita executar um conjunto de operações atomicamente, porém nem todas as operações pertinentes a este método apresentam esta exigência, de modo que não há necessidade de manter os dados bloqueados durante toda sua execução.

```
public void metodoA ( )
{
... comandos;
atomico{
        ... comandos;
      }
...comandos;
}
```

FIGURA 6.11 – Exemplo de uso de guardas em Kan

A implementação desta técnica em Java pôde ser otimizada através da utilização de mecanismos reflexivos. A figura 6.12 mostra como o meta-nível monitora os objetos sendo acessados por *threads*. Os ganhos obtidos foram não apenas quanto ao aspecto de otimização, mas também quanto à transparência. A otimização foi causada pela diminuição da granularidade de bloqueio, pois guardas são elementos menores que *threads*. Podem haver inclusive vários guardas dentro da mesma *thread*. Com isso, o tempo e o conjunto de instruções que compõem uma transação atômica são menores. Quanto à transparência, o mecanismo reflexivo foi utilizado para lidar com guardas, liberando o programador de garantir atomicidade aos comandos internos a cada guarda.

[3] Do inglês *guard*

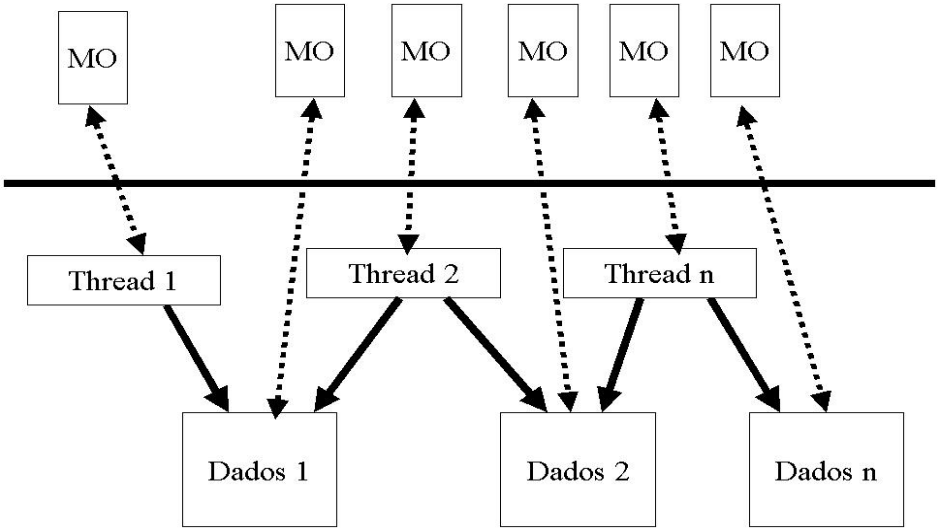

FIGURA 6.12 – Implementação de guardas com reflexão computacional

Este mecanismo pode ser estendido, permitindo a inserção de predicados internos aos guardas. Tais predicados possibilitam maior controle sobre a execução dos comandos internos, pois apenas quando satisfeitos permitem que as operações internas a eles sejam levadas a cabo. Assim, não apenas começamos e terminamos partes atômicas de código, mas também podemos colocar pré e pós condições de execução. Por exemplo, seria possível termos uma seção de código como mostra a figura 6.13:

```
atomico ( saldo>0);
comandos ...;
atomico ( saldo != 0);
```

FIGURA 6.13 – Exemplo de pré e pós-condições em Kan

Neste caso, os comandos só seriam executados se o saldo da conta fosse maior que zero. Ao final das operações, caso o saldo não seja diferente de zero, todas as operações seriam desfeitas, ou seja, a primeira condição (pré-condição) define o início das operações, enquanto que a segunda (pós-condição) define se essas operações serão permanentes ou sofrerão algum mecanismo de *rollback*.

A otimização desta técnica que pode ser obtida com a intervenção da reflexão computacional dá-se no sentido de prospectar que dados estão sendo operados por quais métodos. Uma chamada ao guarda pode ser interceptada e os dados a serem acessados durante esta chamadas serem monitorados a partir do meta-nível.

6.4.5 Descrição das implementações

Duas soluções foram implementadas para este cenário. A primeira, conforme descrito, utilizou-se simplesmente de *threads* sincronizadas e evitou interferência entre as mesmas, porém com tempos de bloqueio maiores que o da implementação de *threads* com guardas. Por ser uma implementação extremamente simples, não merecerá maiores comentários a respeito de sua codificação e suas classes, apenas será comparada com a outra solução.

A segunda implementação utilizou o conceito proposto por [JAM 2000] de guardas internos às *threads*. No cenário descrito na figura 6.14, uma ou mais *threads* acessam um conjunto de objetos. Dentro das *threads*, guardas indicam o início e o fim das transações atômicas. É possível, desta forma, diminuir a granularidade destas, pois não é necessário que toda a execução da *thread* seja executada de forma atômica. Além disso, podem existir dentro de uma *thread* mais de uma transação. Cada uma destas precisará bloquear apenas os objetos

que necessitará atualizar. Quanto menor for o bloco marcado como atômico, menor será a probabilidade de haver um grande número de objetos bloqueados e menor será seu tempo de bloqueio, o que permitirá maior grau de paralelismo.

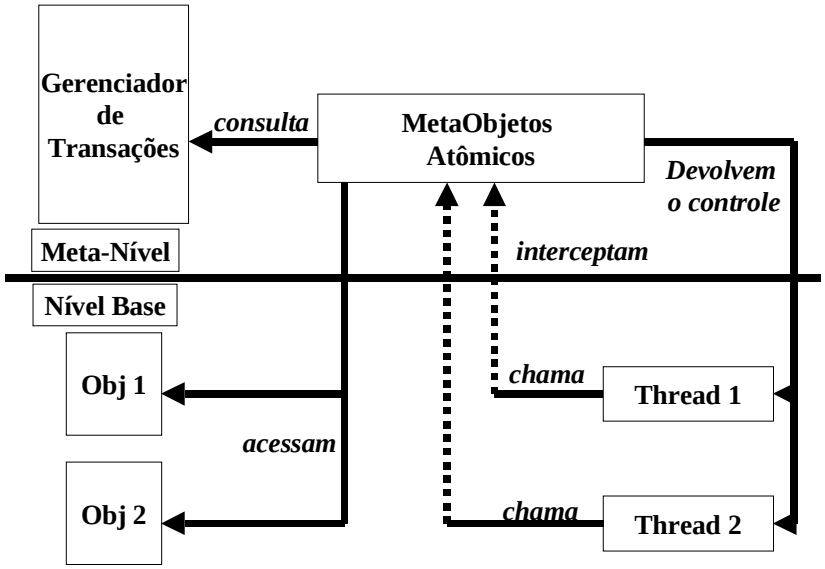

FIGURA 6.14 – Uso de threads com guardas

Quando uma *thread* tenta acessar algum dos objetos alvo, esta chamada é interceptada pelo meta-objeto atômico. Este atua junto com um gerenciador de transações no meta-nível que controla que *threads* estão utilizando que objetos. O gerenciador de transações é uma classe invisível aos objetos do nível base. Uma de suas funções, além de manter registros sobre os bloqueios correntes, é impedir o acesso indiscriminado por parte das *threads* sobre os objetos, mantendo assim a consistência destes. Desta maneira, uma *thread* só pode acessar um objeto que não esteja sendo usado por outra ou que já tenha sido bloqueado por ele mesmo. O gerenciador de transações mantém registro de todas estas operações.

A classe TA (Transação Atômica) guarda o estado original de cada objeto em uso, para eventual *rollback*, assim como uma coleção de métodos de manipulação de objetos. Ela é responsável pela parte inicial das operações. Recebe as *threads* participantes do programa, os objetos por elas acessadas e uma instância da classe Guarda, responsável pela implementação dos "guardas" previamente mencionados. Toda a criação de meta-objetos e sua subsequente associação aos objetos do nível base é feita por uma instância desta classe, que também dispara a execução das *threads* e realiza a reificação dos dados para o meta-nível e sua posterior restauração, caso necessário. A reificação é feita através de procedimentos reflexivos que possibilitam alto grau de reutilização, pois, devido à forma como foram concebidos, podem ser aplicados a qualquer tipo de objeto. Mesmo que vários objetos da mesma classe sejam utilizados, TA poderá reconhecê-los individualmente, pois trabalha com seus códigos Hash. Isto também diminui a necessidade de acesso direto a esses dados, o que poderia resultar em recursão infinita.

Quando uma *thread* chama o método definido para terminar sua transação, os estados dos objetos são confirmados, todos os bloqueios da *thread* são liberados e a mesma é excluída do processo. Uma mesma *thread* pode iniciar e terminar transações mais de uma vez dentro de seu ciclo de vida. Isto permite que, durante os períodos nos quais não há necessidade, ela não bloqueie objetos, o que aumenta o grau de execuções concorrentes de *threads* e, consequentemente, o desempenho do sistema, ao contrário de procedimentos sincronizados, os quais bloqueiam seus dados durante todo seu ciclo de vida.

Alguns dos problemas enfrentados durante a implementação deste cenário dizem respeito a conceitos não inerentes ao meta-protocolo em uso ou mesmo à reflexão computacional. Para que várias *threads* pudessem acessar as mesmas instâncias de uma ou mais determinada(s) classe(s), foi necessário que estas fossem passadas como parâmetros, o que reduziu a facilidade de uso. O ocorrência de recursão no meta-nível foi outra dificuldade encontrada. Uma das técnicas empregadas para diminuir tal situação foi a utilização dos códigos Hash dos objetos. O acesso a uma matriz de códigos Hash passou a substituir a necessidade de testes de verificação do objeto em uso corrente.

Outro problema enfrentado foi a dificuldade em descobrir a *thread* solicitante sem gerar recursão no meta-nível. Como as *threads* não necessitaram ser monitoradas a partir de meta-objetos próprios, foi possível utilizar o método *getThread* do parâmetro *Operation* do método *handle* no meta-objeto do objeto alvo. Caso a *thread* possuísse um meta-objeto associado, tal recurso tornar-se-ia inviável, devido a geração de recursão, pois a chamada ao método *getThread* seria interceptada pelo meta-objeto da *thread* que, por sua vez, chamaria novamente *getThread*..

TABELA 6.3 – Classes da segunda implementação do cenário 2

Nome da Classe	Função
TA	Classe que recebe as *threads* participantes e os objetos acessados e inicializa o ambiente atômico, criando os meta-objetos e associando-os aos objetos do nível base, além de disparar as *threads*. Possui uma cópia dos estados correntes e anterior dos dados. Reifica e restaura os valores dos campos dos objetos.
GerTran	Classe gerenciadora de transações. Encapsula os métodos para manipulação da lista de *threads* ativas e objetos bloqueados.
MO	Meta-objeto das classes CC e CP. Responsável pela interceptação de suas chamadas e pelo descobrimento da *thread* solicitante. Testa as permissões de bloqueios, inicia e termina transações quando algum acesso não permitido for tentado.
Guarda	Implementa os métodos dos guardas
T1, T2	Duas *threads* exemplo acessando os dados de CC e CP
CC, CP	Duas classes exemplo com dados a serem atualizados

Embora na definição do MOP Guaraná tenha sido aventada a possibilidade de incluir um campo *"Caller"* ao parâmetro *Operation* do método *handle*, tal não foi levado adiante devido a preocupações quanto a segurança. Uma maneira indireta de realizar isso é através da utilização de meta-objetos em cada *thread* candidata a realizar uma chamada a um método no objeto alvo. Caberia a este objeto interceptar chamadas de método e comunicar-se com meta-objetos de objetos alvo, informando-os destas chamadas. Os meta-objetos dos objetos alvo poderiam então, ao interceptarem mensagens para seus objetos base, comparar com as informações enviadas pelos meta-objetos das *threads* e descobrir qual realizou tal chamada. Caso o meta-objeto da *thread* receba o resultado da chamada ao método sem que os meta-objetos dos objetos alvo tenham realizado qualquer interceptação, aquele pode alertá-los para o fato de que a chamada não os tinha como alvo.

Também pode ser usado o método *Guarana.broadcast* para enviar solicitações aos meta-objetos das *threads*, através da interface *OperationFactory*. Este método permite a comunicação entre meta-objetos.

6.4.6 Implementação

Para que o programador possa beneficiar-se desta solução, basta criar uma nova instância da classe TA, passando para a mesma em seu construtor uma matriz de *threads* participantes e uma matriz de objetos acessados (dados), além de uma instância da classe Guarda, representada na linha abaixo pela variável "g".

ta = new TA(matriz,dados,g); // cria uma nova instância gerenciadora de transações

O objeto "ta" executará todas as *threads* e controlará sua concorrência e recuperação, garantindo que serão executadas de forma atômica. A seguir, uma descrição das funções mais importantes das três classes principais desta implementação: a meta-classe MO, responsável pela interceptação de mensagens e todo o trabalho no meta-nível, a classe TA, responsável pela garantia de atomicidade, e a classe GerTran, gerenciadora de transações.

A classe MO atua no meta-nível, utilizando as outras classes do mecanismo e sendo responsável por grande parte das tarefas executadas neste cenário (fig. 6.15):

```
public class MO extends MetaObject
{
private GerTran gt;                  // gerenciador de transações
private Integer para_objeto;     // variável local para
armazenar código hash do objeto base private TA ta;
// transação atômica ativa
Object dados;                        // usado para reconfigurar o
MO

public  MO(GerTran gt_, int ID, TA ta_,Object objeto_base)
// construtor da meta-classe
// recebe os participantes da transação

public Result handle (Operation op, Object o)
{
Thread t = op.getThread(); // descobre qual thread está
acessando o objeto base

// usa operações das classes GerTran e TA

// se for transação nova, inseri-la na lista
// se for nova tentativa de iniciar a mesma transação, não
permitir

// se for fim de guarda, terminar a transação e liberar
bloqueios dos objetos

// se for uma acesso a um objeto do nível base
// se não for feito por uma thread já iniciada, abortá-la e
desfazer suas alterações

// se o objeto que a thread quer usar está livre,
//                  a) inseri-lo na lista de objetos bloqueados
//                  b) salvar estado do objeto
```

```
// se o objeto já estiver bloqueado, testa para ver se não foi a
própria thread que o fez
// se não foi, termina transação
```

FIGURA 6.15 – Interface da classe MO

A interface da classe TA apresenta os métodos necessários para a restauração e reificação dos estados dos objetos do nível base (fig. 6.16):

```
public class TA
{
MO[ ] mo;                    // matriz de meta-objetos
GerTran gt;                  // gerenciador de transações
guarda g;                    // instância de Guarda
Object[ ][ ] reserva;        // cópia anterior dos dados originais
Object[ ] original;          // cópia corrente dos dados originais
Integer[ ] codigos_hash;   // matriz de códigos hash dos
objetos do nível base

public TA (Thread t[ ], Object[ ] ob, guarda g_)
// construtor da classe
// armazena os estados iniciais dos objetos
// guarda os códigos hash dos objetos do nível base
// inicializa o gerenciador de transações
// cria lista de meta-objetos participantes
// associa os meta-objetos aos objetos base e ao gerenciador
de transações
// inicializa as threads participantes

//---------funções utilizam reflexão para realizar clonagem
genérica----
public void reificar_dados()
// reifica os estados de todos os objetos do nível base

public void reifica (Object origem)
// reifica o estado do objeto do nível base passado como
parâmetro

public void restaura_dados()
// restaura todos os dados de todos os objetos do nível base,
chamando a função restaura
// usado para terminar uma transação
public void  restaura (Object origem)
// restaura o estado anterior do objeto do nível base passado
como parâmetro

private int achaObjeto (Integer codigo_hash)
// busca um objeto na lista de objetos bloqueados
```

FIGURA 6.16 – Interface da classe TA (Transação Atômica)

A classe GerTran é responsável pelo gerenciamento das transações, armazenando os objetos bloqueados e seus bloqueadores, bem como dando permissão de acesso ou não aos mesmos e encerrando transações falhas (fig. 6.17).

```
public class GerTran
{
private Vector transacoes;  // vetor de transações
private Vector objetos;     // vetor de objetos do nível base
private Vector ativos;      // vetor de threads ativas
public GerTran()
// construtor da classe, inicializa os vetores
public void insere(Thread t, Object hash_codigo)
// insere thread e objeto bloqueado
public void insere(Thread t)
// insere thread na lista de threads participantes
public int achaThread(Thread t)
// descobre se a thread esta na lista de bloqueantes
public int achaAtivo(Thread t)
// descobre se a thread eh uma thread participante
public int achaPosDados( Integer ID)
// procura um objeto na lista de hashcodes
public void eliminaAtivo(Thread t)
// retira a thread da lista de participantes
public void eliminaThread(Thread t)
// libera todos os bloqueios da thread
public Object pegaBloqueante (Object codigo_hash)
// descobre que thread está bloqueando o objeto
public void terminaTransacao(Thread t)
// finaliza transação atômica
```

FIGURA 6.17 – Interface da classe GerTran (Gerenciador de Transações)

6.4.7 Comparação

Percebeu-se com clareza que o grande problema dos mecanismos de controle de concorrência que utilizam bloqueio de dados é sua perda de performance devido à espera impetrada a outros processos concorrentes que desejam acessar os mesmos dados. Quanto menor a granularidade de bloqueio obtido e o tempo requerido, maior a eficiência desse mecanismo. Sob este prisma, *threads* sincronizadas apresentam maior granularidade de bloqueio de dados, ao contrário das *threads* com guardas que, por ficarem restritas a um número menor de operações de cada vez, podem necessitar de menor quantidade de dados em menor fatia de tempo. Mesmo que as *threads* com guardas acabem usando a mesma quantidade de dados, suas solicitações de bloqueio são parciais, ou seja, bloqueiam apenas os dados necessários em determinado instante, deixando para posterior alocação os que for precisar mais em próximas operações. Além disso, o tempo de bloqueio das *threads* com guardas é menor, pois seu número de operações é potencialmente menor, sem mencionar a flexibilidade possibilitada pelas pré e pós condições. Considerando-se a sobrecarga inerente ao uso de reflexão computacional, obteve-se melhores respostas no uso de guardas quando os conjuntos de dados acessados pelas diferentes *threads* concorrentes foram mais disjuntos.

A desvantagem do uso de guardas foi o aumento no número de ações atômicas, o que, quando do uso de bloqueio como técnica de controle de concorrência, pode levar a muitas reinicializações.

6.4.8 Reutilização das técnicas empregadas

A implementação de guardas pôde ser realizada de forma reutilizável devido ao alto grau de adaptabilidade alcançado com a utilização de reflexão computacional. Cada chamada aos guardas pode ser interceptada e o todo tratamento de concorrência e recuperação executado independe das classes dos objetos acessados. Há várias opções a serem exploradas, porém todas passíveis do mesmo grau de reutilização devido à linha de ação adotada.

A aplicação que utiliza os guardas não precisa ser alterada quando de eventuais acréscimos de novas classes ou métodos no meta-nível. A sintaxe básica é simples e é o único conhecimento extra exigido do desenvolvedor da aplicação. Todos os aspectos relativos às transações são isolados em classes no meta-nível, sendo completamente transparentes à aplicação. Podem, além disso, lidar com qualquer classe sem que isso exija adaptação alguma seja no código da aplicação, seja no código das meta classes.

6.4.9 Conclusões

Uma observação que pôde ser feita a partir da implementação de guardas foi a possibilidade de introdução de pré e pós condições de execução. Na própria chamada ao guarda pode-se incluir uma ou mais condições como parâmetro. O guarda apenas irá iniciar a transação se a condição for verdadeira. Da mesma forma, no término da transação pode-se acrescentar uma ou mais condições no guarda. Caso esta condição não seja atendida, a transação pode ser desfeita, num processo semelhante a um *rollback* (fig. 6.18):

```
...
iniciar (condição);
...
...
terminar (condição);
```

FIGURA 6.18 – Condições inicial e final

As únicas alterações a serem feitas no caso de suporte a guardas com condições no início e no fim da transação devem ser feitas na implementação da classe Guarda, o que não afeta o código desenvolvido na aplicação.

6.5 Cenário 3 - Meta-Classe Serializadora

6.5.1 Introdução

Algoritmos baseados em ordenação [ROS 78] dão números de ordem para cada transação, os quais serão usados como ordem de execução. Cada transação em um número e os eventuais conflitos são resolvidos de acordo com os mesmos. Pedidos de leitura e gravação têm seus números comparados com os números das transações que acessaram os dados solicitados e, a partir desta comparação, o algoritmo permite ou não seu acesso [BER 81].

Há várias abordagens, cada uma favorecendo um aspecto diferente em sua implementação. Do ponto de vista de tolerância a falhas, o que deve ser priorizado é o controle da integridade e consistência dos dados, que podem ser logradas através da técnica do bloqueio em duas fases ou do número de ordenação [AZE 96].

6.5.2 Descrição do cenário

Os objetos componentes de uma aplicação são acessados por outros objetos da mesma aplicação ou de aplicações clientes que possam eventualmente estar necessitando de serviços fornecidos por sua interface. Esses acessos podem acontecer de qualquer modo, cronologicamente falando. Isto significa que o objeto está à mercê de chamadas, concorrentes ou não, de vários outros objetos de maneira tal que maneira que torna possível o aparecimento de estados inconsistentes no objeto causados pela falta de controle nestes acessos (fig. 6.19). O problema é evitar que o objeto atinja um estado de inconsistência devido a interferências entre os processos que o acessam. Este mecanismo pode ser utilizado no controle de concorrência de uma transação atômica.

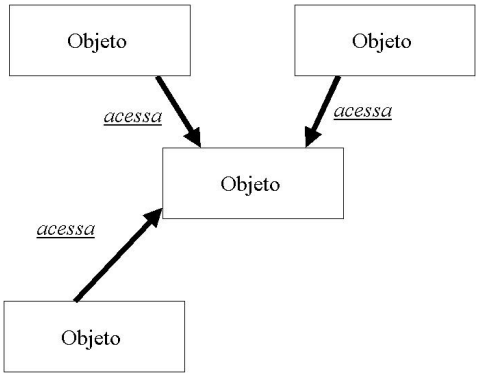

FIGURA 6.19 – Acessos livres a um objeto

O objeto alvo, para evitar a situação descrita, precisa implementar um controle de próprio de concorrência de acessos a seus atributos. Tal solução implica em duplicação de código e aumento na complexidade e tempo de desenvolvimento do mesmo. Uma forma simples de solução altamente reutilizável é o emprego de meta-classes com esta função.

6.5.3 Arquitetura da classe

A meta-classe serializadora (fig. 6.20) assume o controle do acesso ao objeto ou objetos em questão, fazendo com que todos os acessos ao(s) mesmo(s) sejam de seu conhecimento. Nenhum acesso passa a ser feito diretamente sobre o objeto base. Ao invés disso, são enfileirados pela instância desta meta-classe associada a ele. O emprego de configurações múltiplas de meta-objetos permite que implementações de diferentes políticas sejam utilizadas neste controle.

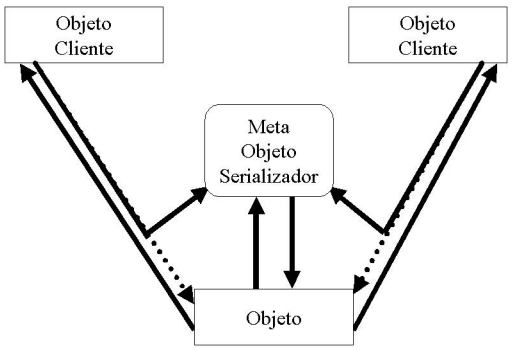

FIGURA 6.20 – Meta-Classe serializadora

Em uma implementação normal de controle de concorrência, o objeto ou seu mecanismo de controle deve decidir no momento do recebimento do acesso pela permissão ou não deste. Algumas políticas usam marcadores de tempo para tal. Uma das possibilidades abertas pela utilização desta meta-classe é o não rechaço imediato da tentativa de acesso. Ao invés de processar todas as operações imediatamente após sua chegada, o meta-objeto pode criar um

spooler interno (fig. 6.21) com mecanismo de ordenação de modo a permitir que todas as operações sejam executadas.

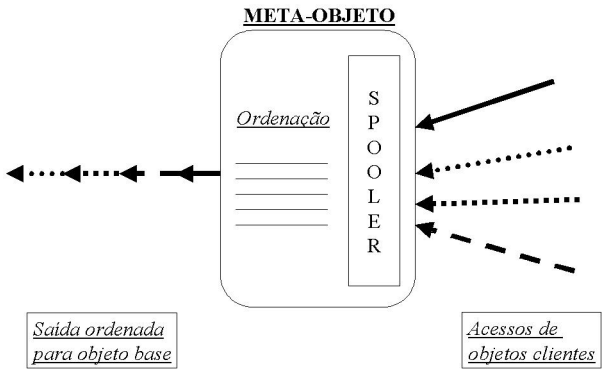

FIGURA 6.21 – Ordenação de chamadas

6.5.4 Comparação

Já foi vista a implementação reflexiva de controle de concorrência utilizando-se técnicas de bloqueio. Esta solução apresenta vantagens e desvantagens, as quais devem ser analisadas de acordo com cada contexto. Políticas de controle de concorrência que busquem diminuir a quantidade de bloqueios podem ser alternativas importantes em certas situações. A grande vantagem da implementação com meta-objetos é que estes, por não lidarem com classes específicas nem estarem presos a uma única política de controle de acessos, permite que esta solução seja altamente reutilizável.

6.5.5 Outras utilizações

Além do já citado uso para implementar algoritmos que utilizam mecanismos de *timestamping*, a meta-classe também pode ser útil com algoritmos baseados em ordenação [ROS 78], que dão números de ordem para cada transação, os quais serão usados para determinar sua ordem de execução. Cada transação apresenta um número e os eventuais conflitos são resolvidos de acordo com os mesmos. Pedidos de leitura e gravação têm seus números comparados com os números das transações que acessaram os dados solicitados e, a partir desta comparação, o algoritmo permite ou não seu acesso [BER 81].

6.5.6 Reutilização das técnicas empregadas

A classe responsável pela serialização de acessos a um objeto pode ser reutilizada realizando esta mesma função com qualquer outra classe. Seu emprego, contudo, pode ser mais abrangente em situações onde possa ser combinado com outras classes, pois sua única preocupação é o controle de concorrência.

6.6 Cenário 4 – Grupos de Meta-Objetos Dinamicamente

Reconfiguráveis

6.6.1 Introdução

Uma das características mais importantes que aplicações tolerantes a falhas devem apresentar é suporte a atomicidade. Os problemas que desenvolvedores de tais programas devem enfrentar são diversos e podem variar de acordo com o contexto do ambiente da aplicação. Este cenário emprega diversas técnicas na construção de um conjunto de meta-classes como uma camada adicional para monitorar os objetos da aplicação, no intuito de conferir suporte a atomicidade de forma transparente, dinamicamente reconfigurável e reutilizável. As técnicas utilizadas são discutidas, procurando-se definir o papel de cada uma das mesmas na introdução de atomicidade ao mesmo tempo em que omitem-se os detalhes de implementação da aplicação. Busca-se independência das classes da aplicação e capacidade de auto-reconfiguração dinâmica para que a mesma possa lidar com as mudanças no ambiente sem que isto demande em alterações em seu código, ou mesmo que seus módulos percebam tais eventos.

Atomicidade é um dos requisitos não funcionais mais importantes que um desenvolvedor deve ter em mente quando se trata de implementar aplicações confiáveis. A diversidade de situações e contextos, com a conseqüente enorme quantidade de demandas que as mesmas introduzem, é um dos maiores problemas a serem enfrentados. É seguramente uma tarefa árdua tentar produzir código que cubra diferentes tipos de aplicações e suas demandas específicas usando métodos tradicionais. No intuito de construir soluções para um maior número de situações sem aumentar o grau de complexidade do desenvolvimento de aplicações, é necessário empregar novas técnicas que auxiliem no aumento da flexibilidade e adaptabilidade da aplicação às mudanças no contexto de seu ambiente. Este esforço também leva à construção de soluções mais reutilizáveis.

Devemos prover uma aplicação com meios para que a mesma seja capaz de se auto-monitorar e, de acordo com o contexto corrente, realizar alterações para melhor lidar com os novos requisitos apresentados. Uma questão a ser respondida quando da execução desta tarefa é como incluir em uma aplicação tais características de forma transparente. Isto não deve aumentar o grau de complexidade para o desenvolvedor. Introspecção, auto-reconfiguração dinâmica e transparência são conceitos chave na obtenção deste objetivo. Este cenário mostra uma combinação de diversas técnicas na construção de uma camada intermediária transparente, flexível, auto-reconfigurável e altamente adaptável para dar suporte a atomicidade e, objetos de uma aplicação, permitindo que a mesma seja reutilizada sem exigir qualquer alteração na aplicação, ou mesmo que a mesma tenha ciência de sua existência.

6.6.2 Reflexão Computacional e Reconfiguração Dinâmica

Aplicações dinamicamente reconfiguráveis permitem modificações em sua arquitetura sem afetar seus módulos adjacentes. Isto não é possível em configurações estáticas, porque estas agrupam seus componentes em tempo de compilação. Configurações dinâmicas, entretanto, precisam de componentes extras responsáveis pela carga e destruição destes objetos adicionais. Esse é o preço a ser pago na obtenção de maior flexibilidade [ROM 2000].

O emprego de reflexão permite a inspeção de mudanças no ambiente computacional com conseqüente adaptação da aplicação a estas através da modificação ou alteração de um método, ou até pela adição de novos [BLA 99]. Neste cenário, isto é exemplificado pelas alterações que as políticas de controle de concorrência e recuperação sofrem de acordo com o contexto do ambiente da aplicação. A arquitetura aqui introduzida foi desenvolvida tendo classes em Java [SUN 99] e o protocolo de meta-objetos Guaraná [OLI 99a][OLI 99b] em mente.

O conjunto de métodos de um objeto pode ser inspecionado e modificado dinamicamente. Até a classe de um objeto pode ser alterada em tempo de execução. Também é possível adicionar ou excluir não apenas métodos, mas também atributos de um objeto [COS 99]. Novos serviços podem ser adicionados a um objeto usando seu meta-objeto para inserir novos métodos [COS 99]. Este papel é desempenhado pela reflexão computacional quando empregada na construção de aplicações dinamicamente reconfiguradas.

Reflexão computacional é um caminho natural para obter-se reconfiguração dinâmica. Mais transparência e reuso podem ser logrados através da utilização de classes no meta-nível, que é o local mais apropriado para as mesmas.

O problema a ser enfrentado neste cenário é garantir controle de concorrência e recuperação de estados a objetos de forma transparente e, principalmente, auto-reconfigurável. A implementação a seguir tem como principal meta ser dinamicamente adaptável às variações do ambiente em tempo de execução.

6.6.3 Um Middleware reflexivo

Middlewares são serviços que residem em uma camada intermediária entre a aplicação e o sistema operacional, no intuito de facilitar o desenvolvimento e gerenciamento de aplicações distribuídas [COU 00]. Estas aplicações são muitas vezes heterogêneas, de forma que um dos aspectos mais importantes a serem considerados na construção de *middlewares* é como mascarar tal heterogeneidade, fornecendo interfaces e métodos padrão de acesso.

Para executar tal tarefa de forma apropriada, um *middleware* deve ser capaz de reconfigurar seus componentes em tempo de execução, de acordo com os novos contextos nos quais estiver inserido. Há uma tendência de aplicação da tecnologia de componentes de processamento distribuído (CORBA, JavaRMI e DCOM). Estas tecnologias são usadas em nível de aplicação. Entretanto, seu uso pode ser estendido a *middlewares* [COU 00].

Middlewares devem ser capazes de adaptar-se de acordo com o contexto e as mudanças passíveis de surgimento em tempo de execução. Para possibilitar tal reconfigurabilidade e adaptabilidade, devem ser construídos utilizando-se engenharia aberta e permitindo inspeção e manipulação dinâmicas de seus componentes. Coulson [COU 00] sugere o uso de reflexão com uma forma de prover a necessária abertura exigida pelo *middleware* sem comprometer sua integridade. O uso de reflexão na construção de uma camada entre a aplicação e os módulos de suporte a seus requisitos não funcionais pode trazer novas possibilidades no que concerne a adição de flexibilidade e adaptabilidade.

Reflexão torna a aplicação mais adaptável a seu ambiente e mais capaz de lidar com mudanças inesperadas no mesmo. [BLA 99] considera a camada intermediária (um *middleware*) o local mais correto para situar a reflexão em um ambiente distribuído. Esta combinação também tem como conseqüência tornar tal camada mais apta a lidar com um maior número de demandas, especialmente quando se lida com requisitos não funcionais, pois é muito difícil abranger tais requisitos com uma única implementação. O uso de reflexão computacional torna as aplicações menores e mais livres de restrições impostas por políticas estáticas de controle de concorrência e recuperação.

O desenvolvimento de *middlewares* é tarefa árdua para os programadores. Uma solução natural é a introdução de reflexão para ajudar a suportar a configuração de novos serviços para a aplicação [BLA 99]. Middlewares construídos sem tais considerações em mente apresentam limitações a sua flexibilidade, pois devem ser capazes de detectar alterações no ambiente e customizar a si próprios de forma a acomodar tais situações [ROM 2000]. É extremamente difícil desenvolver um conjunto fixo de políticas e mecanismos para lidar com aspectos

concernentes a atomicidade e suas demandas. Estratégias diferentes podem ser carregadas quando do início da execução da aplicação e trocadas mais adiante. Isto deve ser feito, entretanto, de modo a não tornar a aplicação grande ou complexa demais. Para que tal não ocorra, instâncias de classes podem ser criadas apenas quando necessárias e destruídas imediatamente após o término de sua utilidade.

O uso de configurações múltiplas permite conexões ponto-a-ponto entre meta-objetos e objetos no nível base, o que leva a um controle maior sobre o suporte provido pelo *middleware*, ou na forma de relacionamentos um-para-muitos, mais útil em situações onde uma única ação deve atingir vários objetos simultaneamente. *Middlewares* reflexivos permitem a interoperação de dispositivos heterogêneos, porém é preciso fornecer mecanismos para disparar as adaptações que eventualmente se façam necessárias.

6.6.4 Agrupando Meta-Objetos

Grupos de objetos (fig. 6.22) são conjuntos de elementos computacionais semelhantes organizados para cooperar em certas tarefas ou para prover redundância. O grupo é externamente visto como um único objeto, atuando e sendo percebido como uma única entidade. Um de seus possíveis usos é no suporte a atomicidade. Objetos confiáveis podem ser obtidos via redundância, alcançada por diversidade de projeto ou replicação [HAE 96]. Desta forma, diferentes objetos implementando diversas políticas podem ser unidos para formar um grupo coordenado por um objeto que inclua algum mecanismo de decisão. Outra forma é replicar os objetos e usar um coordenador para conectá-los a seu grupo de réplicas. Operações que forem enviadas a um objeto são desviadas para suas réplicas e, através de um mecanismo de votação implementado no objeto coordenador, os resultados provenientes de cada réplica são testados e um escolhido.

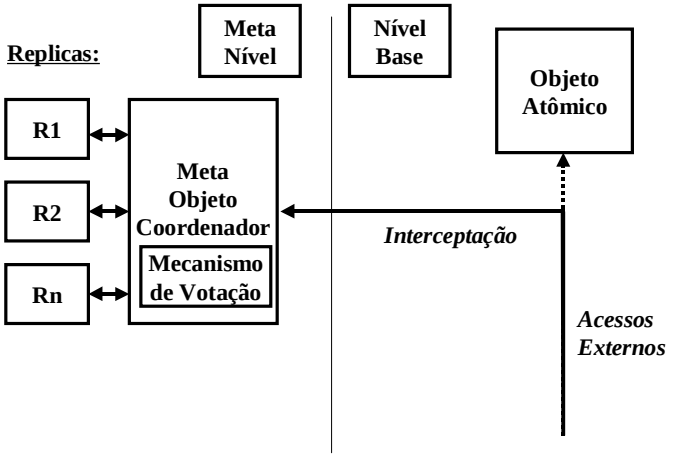

FIGURA 6.22 – Grupos de meta-objetos

A coordenação do grupo pode ser executada usando-se meta-objetos. No MOP Guaraná, uma instância da meta-classe Composer delega a execução de serviços a outros meta-objetos[OLI 99a][OLI 99b]. Este se comporta como coordenador dos meta-objetos membros de uma meta-configuração múltipla. Pode haver mais de um meta-objeto com a mesma função. Cada um pode ter sido implementado de forma distinta, caracterizando assim diversidade de projeto. A instância de Composer pode enviar uma operação qualquer a todos os meta-objetos ligados a si e tomar decisões para aumentar a confiabilidade do resultado comparando as respostas obtidas. Este procedimento pode ser usado para gerar um histórico e, dinamicamente,

excluir meta-objetos criados com problemas de projeto. Estes meta-objetos também podem ser construídos implementando diferentes políticas e algoritmos, permitindo a instância de Composer escolher os mais apropriados para cada situação. Novos meta-objetos podem ser acrescentados dinamicamente ou ainda excluídos da meta-configuração (fig. 6.23).

O MOP Guaraná fornece algumas meta-classes que podem ser úteis na composição de configurações múltiplas, o que implica em aumento de sua efetividade e reuso, pois novas a adição e exclusão de objetos não exige alterações na aplicação no nível base[OLI 99b]. Mecanismos de decisão movidos para o meta-nível podem monitorar e selecionar as meta-classes disponíveis, liberando a aplicação de tal tarefa. Isto leva a soluções mais genéricas e reutilizáveis.

Diversos mecanismos de tolerância a falhas podem ser implementados utilizando-se o conceito de grupos de objetos, entretanto se estes forem colocados no meta-nível, seu reuso pode ser dramaticamente aumentado. Grupos de objetos podem ser usados para implementar diversas técnicas relativas a um único propósito, ou então compor um grupo de instâncias de uma única classe com um mecanismo de votação. A mesma operação é submetida a todas estas instâncias e o objeto implementando o mecanismo de votação recebe seus resultados e chega a uma conclusão sobre sua validade. Estes objetos podem estar distribuídos, o que significa que algum eventual problema de conexão apenas retirará um objeto de ação, mas o conjunto poderá continuar o trabalho sem maiores prejuízos. Qualquer falha pode ser percebida pelo resultado diferentes ou falta do mesmo. Esta configuração com instâncias da mesma classe não é suficiente para lidar com problemas de falhas de projeto. Neste caso, uma solução melhor é desenvolver classes diferentes com diferentes mecanismos. Uma classe que implemente um mecanismo de votação ainda será necessária.

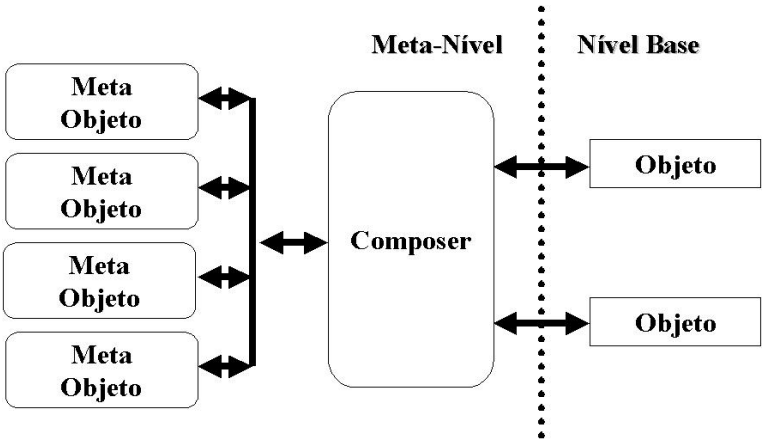

FIGURA 6.23 – Metaconfigurações múltiplas

6.6.5 A Implementação

Meta-informações devem ser representadas e armazenadas de forma a serem facilmente acessadas pelos mecanismos reflexivos. Um modo natural é criar um repositório no meta-nível usado pelos meta-objetos quando da reificação dos objetos do nível base. Isto também pode garantir consistência entre meta-objetos relacionados [COS 2000b].

A implementação sugerida neste cenário lida com os três principais problemas normalmente encontrados no desenvolvimento de *middlewares*: extensibilidade, adaptabilidade e reuso. É extensível porque novas meta-classes podem ser adicionadas a qualquer momento, sem que a aplicação necessite ficar ciente de tal fato. É adaptável porque usa reflexão para lidar com mudanças no ambiente da aplicação, ou ainda com novos requisitos da própria. Finalmente, é reutilizável porque é mantida separada dos módulos da aplicação e pode ser empregada sem

prévio conhecimento das classes desta. Este resultado é obtido utilizando-se reflexão no modelo orientado a objetos.

6.6.6 A Visão do Meta-Nível

A arquitetura mostrada na figura 6.24 exemplifica como os conceitos discutidos neste cenário podem trabalhar juntos. O objeto da aplicação (objeto de nível base) envia uma chamada a uma instância da classe MetaFront, que dispara uma série de operações que conectam esse objeto ao resto da arquitetura de suporte a atomicidade. Tal suporte é totalmente transparente ao objeto. A arquitetura pretende lograr expansibilidade, flexibilidade, adaptabilidade e reuso.

Este modelo é expansível, pois novas classes encapsulando novas políticas de controle de concorrência e recuperação podem ser incluídas ou excluídas a qualquer momento. É flexível, pois tais políticas podem ser mudadas dinamicamente, adaptando os objetos do nível base aos novos contextos da aplicação. A adaptabilidade é maior devido a possibilidade de alterar o comportamento da própria aplicação em tempo de execução através do uso de técnicas reflexivas. O reuso é obtido evitando-se a utilização de código mencionando explicitamente qualquer classe. Ao invés disso, reflexão é empregada para garantir introspecção em tempo de execução. Desta forma, não é necessário qualquer conhecimento prévio das classes da aplicação.

6.6.7 A Arquitetura do *Middleware*

Para lidar com diferentes políticas de controle de concorrência e recuperação, há um repositório de classes onde novas políticas podem ser inseridas e excluídas. Instâncias de uma classe especial chamada Loader são responsáveis pela manutenção destas políticas de forma organizada e passíveis de utilização pelo resto do mecanismo. As classes Policy Manager (Concurrency Policy Manager e Recovery Policy Manager) enviam solicitações às instâncias de Loader cada vez que uma nova política precisa ser inicializada. Cada Policy Manager, entretanto, deve trabalhar em sintonia com as classes Analyzer (Concurrency Analyzer e Recovery Analyzer). Estas implementam algoritmos especiais e possuem mecanismos de decisão para sugerir uma política a ser adotada. Elas são consultadas pelo Policy Manager cada vez que uma instância de Configurator solicitar uma nova política. A instância de Analyzer também informa ao Configurator que política será adotada. Instâncias de Concurrency Policy e Recovery Policies são então criadas para cada novo objeto no nível base que necessite-as. A instância de Configurator mantém registro de cada meta-objeto da classe MO e seu objeto base associado. De acordo com a política em uso, ativa a instância de Reifier, que armazena os objetos de nível base em una instância de Repository, uma estrutura de dados responsável por manter os dados de cada objeto base reflexivo no meta-nível, para que seja possível recuperá-los caso necessário.

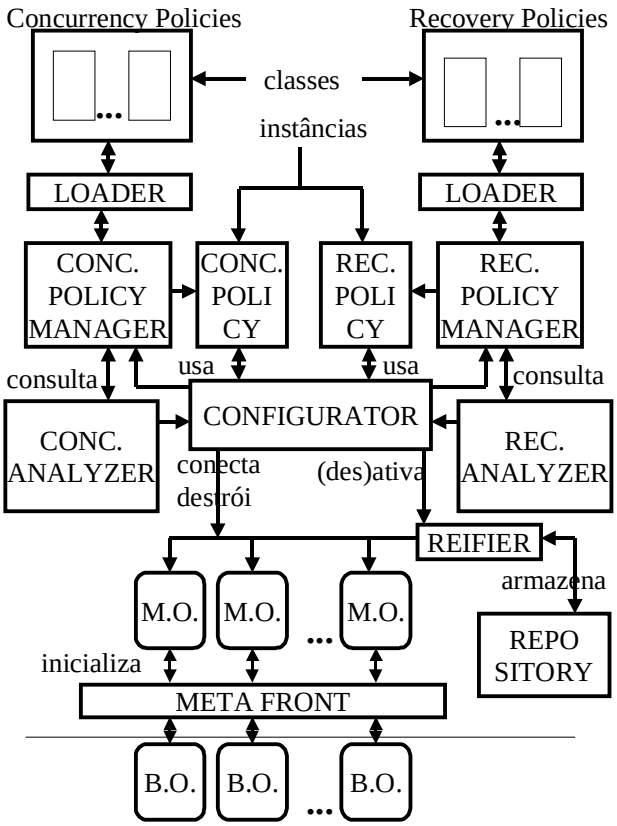

FIGURA 6.24 – Arquitetura do meta-nível

Para simplificar seu uso, há apenas uma “classe de entrada” nesta arquitetura: a MetaFront. Tudo que um objeto de uma aplicação (objeto de nível base) que precise de suporte a atomicidade deve fazer é chamá-lo. Esta classe contém métodos para criar instâncias de MO e então conectá-la ao objeto base solicitante. Deste momento em diante, a instância de MO associada ao objeto base interceptará e monitorará a estrutura e o comportamento deste objeto.

Outra característica que deve ser destacada é o fato de cada ação atômica poder implementar diferentes políticas de controle de concorrência e recuperação, de acordo com a situação corrente. Desta forma, a mesma execução de uma aplicação pode apresentar diferentes respostas ao mesmo problema, sendo que esta será sempre a melhor para o momento.

Tudo que o objeto base precisa fazer para usar a interface é conectar-se a um novo meta-objeto (tal fato é transparente a ele, pois uma nova instância de MO é automaticamente gerada em tempo de execução). O objeto obtém então uma política de controle de concorrência e uma de recuperação, que são escolhidas para ele no meta-nível. Quando a ação atômica chega a seu fim, a instância de Configurator termina a conexão inter-níveis e a instância de MO. Durante seu ciclo de vida, este meta-objeto é responsável pela monitoração e ligação entre o objeto base e o meta-nível. Dados podem ser reificados ou retornados do meta-nível de acordo com as políticas em uso. Tudo é transparente ao objeto base e pode ser usado com qualquer objeto de qualquer classe, pois não exige nenhum tipo de conhecimento prévio. A interface também apresenta métodos para iniciar e terminar uma ação atômica.

A figura 6.25 mostra como implementar redundância ou diversidade de projeto usando um mecanismo de votação. O objeto base é monitorado por um meta-objeto (MO). Esta associação é controlada pela instância de Configurator. Quando uma chamada é enviada para o objeto base, seu meta-objeto a intercepta e envia para o Configurator, o qual, de acordo com as políticas previamente selecionadas, a passa para um grupo de objetos implementando ou redundância ou diversidade de projeto. A instância de Voter possui um mecanismo de decisão

que descobre o correto processamento da chamada. Cada resultado é colocado no repositório Hits History para que instâncias com problemas possam ser excluídas da configuração.

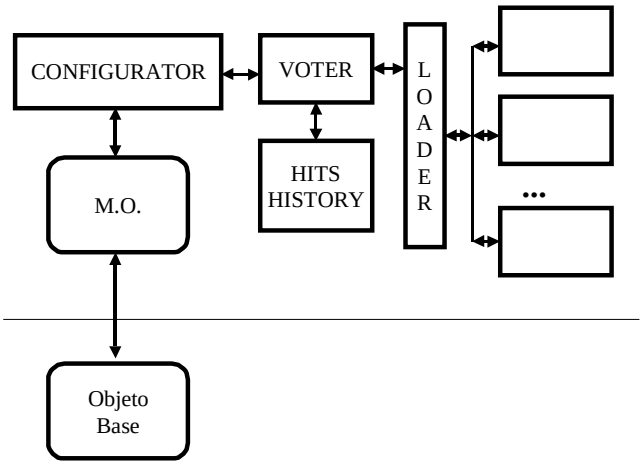

FIGURA 6.25 – Redundância ou diversidade de projeto usando um mecanismo de votação

6.6.8 Conclusões

Requisitos não funcionais, no intuito de obter-se maior grau de reuso e transparência, devem ser colocados à parte da funcionalidade da aplicação. Isto pode ser obtido através da combinação de várias técnicas. Reflexão computacional permite reconfiguração dinâmica e alteração comportamental. *Middlewares* e o modelo orientado a objetos permitem encapsulamento e aumento de reuso. A criação de um ambiente no meta-nível como um *middleware* torna a aplicação muito mais flexível e capaz de lidar com uma grande variedade de contextos e demandas.

Mais importante que tudo, esta combinação permite a alteração do comportamento de uma aplicação em tempo de execução. Este cenário focalizou a atomicidade como requisito não funcional desejado e mostrou a efetividade de algumas características reflexivas quando colocadas em um *middleware*, especialmente ao serem usadas na alteração do comportamento do *middleware*, em resposta às novas demandas provenientes de mudanças no ambiente computacional. Uma arquitetura independente, capaz de detectar alterações e responder às mesmas é útil no desenvolvimento de soluções que substituam o uso da estratégia de configuração estática. Reconfiguração dinâmica é um conceito chave para adaptabilidade e transparência. Ela não apenas ajuda a cobrir uma gama maior de situações, mas o faz de modo a incrementar seu reuso. Desde que não é possível prever o que uma aplicação terá de enfrentar durante sua execução, a melhor maneira de tornar soluções mais reutilizáveis é prover mecanismos que não dependam das classes presentes na aplicação e nem apresentem comportamento fixo. Pelo contrário, devem possuir mecanismos de decisão que permitam selecionar o próximo passo a ser dado.

6.6.9 Reutilização das técnicas empregadas

O MOP Guaraná fornece algumas meta classes que podem ser extremamente úteis na composição de configurações múltiplas [OLI 99a][OLI 99b], o que significa ampliação de sua abrangência e reutilização, pois novas meta classes podem ser acrescentadas e outras alteradas ou excluídas da configuração sem que isto implique em alterações na aplicação. Mecanismos decisórios movidos para o meta-nível podem monitorar e selecionar as meta classes disponíveis

no lugar da aplicação, liberando o desenvolvedor deste trabalho, o que implica em soluções mais genéricas e portanto reutilizáveis.

6.6.10 Conclusões

Diversos mecanismos de tolerância a falhas podem ser implementados com o uso de grupos de objetos, porém se estes forem meta objetos, sua reutilização é potencializada. Grupos de objetos podem ser usados para implementar técnicas diferentes ou ainda pode-se compor várias instâncias de uma mesma classe e um mecanismo de votação. A mesma operação pode ser passada a todas as instâncias desta classe e o mecanismo receber todos os seus resultados e chegar a uma conclusão sobre a validade dos mesmos. Desta forma, caso ocorra falha em uma das instâncias do grupo, seu resultado diferente dos demais é a indicação de algum problema na execução. Esta configuração com instâncias da mesma classe não cobre problemas de desenho de classe. Neste caso, uma solução melhor são instâncias de diversas classes diferentes, com mecanismos diferentes para a resolução do mesmo problema. Novamente, uma classe com mecanismo de votação será necessária.

6.7 Resumo do capítulo

Este capítulo mostrou como as diversas técnicas e paradigmas estudados puderam ser utilizados com o objetivo de produzir soluções com alto grau de reutilização, adaptabilidade, transparência, simplicidade e velocidade de desenvolvimento. Diversos cenários foram montados, com a preocupação de cobrir situações diferentes. A implementação de soluções para os mesmos serviu de substrato para a chegada a conclusões a respeito dos efeitos dos paradigmas estudados em cada contexto. A partir da análise dos mesmos, pode-se chegar então a recomendações sobre seu uso.

7 Trabalhos correlatos

7.1 Introdução

Este capítulo mostra outros estudos e soluções para o problema de suporte a atomicidade em aplicações tolerantes a falhas. Após uma breve descrição de cada uma, uma análise é feita e as mesmas são comparadas com os cenários desenvolvidos neste trabalho.

7.2 Kan

7.2.1 Apresentação

Kan, que em sânscrito significa pequena partícula, átomo ou molécula, é um sistema distribuído orientado a objetos para suporte a aplicações confiáveis distribuídas e paralelas de grande tamanho e complexidade [JAM 2000]. Uma das preocupações deste sistema é com a tolerância a falhas. Foi desenvolvido na Universidade da Califórnia em Santa Barbara, como tese de doutorado. O sistema Kan é implementado em Java e usa o conceito de transações atômicas simples e aninhadas com o objetivo de preservar a integridade de dados.

Procura fornecer ao programador um ambiente para acesso eficiente e seguro a objetos. Este ambiente engloba conceitos como transações atômicas, chamadas a métodos assíncronos e multitarefa. Seu esquema incorpora recuperação de dados *roll-forward* e uso de *log*. Também utiliza estruturas lógicas e combinação de mensagens para reduzir o *overhead* do processo de *logging*.

Kan sustenta que a utilização de linguagens orientadas a objetos juntamente com modelos OO reduz as diferenças entre o modelo e sua implementação, facilitando nas assunções a respeito de seu comportamento. Utiliza transações atômicas para garantir resultados corretos em execuções concorrentes de *threads*. Justifica que tal tipo de execução pode levar a interferências indesejadas. Também discute o uso de métodos sincronizados na serialização de acessos a dados. Além de bloqueios, sugere tornar partes do código atômico, utilizando-se de "guardas". Cada bloco atômico (ou transação) possui um guarda, que pode também conter um predicado que precisa ser satisfeito para que as operações sejam executadas. Tais predicados também podem ser métodos que acessem os objetos a serem acessados a seguir pelas operações internas ao guarda. A única restrição que regula a utilização destes métodos internos aos predicados rege que eles sejam por suposto apenas de leitura. Kan emprega um compilador que produz código Java que utiliza a API Kan.

Este sistema também pretende fornecer ao programador uma interface simples e poderosa para uma camada de gerenciamento de objetos que suporte acesso eficiente e confiável aos mesmos. Na figura 7.1, pode-se ver a implementação de um método utilizando guarda e predicado interno (pré-condição). O bloco atômico é iniciado com a palavra "guard" e uma pré-condição.

```
public Cell dequeue ( ) {
Cell c;

guard (head != null) {
        c = head;
        head = c.next;
        if (head == null)
                tail = null;
```

```
}
c.next = null;
return c;
}
}
```

FIGURA 7.1 – Uso de guardas e pré-condições em Kan

7.2.2 Comparação e conclusões

Um dos cenários desenvolvidos neste trabalho estudou a utilização de uma variante do conceito de guardas utilizado em Kan. O emprego da reflexão computacional neste cenário buscou contornar um dos maiores problemas encontrados no uso simples de guardas: embora a granularidade de bloqueio seja menor, os dados continuam indisponíveis mesmo se não estiverem sendo acessados pelo agente bloqueante. Através de reflexão computacional, chamadas a um guarda podem ser interceptadas e, no meta-nível, descobrem-se os objetos clientes e os dados que os mesmos estão tentando acessar. Este mecanismo permite diminuir o tempo de bloqueio dos dados.

7.3 Arjuna

7.3.1 Apresentação

Arjuna é um sistema de programação orientado a objetos que fornece ferramentas para construção de aplicações distribuídas tolerantes a falhas. Suporta (trans)ações atômicas aninhadas. Os objetos das aplicações podem ser manipulados apenas dentro destas ações atômicas, o que garante sua integridade. Os mecanismos que dão este suporte foram implementados através de uma hierarquia de classes em C++, conforme visto na figura 7.3. A classe AtomicAction provê os métodos Begin, End e Abort. Operações do tipo *protected* implementam Prepare e Commit para as duas fases do protocolo *two-phase commit*. A figura 7.2 mostra o emprego de duas ações atômicas.

```
AtomicAction A, B;
A.Begin ( );
B.Begin ( );
B.Abort( );
A.End( );
```

FIGURA 7.2 – Ações atômicas em Arjuna

A classe ConcurrentAtomicAction derivada de AtomicAction suporta concorrência. O mecanismo padrão de suporte a recuperação grava o estado corrente do objeto antes de sua primeira alteração dentro da ação atômica. Em caso de falha, este estado é gravado de volta sobre o objeto. Este mecanismo encontra-se na classe StateManager, através de sua interface, que contém os métodos *save_state* e *restore_state*, que atuam junto com a classe ObjectState, onde estão os dados salvos. Os métodos *save_state* e *restore_state* devem ser implementados pelo desenvolvedor, pois dependem dos tipos de dados incluídos no objeto a ser salvo. Para maiores detalhes, ver [SHR 89] e [SHR 94].

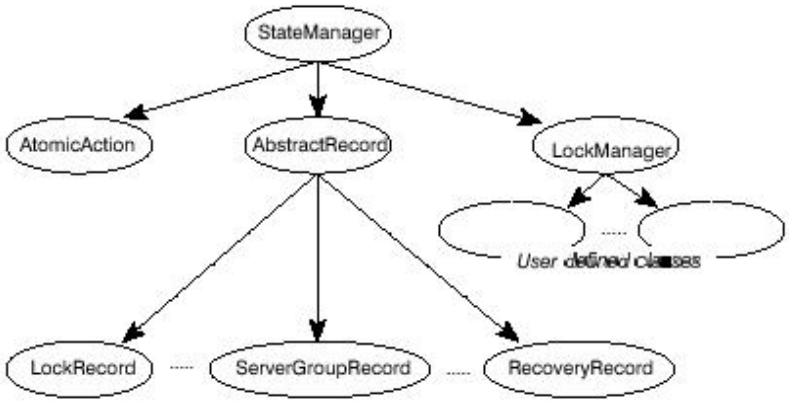

FIGURA 7.3 – Hierarquia de classes Arjuna [SHR 89]

7.3.2 Comparação e conclusões

As classes definidas pelo desenvolvedor devem ser inseridas dentro da hierarquia de classe Arjuna (fig. 7.3) , o que demanda conhecimento das características da mesma. Além disso, exige-se a implementação de métodos customizados para as classes do desenvolvedor, o que limita a utilização das mesmas ao seu contexto específico.

7.4 Arcabouço atômico orientados a objetos

7.4.1 Apresentação

Tekinerdogan [TEK 94] desenvolveu um arcabouço orientado a objetos para suportar transações atômicas. Tal arcabouço apresenta uma série de classes que atuam em conjunto em tal tarefa (fig. 7.4).

Cada módulo envia solicitações e recebe respostas do próximo nível. Cada instância de Transaction tem sua correspondente instância de TransactionManager, que implementa operações como startTransaction, commit e abort. PolicyManager passa as operações para os objetos DMObject. Uma mesma instância de PolicyManager pode servir várias de TransactionManager. As operações recebidas por DMObject são passadas para DataManager, responsável pela consistência do objeto. DataManager consulta um objeto Scheduler que controla a ordem de execução das operações. O Scheduler aceita ou não as operações. Se a operação for aceita, é enviada para o RecoveryManager, caso contrário volta até o Transaction, que chama a operação abort.

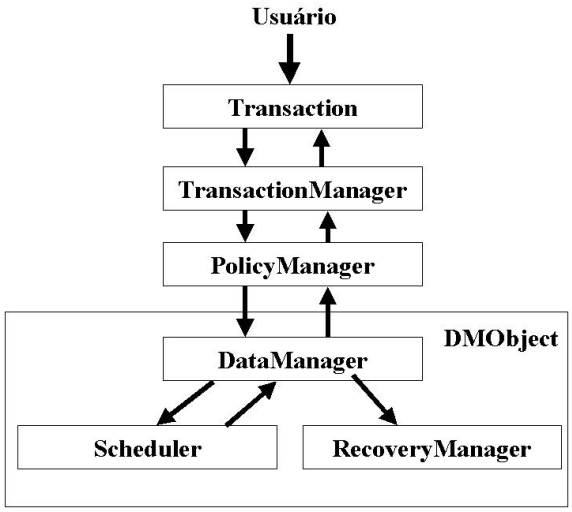

FIGURA 7.4 – Arcabouço OO para suporte a atomicidade (adaptado de [TEK 94])

7.4.2 Comparação e conclusões

Por ser inteiramente baseado nos conceitos de orientação a objetos, sem utilizar qualquer outro tipo de solução, este arcabouço apresenta dificuldades no que tange à adaptabilidade, facilidade de uso e transparência. O estudo desta solução concluiu que o usuário deste arcabouço precisa conhecer certas estruturas do mesmo e deve adaptá-las aos objetos que pretende garantir atomicidade. Não há previsão de adaptação das políticas ao contexto da aplicação através de mecanismos decisórios de adoção para novas políticas.

7.5 Resumo do capítulo

Algumas soluções que buscam facilitar a introdução de atomicidade em aplicações foram estudadas sob a ótica específica da reutilização e suas demandas. Algumas destas soluções implicam em mais conhecimento das mesmas, outras menos, porém todas lidam com classes previamente estabelecidas e desta situação provém sua maior dificuldade de adaptação e, como conseqüência, de reuso. Este poderia ser obtido em maior grau caso paradigmas com características próprias fossem incorporados a estas soluções.

8 Conclusões

8.1 Utilização dos paradigmas neste trabalho

8.1.1 Padrões de software

Padrões de *software* trazem embutido dentro de seu conceito a idéia de reutilização, portanto são pontos a considerar quando da elaboração de aplicações sob o contexto de reusabilidade. Percebeu-se pelo seu estudo que a grande busca dos padrões de *software* é difundir uma solução previamente testada, para que a mesma possa ser reutilizada tantas vezes quanto forem as ocorrências do problema que lhe deu origem. Não é possível, entretanto, prever todas as situações e problemas passíveis de ocorrência, portanto existe uma lacuna a ser preenchida por outro paradigma que permita expandir um pouco mais o conceito de reutilização previsto pelos padrões de *software*. Neste trabalho, implementou-se a associação entre padrões e reflexão computacional como forma de, simbioticamente, potencializar a aplicação destes paradigmas, que apresentam muitos conceitos complementares.

8.1.2 Reflexão computacional

Muitas das características que compõem o paradigma reflexivo podem ser empregadas no aumento do grau de reutilização de uma aplicação. A prospecção de dados e valores é uma importante capacidade fornecida por este paradigma, possibilitando a escrita de código que não lide explicitamente com tipos de classes e atributos. Ao invés, tais informações podem ser obtidas e usadas em tempo de execução, o que torna o código genérico e, por conseguinte, reutilizável com outras classes.

Outra característica reflexiva que se mostrou extremamente valiosa foi a interceptação de mensagens. Através dela, pode-se obter um maior controle sobre a aplicação, permitindo-a mais adaptável. Adaptabilidade e auto configuração dinâmica são dois outros conceitos chave em se tratando de reutilização. O fato de uma aplicação poder alterar a si própria aumenta o espectro de cenários nos quais pode atuar.

A associação da prospecção com interceptação de mensagens permite a criação de mecanismos genéricos de recuperação e restauração de dados. Já a utilização destes conceitos reflexivos com a modularização fornecida pela utilização do modelo orientado a objetos permite isolar estes aspectos da aplicação, permitindo que os mesmos sejam reutilizados em outras da mesma forma.

A possibilidade de associação de meta-classes dinamicamente faz com que não apenas uma mas várias soluções sejam desenvolvidas e utilizadas de acordo com o contexto, permitindo maior abrangência à aplicação e, consequentemente, maior reutilização. Aqui também novamente conceitos do modelo orientado a objetos podem ser usados em associação com reflexão. Implementando meta-classes, modularizam-se as características que cada uma encerra, permitindo com isso diversas combinações de acordo com as necessidades do cenário. Grupos de classes podem estar disponíveis e ser alterados dinamicamente por um mecanismo decisório. Além disso, estas classes podem ser criadas ou alteradas posteriormente e acrescentadas ao conjunto sem que a aplicação precise ser alterada.

8.1.3 Orientação a objetos

O modelo orientado a objetos permite a produção de *software* com certas características que facilitam sua reutilização, incremento na velocidade de desenvolvimento de aplicações e representação hierárquica de dados [FER 97][FER 98b]. A organização em classes e sua correspondente hierarquia, bem como o encapsulamento daí proveniente, possibilitam melhor separar a porção reutilizável construída e facilitar sua adição às novas aplicações. Estas capacidades foram utilizadas em todo este trabalho. Todas estas características contribuem para o aumento na velocidade de desenvolvimento de aplicações.

8.2 Associações

Uma das contribuições deste trabalho foi executar e testar implementações de gerenciamento de atomicidade multi-facetadas, ou seja, empregando associações de paradigmas com o objetivo de fazê-los suprirem-se ou complementarem-se mutuamente. Exploraram-se as características do modelo orientado a objeto e, partindo de implementações baseadas nesta técnica [TEK 94][WU 95], observou-se que havia possibilidade de incremento de transparência dessas implementações com a introdução de conceitos de reflexão computacional. Este incremento de transparência significou aumento no grau de reutilização da solução proposta, pois escondeu da aplicação detalhes como tipos e quantidades de classes, tornando a solução mais genérica. Um dos exemplos foi a substituição do uso do mecanismo de herança oriundo da orientação a objetos pelo conceito de interceptação de mensagens proveniente da reflexão computacional para conectar classes que implementam requisito funcionais de classes que implementam requisitos não funcionais. Este processo não ocorreu simplesmente sob a forma de substituição de conceitos. Ao contrário, papel mais relevante foi exercido pela complementação de conceitos. Desta forma, a modularização proveniente da orientação a objetos mostrou-se uma ótima maneira de implementar a arquitetura reflexiva, que tem como um de seus pilares a separação de níveis.

A associação de conceitos de padrões de *software* com reflexão também mostrou-se produtiva nas implementações realizadas. Padrões de *software* tornam-se mais abrangente se implementados com conceitos de orientação a objetos e, especialmente, reflexão computacional. As soluções preconizadas por aquele paradigma tornam-se mais flexíveis e adaptáveis quando implementadas de forma reflexiva. Padrões de *software* mostram uma solução. Reflexão computacional torna a implementação desta solução mais abrangente e reutilizável, o que, em última análise, é o objetivo daqueles. Isto pôde ser observado não apenas quanto ao número de diferentes cenários que puderam ser abrangidos com a mesma solução. A adaptabilidade deu-se também em nível de execução, pois conceitos reflexivos implantados nas soluções deram a estas capacidade de reconfiguração dinâmica. Todas estas características (adaptabilidade, auto-reconfiguração, flexibilidade, transparência, entre outras) permitiram aumentar-se o grau de reutilização das soluções. O cuidado tomado durante sua implementação foi evitar que as soluções se tornassem complexas, lentas, ou ainda demandassem conhecimento adicional do desenvolvedor o que, em última análise, contribuiria para a diminuição da transparência e da velocidade de desenvolvimento.

Finalmente, observou-se que o grau de participação de cada conjunto de conceitos deve ser medido com cuidado, pois cada um deles por vezes traz embutidas desvantagens que podem ser potencializadas com a associação em estudo. O exemplo mais notório é a sobrecarga intrínseca à aplicação de certos conceitos reflexivos, como a interceptação de mensagens. Devem-se pesar os ganhos e complicações que cada solução traz embutida.

8.3 Considerações finais

Aplicações precisam produzir resultados confiáveis, particularmente as que manipulam informações críticas. Execuções parciais de operações, dados inconsistentes, variações de resultados e situações afins não podem ser consideradas possibilidades aceitáveis. Sob este prisma, percebe-se a importância de aplicações e sistemas possuírem mecanismos de tolerância a falha que contemplem e evitem tais cenários.

A manipulação de dados, desde atributos de objetos alocados em endereços de memória volátil até registros em qualquer formato de mídia permanente, é uma das situações mais propícias à ocorrência de falhas e conseqüentes incorreções delas provenientes. Associando-se este fato à importância e volume dos mesmos nos sistemas computacionais, conclui-se que, dentro do contexto de tolerância a falhas, ocupa lugar de destaque o gerenciamento de atomicidade dos dados e suas duas vertentes, o controle de concorrência e a recuperação após falhas.

Uma grande dificuldade encontrada por aplicações que necessitem de gerenciamento de atomicidade é a diversidade de situações onde este deve ser aplicado, além das diferentes estratégias que podem ser adotadas em cada uma delas. O problema não é apenas decidir e executar a estratégia mais correta, mas poder vislumbrar todas as possibilidades e contemplá-las de forma viável, sem que a aplicação vire um apêndice de seus módulos de suporte a atomicidade. Isto leva a um conceito chave, que foi objetivo deste estudo: a reutilização de soluções de gerenciamento de atomicidade.

Produzir soluções reutilizáveis simplesmente não é o suficiente, se as mesmas não apresentarem uma série de outros requisitos mínimos que advoguem sua viabilidade. Para que possa ser explorada em todo seu potencial, deve fazer-se acompanhar de outras características que permitam-na desempenhar seu papel de forma eficiente. Deve haver transparência e simplicidade, ou seja, o desenvolvedor não deve se preocupar em conhecer classes extras ou pulverizar o código da aplicação com comandos especiais. A solução também deve ser adaptável, ou seja, não deve conter código que manipule diretamente tipos específicos de classes, o que tolheria sua capacidade de lidar com outras classes senão as já previamente conhecidas e também exigiria recodificações específicas a cada nova reutilização. Outra característica importante é a modularização, pois a atomicidade, como requisito não funcional, deve estar isolada do resto da aplicação. Por fim, a solução deve incrementar a velocidade da aplicação, pois a parte relativa à atomicidade deve ser inserida ao contexto global de forma imediata.

A modularização pode ser utilizada na separação de requisitos funcionais e não funcionais. Posteriormente, este isolamento pode ser complementado pela colocação dos módulos responsáveis pelos diferentes tipos de requisitos em camadas à parte, valendo-se para isto da utilização de uma arquitetura reflexiva. Esta combinação permite maior grau de reutilização das soluções de gerenciamento de atomicidade pois esta, sendo um típico requisito não funcional, pode ser isolada no meta-nível da arquitetura reflexiva e suas classes reutilizadas em outras aplicações sem que, para isto, seja necessária qualquer alteração nas mesmas ou nas aplicações. Esta implementação da arquitetura reflexiva utilizando-se de conceitos de orientação a objetos foi complementada pelos mecanismos de interceptação de mensagens e prospecção de atributos, os quais tornaram as aplicações mais adaptáveis dinamicamente, o que também serviu para aumentar seu grau de reutilização.

Percebeu-se neste estudo que o conceito de reutilização deve ser tratado não apenas sob o ponto de vista estático, ou seja, pensando-se em soluções predeterminadas que abranjam o maior número de casos possíveis. Deve-se pensar também em soluções que ataquem casos não previstos *a priori*. Para que isso seja possível, deve-se dotar as soluções de capacidade decisória seja através de mecanismos de votação, algoritmos de decisão ou outra solução similar. Foi

possível dotar as soluções de alto grau de decisão através da utilização de classes específicas colocadas no meta-nível com tal propósito. O uso de reconfiguração dinâmica permitida pela reflexão computacional permitiu também que classes implementando diferentes soluções fossem escolhidas ou substituídas de acordo com o contexto corrente da aplicação. Esta possibilidade deu novo significado ao termo reutilização, pois abriu a perspectiva não apenas de utilização alternada das meta-classes já disponíveis, mas também da inserção de novas meta-classes com outras soluções mesmo após a aplicação ter sido codificada, sem que isto significasse recodificação desta.

O não atrelamento das meta-classes implementadoras de requisitos não funcionais a tipos específicos tornou possível a criação de soluções que foram capazes de lidar com qualquer tipo de objetos, ao mesmo tempo em que apresentaram maior grau de simplicidade e transparência de utilização, pois não demandaram alteração a cada nova utilização em outra aplicação. Para isso, as soluções aqui implementadas valeram-se das já citadas características reflexivas de introspecção de dados e interceptação de mensagens.

O estudo dos conceitos de reflexão computacional permite concluir que muitos de seus conceitos são talhados para uso em *software* reutilizável. Há, contudo, vários fatores a contribuir para que seja empregado em menor grau do que poderia. Inicialmente, pode-se citar a existência de *overhead* causado pelos constantes desvios e tratamentos do fluxo de execução do código da aplicação. Se a aplicação for bem planejada, todavia, o ganho que pode ser obtido evitando-se execuções desnecessárias ou prevenção de erros pode superar tal prejuízo. O maior problema enfrentado quando da utilização de reflexão computacional e suas implementações foi o estágio de maturidade encontrado nestes. Entre outros fatores, pode-se citar como dificuldades encontradas a dificuldade de uso (e, em certos casos, até de compreensão) dos protocolos, a falta de abrangência de alguns conceitos (desde impossibilidade de interceptação de mensagens até falta de suporte a interceptação ativa) e problemas de implementação (dificuldades de utilização de parâmetros em métodos de classes reflexivas, recursões infinitas, etc). Na verdade, os recursos reflexivos serão mais utilizados quando fizerem parte do pacote da linguagem, como no caso de Java. O problema desta é, entretanto, a pequena quantidade de características implementadas. O uso de reflexão computacional será mais difundido quando outras linguagens de programação adotarem seus conceitos. Para estas, o chamariz provavelmente não será a implementação de tolerância a falhas, mas a possibilidade de gerar código reutilizável através de programação sem referências a classes específicas. As sugestões feitas durante este trabalho, porém, podem servir para tornar os protocolos reflexivos mais propícios à sua utilização para gerenciamento de atomicidade.

Observou-se que os conceitos de padrões de *software* e reflexão computacional apresentam muitos pontos complementares quanto à reutilização e podem ser utilizados em uma simbiose muito produtiva. O objetivo deste estudo não foi validar padrões de *software*, mas observar seu uso junto a outros paradigmas na implementação de atomicidade. Sob este aspecto, as implementações desenvolvidas mostraram que esta é uma ótima opção no que tange ao objetivo deste trabalho, ou seja, estudar soluções reutilizáveis para gerenciamento de atomicidade.

O modelo orientado a objetos tem como uma de suas preocupações produzir *software* reutilizável através do emprego de vários de seus conceitos. Estes foram utilizados em combinação com conceitos de outros paradigmas e mostraram ser importantes ferramentas auxiliares e, às vezes, até principais em certas implementações. Possibilidades coincidentes fornecidas por mais de um paradigma foram testadas e comparadas. Entre outras, percebeu-se que a aplicação do mecanismo de herança é menos eficiente que o emprego de interceptação de mensagens. Diversos cenários implementados confirmaram, entretanto, que soluções reutilizáveis passam pelo emprego de conceitos deste modelo.

Estudou-se um grande número de situações onde o gerenciamento de atomicidade é condição *si ne qua non* para a execução confiável da aplicação. Percebeu-se que esta variedade de situações é o principal empecilho ao desenvolvimento de soluções globais para este problema. O que pode ser feito, contudo, é produzir soluções que abranjam o maior número possível de situações ou possam ser adaptadas dinamicamente a elas. Soluções ou partes destas que possam ser reaproveitadas em outras aplicações adquirem maior importância diante deste contexto. Para que isto possa ser obtido, não há uma receita única. A combinação de características de diferentes paradigmas, conforme foi mostrado neste trabalho, é o caminho mais curto para a obtenção deste objetivo.

Para uma grande variação de problemas a serem enfrentados, nada melhor que uma grande variedade de técnicas que possuam características complementares, de modo que cada uma possa atuar a partir do ponto limitante da outra. Para que as soluções possam ser reutilizadas, não podem estar presas a especificidades, tampouco podem apresentar comportamento fixo. A combinação das técnicas empregadas deve resultar em uma solução que seja genérica e auto-adaptável. Genérica para atuar em um maior número de situações. Auto-adaptável para poder suportar variações em tempo de execução que levariam uma solução de comportamento fixo ao colapso. Tais soluções, entretanto, devem ser de fácil utilização e troca, além de exigirem do desenvolvedor um mínimo de conhecimento a seu respeito e um mínimo de modificações em seu código original. Um baixo acoplamento entre os módulos da aplicação e os responsáveis pela implementação das soluções, neste caso particular o gerenciamento de atomicidade, facilita o trabalho do responsável pela atualização da aplicação à medida em que novas e melhores soluções são propostas e implementadas.

Bibliografia

[ANC 95] ANCONA, M.; CLEMATIS, A.; LISBÔA, M.L.B. et al. Reflective Architectures for Reusable Fault-Tolerant Software**.** In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 22., 1995, Canela, RS. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1995. v.1, p. 87-98.

[APP 2000] APPLETON, B. **Patterns and Software**: Essential Concepts and Terminology. Disponível em: <http://www.enteract.com/~bradapp/>. Acesso em: 14 maio 2001.

[AZE 96] AZEVEDO, R.L.G.; JANSCH-PÔRTO, I. Algoritmos de Controle de Concorrência no Acesso a Dados. In: SIMPÓSIO REGIONAL DE TOLERÂNCIA A FALHAS, Porto Alegre. **Anais...** Porto Alegre: UFRGS, 1996.

[BED 98] BEDER, D.M. ; RUBIRA, C.F. **Uma Abordagem Reflexiva baseada em Padrões de Projeto para o desenvolvimento de Aplicações Distribuídas Confiáveis.** Campinas: Instituto de Computação, Unicamp, 1998.

[BED 99] BEDER, D.M.; RUBIRA, C.F. **A comparative study of fault-tolerant concurrent mechanisms:** atomic transactions, conversations and coordinated atomic actions. Campinas: Instituto de Computação, Unicamp, 1999.

[BER 81] BERNSTEIN, P.; GOODMAN, N. Concurrency Control in Distributed Database Systems. **Computing Surveys**, New York, v.13, n. 2, June 1981.

[BER 87] BERNSTEIN, P.A.; HADZILACOS, V.; GOODMAN, N**. Concurrency control and recovery in database systems**. New York: Addison Wesley, 1987.

[BES 99] BERTAGNOLLI, S. **Taxonomia de protocolos de reflexão computacional e sua aplicação em Java.** 1999. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BLA 2000] BLAIR, G.S.; COULSON, G.; COSTA, F.; DURAN, H. **On the Design of Reflective Middleware Platforms**. Chicago: Department of Computer Science, University of Illinois at Urbana-Champaign, 2000.

[BLA 99] BLAIR, G.; PAPATHOMAS, M. **The Case for Reflective Middleware**. Lancaster: Department of Computing, Lancaster University, 1999.

[BOU 90] BOUDOL, G. **Atomic actions**. Antipolis: INRIA Sophia, 1990. Disponível em: <http://www-sop.inria.fr/mimosa/personnel/Gerard Boudol.html>. Acesso em: 10 set. 2000.

[BUS 96] BUSCHMANN, F. et al. **A System of Patterns**: Patterns-Oriented Software**.** New York: John Wiley & Sons, 1996.

[BUZ 95] BUZATO, L.E. Tolerância a falhas, orientação a objetos e ações atômicas. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6. , 1995. Campina Grande, PB. **Anais...** Campina Grande: SBCTF, 1995. p 33-47.

[BUZ 97] BUZATO, L.E.; RUBIRA, C.M.; LISBÔA, M.L.B**.** A reflective object-oriented architecture for developing fault-tolerant software**. Journal of the Brazilian Computer Society**; [S.l.] v.4, n.2, 1997.

[CAM 95] CAMPOS, J.B. **Atomicidade e durabilidade em sistemas de gerenciamento distribuído de transações**. 1995. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[CAZ 99] CAZZOLA, W. et al. **Architectural reflection:** concepts, design and evaluation. Milão: Milano University, 1999

[CEL 89] CELLARY, W.; GELENBE, E.; MORZY, T. **Concurrency control in distributed database systems.** Amsterdam: North-Holland Press, 1989.

[COP 95] COPLIEN, J.; SCHMIDT, D. **Pattern Languages of Program Design.** Reading, Massachussets: Addison-Wesley, 1995.

[COR 96] CORRÊA, S.L. et al. Tolerância a Falhas de Software em Linguagens Orientadas a Objetos Reflexivas. In: SIMPÓSIO REGIONAL DE TOLERÂNCIA A FALHAS, Porto Alegre. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1996.

[CON 98] CORNELL, G.; HORSTMANN, C. **Core Java.** São Paulo: Makron Books, 1998.

[COS 2000a] COSTA, F. **Reflective Middleware Platforms:** Design and Implementation Chicago: Department of Computer Science, University of Illinois at Urbana-Champaign, 2000.

[COS 2000b] COSTA, F.; BLAIR, G. **Integrating Meta-Information and Reflection in Middleware.** Lancater: Department of Computing, Lancaster University, 2000.

[COS 99] COSTA, F.; BLAIR, G.; COULSON, G. **Experiments with Reflective Middleware.** Lancaster: Distributed Multimedia Research Group, Department of Computing, Lancaster University, 1999.

[COU 2000] COULSON, G. **What is Reflective Middleware?** Lancaster: Distributed Multimedia Research Group, Department of Computing, Lancaster University, 2000.

[DAT 91] DATE, C.J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro: Campus, 1991.

[ELM 91] ELMAGARMID, A.K. **Transaction management in database systems.** New York: Morgan Kaufman Publishers, 1991.

[ESW 76] ESWARAN, K. et al. The notions of consistency and predicate locks in a database system. **Communications of the ACM**, New York, v. 19, n.11, 1976.

[FER 2000] FERNANDES, A. P. **Mecanismos de suporte a atomicidade em Java.** 2000. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[FER 2001a] FERNANDES, A. P., LISBÔA, M.L.B. Reflective Implementation of an Object Recovery Design Pattern. In: CONGRESSO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 7. , 2001. El Calafate. **Anais...** El Calafate: Redunci, 2001.

[FER 2001b] FERNANDES, A. P. Reflexão computacional. **Revista CCEI-URCAMP**, Bagé, RS, v.5, n.7, p. 61-66, mar. 2001.

[FER 2001c] FERNANDES, A. P. ; LISBÔA, M.L.B. Implementação Reflexiva de um Padrão de Projeto para Recuperação de Estados de Objetos. **Revista CCEI-URCAMP**, Bagé, RS, v.5, n.8, p. 41-48, ago. 2001.

[FER 97] FERNANDES, A. P. O próximo passo na difusão da orientação a objetos. **Revista Developers Magazine**, Rio de Janeiro, RJ, n.7, p. 42-43, mar. 1997.

[FER 98a] FERNANDES, A. P. Orientação a objetos em ferramentas RAD. **Revista Developers Magazine**, Rio de Janeiro, RJ, n.19, p. 36-37, mar. 1998.

[FER 98b] FERNANDES, A. P. Implemente a orientação a objetos em C++ e Delphi. **Revista Developers Magazine**, Rio de Janeiro, RJ, n.26, p. 30-32, out. 1998.

[GAB 96] GABRIEL, R.P. **Patterns of Software**: Tales from the Software Community. Oxford: Oxford Press, 1996.

[GAM 95] GAMMA, E. et al. **Design Patterns**: Elements of Reusable Design. Reading,

Massachussets: Addison-Wesley, 1995.

[GOL 97] GOLM, M.; KLEINODER, J. **A plataform for adaptable operating-system mechanisms.** Alemanha: Universidade de Erlangen, 1997. Disponível em: <http://www4.informatik.uni-erlangen.de/TR/TR-14-97-10.abs.html>. Acesso em: 14 dez. 1999.

[HAE 83] HAERDER, T.; REUTER, A**.** Principles of transaction-oriented database recovery. **ACM Computing Surveys**, New York, v. 15, n. 4, 1983.

[HAE 96] HAETINGER, W.; LISBOA, M.L.B. **Um Exemplo de Técnica de Tolerância a Falhas Implementada com Grupos de Objetos**. In: SIMPÓSIO REGIONAL DE TOLERÂNCIA A FALHAS, 1996, Porto Alegre. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1996.

[HAE 98] HAETINGER, W. **Troca dinâmica de versões de componentes programas no modelo de objetos**. 1998. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[HER 88] HERLIHY, M.; WEIHL, W.E. Hybrid concurrency control for abstract data types. In: ACM-SIGACT-SIGMOD-SIGART SYMPOSIUM ON PRINCIPLES OF FATABASE SYSTEMS, 7., 1988. **Proceedings...** New York: ACM Press, 1988.

[ITO 95] ITOH, J.; LEA, R.; YOKOTE, Y. Using meta-objects to support optimisation in the Apertos operating system. In: USENIX CONFERENCE ON OO TECNOLOGIES - COOTS, Monterrey, 1995. **Proceedings...** Monterrey, California, 1995.

[JAL 95] JALOTE, P. **Fault tolerance in distributed systems**. New Jersey: PTR Prentice Hall, 1995. cap. 6, p.217-253.

[JAM 2000] JAMES, J.W. **Reliable Distributed Objects: Reasoning, Analysis and Implementation.** 2000. Tese de Doutorado - UCSB - University of California em Santa Barbara. Santa Barbara, California.

[KAS 99] KASBEKAR, M.; NARAYANAN, C.; DAS, C.R. **Using reflection for checkpointing concurrent object oriented programs**. Pennsylvania: Department of Computer Science and Engineering, Pennsylvania State University, 1999.

[KIC 91] KICZALES, G.; des RIVIÉRES, J.; BOBROW, D.G. **The Art of MetaObject Protocol.** Cambridge: MIT Press, 199.

[KIC 97] KICZALES, G. et al. Aspect-Oriented Programming. In: ECOOP, 1997. **Proceedings...** Berlin: Springer-Verlag, 1997.

[KUN 81] KUNG, H.T.; ROBINSON, J.T. On optimistic methods for concurrency control, **ACM Transactions on Database Systems**, New York, v. 6, n.1, 1881.

[LAM 79] LAMPSON, B.; STURGIS, H. **Crash recovery in a distributed data storage system**. 1979. Não publicado.

[LIS 94] LISKOV, B. **Overview of the Argus language and system**. Cambridge: MIT, 1984. (Programming Methodology Group Memo 40).

[LIS 95a] LISBÔA, M.L.B.; CAVALHEIRO, G.C.H. Reflexão computacional sobre técnicas de tolerância a falhas em software. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6. , 1995. Canela. **Anais...** Canela: SBC, 1995.

[LIS 95b] LISBÔA, M. L. B. **MOTF Meta-objetos para tolerância a falhas.** 1995. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[LIS 96] LISBÔA, M.L.B.; RUBIRA, C.M.; BUZATO, L.E. Arquitetura reflexiva para o desenvolvimento de software tolerante a falhas. In: CONGRESSO NACIONAL DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 15., Rio de Janeiro. **Anais...**

Rio de Janeiro: SBC, 1995.

[LIS 97a] LISBÔA, M.L.B. **Tutorial:** arquiteturas de meta-nível.Fortaleza: UFC, 1997. 35 p. Tutorial apresentado no Simpósio Brasileiro de Engenharia de Software, SBES, 11. 1997.

[LIS 97b] LISBÔA, M.L.B. A new trend on the development of fault-tolerant applications: software meta-level architectures. **Journal of the Brazilian Computer Society**, [S.l.], v.4, n.2, p.31-38, 1991.

[LIS 98] LISBÔA, M.L.B. **Reflexão computacional no modelo de objetos**. Porto Alegre: CPGCC da UFRGS, 1998.

[LIT 90] LITTLE, M.; SHRIVASTAVA, S. Understanding the role of atomic transactions and group communications in implementing persistent replicated objects. In: INTERNATIONAL WORKSHOP ON PERSISTENT OBJECT SYSTEMS: DESIGN, IMPLEMENTATION AND USE, 8., 1990. **Proceedings...** [S.l.:s.n.], 1990.

[LIT 97] LITTLE, M. **Building reliable Web applications using atomic actions**. Department of Computing Science, University of Newcastle upon Tyne, 1998. Disponível em: <http://arjuna.newcastle.ac.uk/index.html>. Acesso em: 28 dez. 1999.

[MAE 87] MAES, P. Concepts and experiments in computational reflection. **ACM SIGPLAN Notices**, New York, v. 22, p. 147-155, 1987. Trabalho apresentado na OOPSLA, 1987.

[MCC 97a] MCCMANIS, C. **Take an in-depth look at Java Reflection**. Disponível em: <http://www.javaworld.com/jw-09-1997/jw-09-indepth.html>. Acesso em: 10 nov. 1999.

[MCC 97b] MCCMANIS, C. **Take a look inside Java classes**. Disponível em: <http://www.javaworld.com/jw-08-1997/jw-08-indepth.html>. Acesso em: 10 nov. 1999.

[OLI 98a] OLIVA, A. **Guaraná – uma arquitetura reflexiva**. **.** 1998. Disponível em: <http://www.sunsite.unicamp.br/~oliva/guarana/index.html>. Acesso em : 11 dez. 1999.

[OLI 98b] OLIVA, A.; BUZATO, L.E. **Guaraná**: A Tutorial**.** Campinas: Instituto de Computação, Unicamp, 1998. (Relatório técnico IC-98-31).

[OLI 98c] OLIVA, A. – **Guaraná**: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java. 1998. Dissertação (Mestrado em Ciência da Computação) – Instituto de Computação, Unicamp, Campinas.

[OLI 98d] OLIVA, A.; GARCIA, I.C.; BUZATO, L.E. **The Reflective Architecture of Guaraná.** Campinas: Instituto de Computação da Unicamp, 1998. Disponível em: <http://www.dcc.unicamp.br/~oliva/guarana>. Acesso em: 11 dez. 1999.

[OLI 99a] OLIVA, A.; BUZATO, L.E. **Composition of MetaObjects in Guaraná.** Campinas: Instituto de Computação, Unicamp, 1999.

[OLI 99b] OLIVA, A.; BUZATO, L.E. **Designing a Secure and Reconfigurable Meta-Object Protocol.** Campinas: Instituto de Computação, Unicamp, 1999. (Relatório técnico IC-99-08).

[PAP 86] PAPADIMITRIU, C.H. **The theory of database concurrency control.** New York: Computer Science Press, 1986.

[PAR 88] PARRINGTON, G.D.; SHRIVASTAVA, S. Implementing concurrency control in reliable distributed object-oriented systems. In: ECOOP, Oslo, 1998. **Proceedings...**

Oslo: Springer-Verlag, 1988.

[PRI 96] PRIEBE, J.; WANG, L. **Parallel and distributed computing – transactions** . Disponível em: <http://www.uncc.edu>. Acesso em: 5 jan. 2000.

[PU 88] PU, C.; KAISER, G.; HUTCHINSON, N. Split-transactions for open-ended activities. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 14., 1988. **Proceedings...** Los Angeles: Morgan Kaufmann, 1988.

[QUA 99] QUADROS, N. **Um arcabouço de software reflexivo para persistência de objetos em bases de dados heterogêneos**. 1999. Dissertação (Mestrado em Ciência da Computação) – Instituto de Computação, Unicamp, Campinas.

[RAN 98a] RANDELL, B. et al. **Coordinated atomic actions**: formal model, case study and system implementation. Newcastle: Department of Computing Science, University of Newcastle upon Tyne, 1998.

[RAN 98b] RANDELL, B. et al. **Coordinated atomic actions**: from concept to implementation. Newcastle: Department of Computing Science, University of Newcastle upon Tyne, 1998.

[REE 83] REED, D.P. Implementing atomic actions on decentralized data. **ACM Transactions on Computer Systems**, New York, v.1, n.1, 1983.

[RIE 96] RIEHLE, D.; ZÜLLOGHOVEN, H. **Understanding and Using Patterns in Software Development.** Disponível em: <http://www.citeseer.nj.nec.com/riehle96undertanding.html>. Acesso em: 12 dez. 2000.

[ROM 2000] ROMÁN, M.; KON, F.; CAMPBELL, R. **Reflective Middleware**: from your Desk to your Hand. Chicago: Department of Computer Science, University of Illinois at Urbana-Champaign, 2000.

[ROM 97a] ROMANOVSKY, A. et al. **Implementing synchronous coordinated atomic actions based on forward error recovery**. Newcastle: Department of Computer Science, University of Newcastle upon Tyne, 1997.

[ROM 97b] ROMANOVSKY, A.; ZORZO, A. **A distributed coordinated atomic actions schem*e***. Newcastle: Department of Computer Science, University of Newcastle upon Tyne, 1997.

[ROS 78] ROSENKRANTZ, D.J.; STERNS, R.E.; LEWIS, P.M. System Level Control for Distributed Database Systems. **ACM Trans on Database Systems**, New York, v.2, n.2, 1978.

[RUB 94] RUBIRA, C. M. F.; STROUD, R**. Forward and backward error recovery in C++.** (Technical Report Series n. 417). Disponível em: <http://www.cs.newcastle.ac.uk/events/ann...tracts/417.html>. Acesso em: 07 mar. 2001

[RUM 94] RUMBAUGH, J. et al. **Modelagem e projetos baseados em objetos.** Rio de Janeiro: Campus, 1994.

[SAC 97] SACHETT, R. **Um estudo das transações atômicas em ambientes de computação distribuída**. 1999. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[SEN 2001] SENRA, R. **Programação reflexiva sobre o MOP Guaraná**. 1998. Dissertação (Mestrado em Ciência da Computação) – Instituto de Computação, Unicamp, Campinas.

[SHA 88] SHA, L.; LEHOCZKY, J.P.; JENSEN, E.D. Modular concurrency control and

failure recovery. **IEEE Transactions on computers**, New York, v. 37, 1988.

[SHR 88] SHRIVASTAVA, S.K.; DIXON, G.N.; PARRINGTON, G.D**. Objects and actions in reliable distributed systems**. Newcastle: Computing Laboratory, University of Newcastle upon Tyne, 1988.

[SHR 89] SHRIVASTAVA, S.K.; DIXON, G.N.; PARRINGTON, G.D**. An Overview of the Arjuna Distributed Programming System**. Newcastle: Computing Laboratory, University of Newcastle upon Tyne, 1989.

[SHR 90] SHRIVASTAVA, S.; WHEATHER, S. **Implementing fault-tolerant distributed applications using objects and multi-coloured actions**. Newcastle: Computing Laboratory, University of Newcastle upon Tyne, 1990.

[SHR 93] SHRIVASTAVA, S.; McCUE, D.L. **Structuring fault-tolerant object systems for modularity in a distributed environment**. Nescastle: Department of Computing Science, University of Newcastle upon Tyne, 1993.

[SHR 94] SHRIVASTAVA,S. et al. **The Arjuna System programmers' guide v3.0 .** Nescastle: Department of Computing Science, University of Newcastle upon Tyne, 1994.

[SIL 96a] SILVA, A. et al. **Atomicity Policies using Design Patterns**. Disponível em: <http://www.esw. inesc.pt/~ars/ps/apap96.ps>. Acesso em: 14 mar. 2001.

[SIL 96b] SILVA, A.R.; PEREIRA, J.; MARQUES, J.A. **Customizable Object Recovery Pattern.** Disponível em <http://www-rodin.inria.fr/ ~pereira/pub.html>. Acesso em: 10 mar. 2001.

[SIB 99] SILBERSCHATZ, A.; KORTH, H.; SUDARSHAN, S. **Sistemas de Bancos de Dados**. São Paulo: Makron Books, 1999.

[SMI 82] SMITH, B.C. **Procedural Reflection in Programming Languages.** 1982. Tese de Doutorado - MIT, Cambridge.

[STR 91] STRIGINI, L.; Di GIANDORMENICO, F. Flexible schemes for application-level fault tolerance. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 10., 1991. **Proceedings...** Pisa: IEEE Press, 1991.

[STR 95a] STROUD, R.J.; WU, Z. **Using metaobject protocols to implement atomic data types**. Nescastle: Department of Computing Science, University of Newcastle upon Tyne, 1995. (Technical report series, n. 512).

[STR 95b] STROUD, R.J.; WU, Z. Using metaobject protocols to satisfy non-functional requirements. In: ADVANCES IN OBJECT-ORIENTED METALEVEL ARCHITECTURES AND REFLECTION. **Proceedings...** [S.l.] CRC Press, 1995.

[SUN 99] **The Java tutorial – a practical guide for programmers.** Disponível em : <http://java.sun.com/docs/books/tutorial/index.html>. Acesso em: 10 nov. 1999.

[TAN 92] TANENBAUM, A. **A modern operating system** Englewood Cliffs: Prentice Hall, 1992.

[TAT 99] TATSUBORI, M. **Open Java** . Disponível em: <http://www.hlla.is.tsukuba.ac.jp/~mich/openjava.html>. Acesso em: 05 jan. 2000.

[TEK 94] TEKINERDOGAN, B. **Design of an object-oriented framework for atomic transactions.** 1994. Dissertação de Mestrado - Universiteit Twente, Twente.

[WEI 85] WEIHL, W.E.; LISKOV. B. Implementation of resilient, atomic data types. **ACM Transactions on programming languages and systems**, New York, v.7, n. 2, p. 244-269, 1985.

[WHE 90] WHEATER, S.M. **Constructing reliable distributed applications using actions**

and objects. 1990. Dissertação de Mestrado - University of Newcastle, Newcastle.

[WU 95] WU, Z.; MOODY, K.; BACON, J.; STROUD, R.J. Data consistency in a distributed persistent object system. In: HAWAII INTERNATIONAL CONFERENCE IN SYSTEM SCIENCE, 28., 1995. **Proceedings...** Honolulu: Press, 1995.

[WU 97a] WU, Z.; SCHWIDERSKI, S. **Reflective Java: making Java even more flexible**. ANSA Phase III. UK, 1997. Disponível em: <http://www.ansa.co.uk/ANSATech/ANSAhtml/htmltree/all_title.html>. Acesso em 19 dez. 1999.

[WU 97b] WU, Z.; SCHWIDERSKI, S. **Reflective java.** Technical Report APM 1931.01, ANSA Phase III. UK, 1997. Disponível em: <http://www.ansa.co.uk/ANSATech/ANSAhtml/htmltree/all_title.html>. Acesso em 19 dez. 1999.

[ZOR 99] ZORZO, A. et al. **Using coordinated atomic actions to design dependable distributred object systems.** Newcastle: Department of Computing Science, University of Newcastle, 1999.