

Um Arcabouço para Suporte a Reconfiguração Dinâmica em Ambiente Java

Ricardo Koji Ushizaki

DISSERTAÇÃO APRESENTADA AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA DA
UNIVERSIDADE DE SÃO PAULO
PARA A OBTENÇÃO DO GRAU DE
MESTRE EM CIÊNCIAS.

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Fabio Kon

São Paulo, setembro de 2005.

Um Arcabouço para Suporte a Reconfiguração Dinâmica em Ambiente Java

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Ricardo Koji Ushizaki e aprovada pela comissão julgadora.

São Paulo, 04 de setembro de 2005.

Comissão julgadora:

- Prof. Dr. Fabio Kon (orientador) - IME/USP
- Prof. Dr. Francisco Reverbel - IME-USP
- Prof. Dr. Fabio Moreira Costa - INF-UFG

Agradecimentos

Inicialmente, agradeço especialmente ao meu orientador Professor Dr. Fabio Kon pela sua paciência e dedicação nesse meu projeto de mestrado. Sou grato pela sua confiança depositada em mim e por ter acreditado e me orientado durante todo o mestrado. Não fosse a sua orientação, esse projeto não teria sido realizado.

Agradeço à minha esposa Yuka e toda família pela enorme paciência e carinho dedicados durante o mestrado. Todos foram e sempre serão muito importantes para mim na minha vida. Especial obrigado ao meu futuro filho Naoto, que ainda nem nasceu e já nos proporciona momentos incríveis com seus pequenos chutes.

A todos os meus amigos que me apoiaram e confiaram em mim desde o início, contribuindo e discutindo várias idéias, muito obrigado.

E finalmente, agradeço aos meus colegas do GSD pelas animadas reuniões que participei e que ficarão na memória. Obrigado.

Resumo

Cada vez mais sistemas de software necessitam de uma alta disponibilidade de seus serviços exigindo um extenso tempo de vida de execução sem que haja paradas no fornecimento de seus serviços. Mas sistemas críticos não podem ser parados e alterados, pois há a possibilidade de existir vidas de pessoas em jogo ou, no caso de empresas, isso pode resultar em grandes prejuízos financeiros.

Sistemas de pequeno, médio e grande porte precisam ser reconfigurados através de manutenções, programadas ou não. A reconfiguração dinâmica consiste em modificar a configuração dos sistemas durante a sua execução, sem que haja a necessidade de paradas para implantar as modificações, contribuindo para aumentar a disponibilidade de seus serviços.

Nessa dissertação fizemos um estudo dos diversos mecanismos existentes de reconfiguração dinâmica e desenvolvemos um arcabouço em Java para dar suporte a reconfiguração de sistemas baseados em componentes. Estendemos o modelo existente de Configuradores de Componentes e definimos o conceito de políticas de reconfiguração a serem seguidas durante esse processo. A partir de nosso arcabouço criamos uma aplicação exemplo que demonstra a utilização da estrutura criada.

Abstract

Many software systems need high availability of their services demanding long execution lifetime without stopping the services. Mission critical systems cannot be stopped, as there is the possibility of risking people's lives, or incurring in great financial losses in the case of enterprise systems.

Small, medium and large systems need to be reconfigured for maintenance, be it scheduled or not. Dynamic reconfiguration consists in modifying system configuration at runtime, without the need to stop its execution, thus increasing the availability of its services.

In this thesis, we studied several existing mechanisms for dynamic reconfiguration and developed a framework in Java to give support for component-based system reconfiguration. We extended the Component Configurator model and defined the concept of dynamic reconfiguration policies to be followed during this process. With our framework, we developed an application to demonstrate how to use the structure we created.

Sumário

1	Introdução	1
1.1	Motivação	2
1.2	Reconfiguração Dinâmica	3
1.3	Organização da Dissertação	5
2	Trabalhos Relacionados	7
2.1	Substituição Dinâmica de Módulos no Argus	7
2.2	Configuração Dinâmica de Sistemas Distribuídos com Conic	8
2.3	POLYLITH	9
2.4	Um Serviço de Reconfiguração Dinâmica para CORBA	10
2.5	Reconfiguração Dinâmica de Aplicações Java Baseadas em Componentes	10
2.6	<i>Reconfiguração Dinâmica Transparente para CORBA</i>	12
2.7	<i>O Sistema Operacional 2k</i>	13
2.8	<i>Generic Connector e LuaSpace</i>	14
2.9	Adaptação Dinâmica de Sistemas Distribuídos	15
2.10	JBoss e <i>Java Management Extensions</i> (JMX)	16
2.11	Quadro Comparativo	18
3	Reconfiguração Dinâmica de Componentes	21
3.1	Etapas da Reconfiguração Dinâmica	21
3.2	Requisitos para Reconfiguração Dinâmica	22
3.3	Mecanismos de Reconfiguração	25
3.3.1	Mecanismo de Bloqueio de Chamadas	26
3.3.2	Mecanismo de Indireção de Chamadas	28
3.3.3	Mecanismo de Pontos de Reconfiguração	29

Sumário

4	Configuradores de Componentes	33
4.1	O que são Componentes	33
4.2	Modelo dos Configuradores de Componentes	34
5	Extensão do Modelo	43
5.1	Objetos Reconfiguráveis	43
5.2	Transferência de Estado	45
5.3	Substituição de Implementação	46
5.4	Políticas de Reconfiguração Dinâmica	48
5.4.1	Política de Consistência Fraca	49
5.4.2	Política de Consistência Média	50
5.4.3	Política de Consistência Forte	50
5.4.4	Diferenças entre as Políticas	51
5.4.5	Vantagens e Desvantagens	52
5.4.6	Implementação das Políticas	52
5.5	Aplicação Gráfica	53
5.6	Análise de desempenho	61
5.7	Considerações Finais	62
6	Conclusões	65
6.1	Contribuição	65
6.2	Trabalhos Futuros	68
	Referências Bibliográficas	69
	Referências a Sítios na Internet	72
	Índice Remissivo	74

Lista de Figuras

1.1	Cenário Típico de Sistemas sem Reconfiguração Dinâmica	4
1.2	Cenário de Sistemas com Reconfiguração Dinâmica	5
3.1	Exemplo de substituição de componente com estado.	23
3.2	Componente atingindo estado seguro.	25
3.3	Mecanismo de bloqueio no lado do cliente	27
3.4	Mecanismo de bloqueio no lado do componente	27
3.5	Mecanismo de bloqueio entre o cliente e o componente	28
4.1	Configurador de Componente associado ao seu Componente	35
4.2	Dependência entre Configuradores de Componentes	35
4.3	Reificação das Dependências da Componente	36
4.4	Diagrama de Classes do Modelo de Configuradores de Componentes	37
5.1	Visão do Modelo de Objetos Reconfiguráveis	44
5.2	Diagrama de Classes da Aplicação Gráfica.	54
5.3	Aplicação Gráfica no seu estado inicial.	55
5.4	Substituição dinâmica do componente círculo por um quadrado.	57
5.5	Resultado da substituição do componente círculo por um quadrado.	57
5.6	Adicionando ganchos para indicar dependências.	58
5.7	Adicionando novo componente retângulo arredondado e dependências.	58
5.8	Notificando clientes para alteração de cor.	60
5.9	Resultado final da notificação de alteração de cor.	60

Lista de Figuras

Lista de Tabelas

2.1	Tabela Comparativa	19
5.1	Tabela de Tempo Médio de Reconfiguração (em milissegundos)	62
6.1	Tabela de Linhas de Código (LOC)	66
6.2	Tabela de Linhas de Código Sem Comentários (NLOC)	67

Lista de Tabelas

Listagens de Código

3.1	Uso de Pontos de Reconfiguração	31
3.2	Mapeando estado com Pontos de Reconfiguração	31
4.1	Interface Java do Configurador de Componentes	38
5.1	Implementação do método <code>replaceImplementation()</code>	46
5.2	Implementação da classe <code>DynamicPolicy</code>	52
5.3	Exemplo de chamada do método <code>replaceImplementation()</code> na aplicação gráfica .	59

Listagens de Código

Capítulo 1

Introdução

Com a diversidade crescente dos sistemas computacionais modernos, como telefones celulares e PDAs interagindo com computadores pessoais e servidores, ficou mais difícil a criação, desenvolvimento e manutenção de sistemas que atendam às necessidades dos usuários desses dispositivos. Aumentou-se a mobilidade, heterogeneidade e complexidade dos programas existentes. Com o avanço tecnológico nos níveis de hardware e software, essa diversidade e complexidade tendem a aumentar cada vez mais.

Sistemas modernos possuem alto grau de dinamismo. Há uma dinâmica no nível de hardware quanto à disponibilidade de CPU, memória e banda de rede, que podem variar dependendo do hardware instalado e das aplicações executando no sistema. A disponibilidade de conexão na computação móvel é outro fator que eleva o grau de dinamismo nesses sistemas. Quanto maior o grau de dinamismo e integração das várias partes do sistema, maior a complexidade e heterogeneidade existente.

No nível de software a dinâmica é maior. Atualizações nos protocolos e APIs dos sistemas são freqüentes, sistemas operacionais estão em constante atualização e modificação de seu código através de *patches*, alterando a estrutura interna desses sistemas e aumentando a sua complexidade, o que torna mais difícil a tarefa de executar programas já existentes com novas modificações a serem implantadas.

Devido a todas essas mudanças e dificuldades, novas tecnologias de desenvolvimento de sistemas tornaram-se necessárias. Uma delas é a Programação Orientada a Objetos (POO), que surgiu propondo novas técnicas como a modelagem dos sistemas em objetos, utilizando práticas como herança, polimorfismo e encapsulamento dos dados em objetos, procurando buscar uma melhor organização do código tornando a tarefa de desenvolvimento e manutenção dos sistemas um pouco menos complexa.

1 Introdução

Com o avanço das técnicas de POO, surge a tecnologia de desenvolvimento de sistemas baseados em componentes [Szyperski, 2002] (*component-based systems*). Esta tecnologia surgiu para dar suporte a essas novas mudanças permitindo o desenvolvimento de sistemas complexos da forma mais confiável possível, combinando-se componentes de prateleira (*off-the-shelf components*) com módulos do sistema que possam ser reutilizados em diferentes contextos.

O desenvolvimento de sistemas baseados em componentes, ou “programação baseada em componentes”, tem como principal característica a separação entre a interface e a implementação dos componentes, que são a unidade de distribuição e implantação (*deployment*) desses sistemas. Exemplos de arquiteturas baseadas em componentes são CCM (*CORBA Component Model*) [OMG, 2002], Microsoft DCOM (*Distributed Component Object Model*) [Microsoft, 1998], Microsoft .NET [Microsoft, 2000] e *Enterprise JavaBeans* [Sun Microsystems, 2003], que definem as abstrações necessárias e interfaces para as invocações de suas operações.

Inicialmente, esses sistemas eram desenvolvidos de modo a esconder de seus usuários detalhes relativos ao ambiente de execução dos componentes, tentando tornar o mais transparente possível a tarefa de integração das aplicações. Entretanto, com o aumento da demanda por aplicações multimídia, de tempo real e distribuídas como na computação móvel, ficou mais difícil não detalhar requisitos específicos sobre as condições de execução, como disponibilidade de banda de rede, conexão e segurança.

Para piorar o cenário, a construção robusta e eficiente de componentes não é uma tarefa trivial. Componentes são escritos por programadores diferentes que geralmente trabalham em grupos distintos utilizando suas próprias metodologias. Ainda, requisitos não-funcionais do sistema, como disponibilidade de recursos e desempenho, sempre apresentam variações durante o tempo de vida dos programas, tornando mais complexa a tarefa de se desenvolver componentes facilmente integráveis aos sistemas existentes. Como mudanças no ambiente são frequentes, isso causa um alto impacto no desempenho dos sistemas.

1.1 Motivação

Nesse contexto de desenvolvimento de sistemas, surgiram extensões [Bidan et al., 1998] [Almeida et al., 2001b] [Kon, 2000] [da Silva e Silva, 2003] aos sistemas baseados em componentes que permitissem uma adaptação dinâmica dos sistemas em execução, sem a diminuição da Qualidade de Serviço (*QoS - Quality of Service*). Surgiram também conferências como a ICCDS (*International Conference on Configurable Distributable Systems*) [1stICCDS, 1992, 2ndICCDS, 1994, 3rdICCDS, 1996, 4thICCDS, 1998] e também o *Workshop on Adaptive and Reflective Middleware*

[RM2000, 2000, RM2003, 2003, RM2004, 2004] para discutir os problemas referentes na área de reconfiguração dinâmica.

Sistemas atuais oferecem pouco ou nenhum suporte para gerenciamento e adaptação a essas mudanças e geralmente necessitam de uma intervenção manual por operadores ou usuários da aplicação. A intervenção manual torna difícil a tarefa de gerenciar ambientes onde a heterogeneidade de sistemas é grande, como existe em grandes corporações ou redes acadêmicas, resultando em um trabalho de configuração multiplicado pelo número de plataformas presentes no sistema.

Os componentes dos sistemas devem estar preparados para se adequarem rapidamente a uma reconfiguração dinâmica do sistema a fim de não agravar ainda mais o impacto no seu desempenho. Portanto, devem oferecer suporte a uma auto-configuração e adaptação evitando perda da Qualidade de Serviço, sendo sensíveis às mudanças que possam ocorrer durante sua execução, como novos serviços e novas interfaces que venham a ser disponibilizados ao longo do tempo de vida dos programas.

Novos serviços surgem à medida que os sistemas evoluem para atender novos requisitos e funcionalidades. Em muitos casos os componentes já existentes ficam obsoletos, devendo ser substituídos por novos que atendam melhor essas funções. Em outros casos, novos componentes surgem com novas interfaces. O mecanismo de configuração do sistema deve ser flexível o bastante para incorporar essas variações.

Essa configuração e adaptação do sistema deverá ser executada dinamicamente para que não haja perda de desempenho e nem dos serviços oferecidos pela aplicação. Sistemas que não oferecem suporte a uma reconfiguração dinâmica devem ser parados, reconfigurados e reinicializados para que a nova configuração seja aplicada. Em muitos casos isso é inviável. Paradas em sistemas podem desde resultar em prejuízos financeiros para empresas e até colocar vidas em perigo no caso de sistemas críticos como os de hospitais.

A Figura 1.1 mostra esse cenário típico de necessidade de parada do sistema. A linha contínua indica que o sistema está em execução normalmente. Mas em um determinado ponto surge a necessidade de atualização do sistema, e por isso o sistema deve ser parado e atualizado, conforme mostra a linha tracejada na Figura. Feita a atualização, o sistema é reiniciado e volta a prover os seus serviços.

1.2 Reconfiguração Dinâmica

O termo reconfiguração dinâmica se aplica ao ato de mudar a configuração da aplicação durante a sua execução, trazendo pouca ou nenhuma interrupção no provimento de serviços do

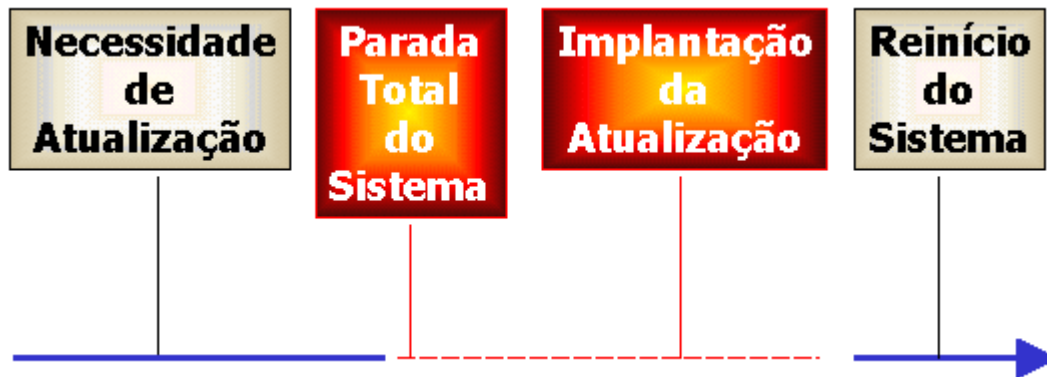


Figura 1.1: Cenário Típico de Sistemas sem Reconfiguração Dinâmica

sistema. Utilizando a reconfiguração dinâmica não existe a necessidade de se parar o sistema para reconfigurá-lo. Exemplos de reconfiguração dinâmica são adicionar novos componentes ao sistema, remoção de componentes já existentes ou substituição por melhores implementações.

A reconfiguração dinâmica tem como objetivo principal permitir a evolução dos sistemas em tempo de execução, sem que haja interrupções dos seus serviços, trazendo pouco ou nenhum impacto negativo ao desempenho dos sistemas.

A Figura 1.2 mostra o cenário de um sistema com reconfiguração dinâmica. Surgindo a necessidade de atualização, o sistema é reconfigurado dinamicamente e continua a sua execução sem ter que parar os seus serviços. Apenas as partes afetadas têm uma interrupção parcial momentânea durante a reconfiguração.

Durante a reconfiguração dinâmica existem as seguintes operações básicas: criação, remoção ou substituição de componentes, criação e remoção de referências entre os componentes e a transferência de estado entre eles. Isso tudo deve ser executado preservando-se a consistência do sistema e ao mesmo tempo introduzindo o mínimo de interrupção dos serviços oferecidos pela aplicação.

A maioria das aplicações possuem estado que não deve ser perdido e muito menos corrompido após a atualização do sistema. Devido a esse fato diversos mecanismos foram propostos para garantir a consistência do sistema, como por exemplo bloquear as chamadas ao componente sendo reconfigurado para que o seu estado não se altere durante a reconfiguração.

O trabalho descrito nesta dissertação visa criar uma extensão do modelo de Configuradores de Componentes [Kon and Campbell, 2000] de modo a implementar diferentes políticas de manipulação das dependências entre componentes durante o processo de reconfiguração dinâmica. É através dessas políticas que definimos qual o comportamento dos Configuradores de Componentes

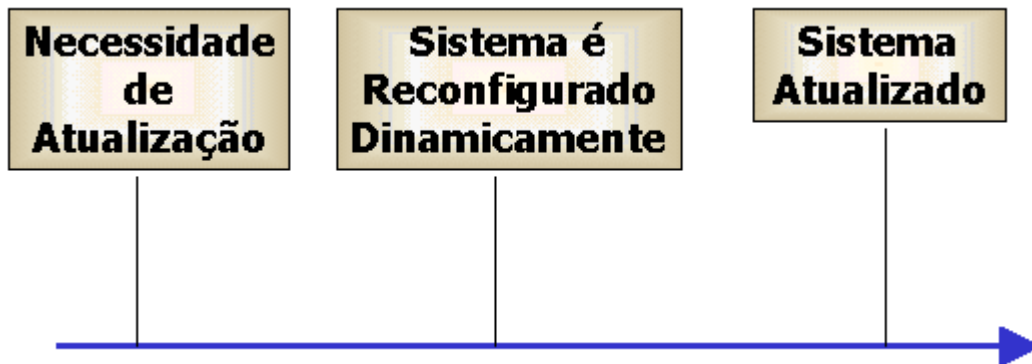


Figura 1.2: Cenário de Sistemas com Reconfiguração Dinâmica

no momento da reconfiguração, por exemplo, se preocupar ou não com o estado do componente. Os Configuradores de Componentes são objetos que representam as dependências dinâmicas entre os componentes que estão sendo executados. Deste modo, durante a reconfiguração, podemos saber em tempo de execução quais são as partes dependentes envolvidas para então agir da melhor forma possível.

Nesta dissertação fizemos também um estudo dos diversos mecanismos propostos até hoje para se executar a reconfiguração dinâmica. Esses mecanismos são importantes pois garantem que o sistema preserve a sua consistência durante e após a reconfiguração.

Foi criado em nossa pesquisa um arcabouço de Objetos Reconfiguráveis (*Reconfigurable Objects*) implementando interfaces em Java [Java, [sítio](#)] para a execução da reconfiguração de componentes, apoiando-se nas interdependências dos componentes reificadas pelos Configuradores de Componentes.

Pretendemos com esse arcabouço contribuir para a área de reconfiguração dinâmica criando classes genéricas o suficiente que auxiliem no processo de reconfiguração.

A partir desse arcabouço criamos uma aplicação gráfica de exemplo para demonstrar como é feita a reconfiguração e definimos quais políticas são utilizadas para se reconfigurar um componente no momento da reconfiguração.

1.3 Organização da Dissertação

Esta dissertação está organizada na seguinte forma: no Capítulo 2 estão os trabalhos relacionados na área de reconfiguração dinâmica. Os detalhes sobre reconfiguração dinâmica de sistemas baseados em componentes e seus mecanismos de criação, remoção e substituição são descritos no

1 Introdução

Capítulo 3. O Capítulo 4 explica o funcionamento dos Configuradores de Componentes, cujo modelo de componentes nós estendemos em nossa pesquisa. A descrição e funcionamento do arcabouço criado de Objetos Reconfiguráveis bem como a definição das políticas de consistência e a aplicação gráfica de exemplo são apresentados em detalhes no Capítulo 5. O Capítulo 6 conclui a dissertação e propõe áreas de pesquisa para trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Este capítulo mostra um histórico dos trabalhos mais relacionados à nossa pesquisa. Alguns criam um barramento de dados [Hofmeister and Purtilo, 1993] por onde é possível administrar todas chamadas e reconfigurações do sistema, outros criam arcabouços [Kon et al., 2000a] gerenciando as interdependências entre os componentes do sistema e, com essa informação, executam o processo de reconfiguração da melhor forma possível.

2.1 Substituição Dinâmica de Módulos no Argus

Um dos primeiros trabalhos a tratar o problema da reconfiguração dinâmica foi proposto em 1983 por Toby Bloom [Bloom, 1983]. Em sua tese de doutorado no MIT, Bloom investiga as questões pertinentes à substituição dinâmica de módulos do sistema em execução. Ele define um modelo formal que explica quando é possível executar a substituição de um módulo, quais são as condições para que essa substituição continue mantendo o sistema correto e quais restrições existem durante uma substituição dinâmica. Bloom utiliza o sistema distribuído Argus, que tem como característica dar suporte a aplicações de longa duração (*long-lived applications*) e distribuídas geograficamente em uma rede heterogênea.

Programas no Argus são compostos por módulos chamados *guardians* (guardiões). Um guardião provê algum serviço ou encapsula algum recurso, e contém internamente um conjunto de processos e objetos. Guardiões são os nós lógicos do sistema que se comunicam via troca de mensagens através de uma comunicação independente de localização: seus clientes não precisam saber em qual nó físico do sistema estão os guardiões.

As operações de um guardião podem ser remotamente invocadas e são chamadas de *handlers*. Clientes utilizam os guardiões passando mensagens a cada um dos *handlers* existentes. Cada

2 Trabalhos Relacionados

guardião possui um estado, composto pelos objetos compartilhados pelos seus *handlers*. O Argus garante que um estado estável sobrevive a falhas do sistema e que todas as ações nos objetos que compõem o estado sejam atômicas.

O modelo de Bloom formaliza quais substituições são permitidas, que são aquelas que preservam ou estendam as funcionalidades do módulo substituído, e define um mecanismo que permite a um usuário manipular manualmente os guardiões para executar dinamicamente as substituições.

As substituições necessitam de um *lock* exclusivo nos guardiões afetados e para evitar *deadlocks* no sistema (por exemplo, se um guardião *A* precisa do *lock* do guardião *B*, e este tenta obter o *lock* do guardião *A*) as chamadas dos clientes são abortadas e seus guardiões se tornam inativos. Entretanto Bloom não detalha uma implementação para esses casos, e cita a necessidade de uma interface de mais alto nível para o usuário executar essas substituições.

2.2 Configuração Dinâmica de Sistemas Distribuídos com Conic

Em 1985, Jeff Kramer e Jeff Magee do Imperial College descreveram como configurar dinamicamente um sistema distribuído [Kramer and Magee, 1985] especificado em Conic [Magee et al., 1989], um ambiente de programação que provê um conjunto de ferramentas para compilar, configurar, depurar e executar programas, separando a programação de módulos individuais (*programming in the small*) da configuração do sistema como um todo (*programming in the large*). É através desse ambiente que são definidos os módulos e as conexões entre eles.

As reconfigurações são especificadas através de mensagens passadas aos módulos como comandos (*link, unlink, create*). Existe um *Configuration Manager* (CM) que traduz esses comandos para o sistema operacional e os executa, checando se as mudanças obedecem à interface do módulo reconfigurado.

O sistema não dá suporte à transferência ou migração de estado, e nem garante sua consistência, já que não há atomicidade nas reconfigurações.

Entretanto, [Kramer and Magee, 1990] argumenta que o CM tem a capacidade de deixar um componente em modo “passivo”, ou seja, esse componente pára de iniciar novas transações e espera até terminar aquelas já iniciadas. Dada essa capacidade, o CM pode tornar “passivos” os componentes afetados antes de iniciar a reconfiguração.

Os autores definem também o termo “*quiescence*” como parte do sistema para a reconfiguração dinâmica do componente. Esse termo significa um estado de um componente em modo “passivo” e cujas conexões a ele também são “passivas”, ou seja, todas as transações pendentes dessa instância do componente estão completadas ou serão finalizadas sem que ele inicie novas transações até a

sua reconfiguração. Nesse ponto, diz-se que a aplicação está em estado consistente e congelada, pois possui transações completadas e nenhum componente irá iniciar uma nova transação.

Existe também o conceito de transações dependentes, que envolvem uma ou mais transações subseqüentes, ou seja, uma depende da finalização de outra. Desse modo, enquanto houver alguma transação pendente aguardando a finalização de outras, o sistema requer que o iniciador da transação de reconfiguração seja informado de quando ela se completar para que se inicie a reconfiguração dinâmica do componente.

2.3 POLYLITH

POLYLITH [Purtilo, 1990, Hofmeister, 1994] é um sistema distribuído para interconectar módulos de aplicações para uso em ambientes heterogêneos desenvolvido por Hofmeister e Purtilo na Universidade de Maryland. O desenvolvedor da aplicação provê uma especificação, descrevendo a estrutura modular do sistema. Essa especificação descreve os atributos de cada módulo, incluindo suas interfaces, e define as ligações entre elas. Hofmeister e Purtilo estenderam o trabalho de Kramer e Magee incluindo suporte para capturar e restaurar o estado de processos no ambiente do POLYLITH.

Dadas a especificação da arquitetura da aplicação juntamente com as implementações dos módulos, POLYLITH utiliza um “empacotador” (*packager*) chamado Surgeon [Hofmeister et al., 1992] que é responsável por empacotar e executar os processos, analisar as interfaces dos módulos, manter o sistema em estado consistente, sincronizar e transferir dados durante a comunicação. Graças ao conjunto de informações detalhadas pelo programador na especificação da arquitetura da aplicação, POLYLITH é capaz de reconfigurar dinamicamente a aplicação e restaurar o estado do sistema.

Utilizam também o termo *quiescence* como um pré-requisito para que seja executada a reconfiguração dos módulos, sendo que as mensagens recebidas durante o processo de reconfiguração são colocadas em um *buffer* e são entregues após a inicialização do módulo reconfigurado.

Neste modelo, existe uma separação entre a configuração da aplicação e a implementação dos módulos, ou seja, separa-se a parte funcional da parte de interação entre os módulos. A informação de configuração direciona o início da aplicação, e continua acessível durante o tempo de execução. Interações são descritas nessa configuração, e toda mudança dinâmica entre as interações dos módulos ocorre no nível de configuração.

2.4 Um Serviço de Reconfiguração Dinâmica para CORBA

Bidan et al. [Bidan et al., 1998] apresentaram um serviço de reconfiguração dinâmica baseado em CORBA. Foi aproveitado o fato de que ambientes de programação escritos para CORBA já possuem uma clara separação entre os conceitos funcionais e estruturais, além de utilizarem componentes que se comunicam utilizando um ORB (*Object Request Broker*) [OMG, 2004]. Ainda, várias funcionalidades (controle do ciclo de vida de componentes, persistência, transações etc) podem ser integradas a um ORB como um serviço CORBA (*Common Object Services - COS*[OMG, 1998]) e podem ser usadas diretamente pelos componentes nas aplicações.

Os autores propõem a criação de um Gerenciador de Reconfiguração Dinâmica (*Dynamic Reconfiguration Manager - DRM*) que interage com os objetos durante a reconfiguração. Ao invés de tornarem os componentes passivos, o DRM torna passivas as conexões entre os componentes a serem reconfigurados, ou seja, pede a cada cliente do componente que não faça mais requisições a ele. O mesmo é feito para as conexões que saem do componente. Essa operação é chamada de `passivateLink`.

Para que isso tudo seja possível, os componentes devem ser “reconfiguráveis” e são obrigados a implementar uma interface `RO_Object`, que possui primitivas para a reconfiguração, como a que torna as conexões passivas, as que transferem estado do componente antigo para o novo e outras para notificar o início e fim da reconfiguração.

O algoritmo de reconfiguração é construído estendendo as semânticas do serviço CORBA de ciclo de vida dos objetos (*CORBA Life Cycle COS*), e os autores alegam que a reconfiguração é ótima, ou seja, ocorre o mínimo possível de interrupção durante essa operação, já que apenas as conexões se tornam passivas, bloqueando somente os clientes que as utilizam, e não todos os objetos do sistema.

Mas se ações internas do componente, tais como *timers* que são ativados de tempos em tempos, alterarem o seu estado durante a reconfiguração esse modelo de serviço não consegue “tornar passivas” essas ações.

2.5 Reconfiguração Dinâmica de Aplicações Java Baseadas em Componentes

No ano de 2000 no *Massachusetts Institute of Technology*, Ziqiang Tang em seu trabalho de mestrado [Tang, 2000] apresentou um modelo de programação para reconfigurar dinamicamente aplicações baseadas em componentes utilizando pontos de reconfiguração (*Reconfiguration*

Points), que são pontos específicos no código dos componentes indicando quando é possível realizar uma evolução, termo utilizado pelo autor para a substituição de implementação.

O autor argumenta que aguardar pelo momento ideal para reconfigurar o componente em alguns casos é inviável, já que métodos de longa execução podem atrasar a reconfiguração significativamente, ou pior ainda, podem existir métodos que nunca terminem enquanto a aplicação não se finalizar.

Para que ocorra a reconfiguração é necessário esperar que o método sendo processado termine toda computação e não invoque e nem seja invocado mais nenhum outro método garantindo que o estado do componente não seja alterado durante a reconfiguração.

Mas utilizando pontos de reconfiguração é possível indicar no meio do código original qual o momento ideal para se executar uma reconfiguração sem esperar que o método retorne. Neste ponto garante-se que o componente está em um estado seguro para ser reconfigurado. Esse mecanismo é explicado com mais detalhes na Seção 3.3.3.

Uma diferença desse mecanismo refere-se à necessidade de mapear não só o estado do componente como também o estado das variáveis locais do método, para que a sua execução continue corretamente a partir daquele ponto de reconfiguração na nova implementação.

Nesse modelo, é criada uma estrutura chamada de *component mold*, um molde de componente que possui duas interfaces: externa e interna. A interface externa possui as declarações de construtores e métodos do componente, enquanto que a interna provê as abstrações necessárias para o mapeamento do estado entre as implementações (atual e futuras) do componente. Essa interface interna exige que as implementações do componente definam funções chamadas *encode* e *decode*, permitindo que suas instâncias transformem o estado da implementação atual para a futura e vice-versa.

O modelo de [Tang, 2000] também cria novas palavras-chave na linguagem Java para descrever e implementar a evolução dos componentes. Essas palavras são:

- ***component*** declara um novo molde de componente, de forma análoga à declaração de classe ou interface em Java;
- ***encode*** define um trecho de código para “exportar” o estado do componente;
- ***decode*** define um trecho de código para “importar” o estado do componente;
- ***fulfills*** declara que um objeto implementa um molde de componente;
- ***reconfigurables*** indica que um método possui um ponto de reconfiguração (análogo à palavra-chave *throws* em Java);

2 Trabalhos Relacionados

- *reconfigurable* indica um ponto de reconfiguração no meio de um método;

No entanto, o autor não chega a criar nenhuma implementação na linguagem Java para validar esse modelo na prática, reconhecendo que seria desejável criar um compilador ou uma máquina virtual que permitisse o desenvolvimento de aplicações reconfiguráveis baseadas nesse modelo, além de prover uma compilação independente de linguagem que dê suporte à integração de componentes em outras linguagens.

O seu modelo também não permite que novas implementações alterem a interface do componente (por exemplo, adicionando novos métodos), pois argumenta que a evolução do sistema deva ser *backward-compatible*, sendo possível reconfigurar o componente e voltar à implementação original do sistema. O problema é que essa abordagem resulta em componentes com interface estática durante todo o seu tempo de vida.

A diferença desse trabalho com o nosso arcabouço é o mecanismo adotado pelo autor para executar a reconfiguração dinâmica, chamado de pontos de reconfiguração. A nossa abordagem utiliza bloqueios de chamadas aos métodos.

2.6 *Reconfiguração Dinâmica Transparente para CORBA*

Em [Almeida et al., 2001b, Almeida et al., 2001a] é proposto uma arquitetura para um Serviço de Reconfiguração Dinâmica em CORBA, que permite a reconfiguração de um sistema em execução com o máximo de transparência possível tanto para o cliente quanto para o servidor da aplicação. O trabalho vai além da abordagem de Bidan et al. descrito na seção 2.4 adicionando suporte para:

- chamadas reentrantes no sistema;
- substituição atômica de múltiplos objetos;
- maior transparência utilizando extensões do ORB do CORBA.

A arquitetura é composta por um *Reconfiguration Manager*, um *Location Agent* e *Reconfiguration Agents*. O *Reconfiguration Manager* delega a criação e remoção de objetos a *factories* (fábricas de objetos reconfiguráveis), gerencia o registro e saída de objetos interagindo com o *Location Agent* e coordena os *Reconfiguration Agents* para levar o sistema a um estado seguro para ser reconfigurado.

No caso de substituição de um objeto, o *Reconfiguration Manager* delega a criação do novo objeto para a *Factory*, e então o *Manager* notifica o objeto a ser substituído para iniciar a

reconfiguração. O *Reconfiguration Agent* notificará então o *Manager* quando for possível dar início à reconfiguração. Ocorre então a transferência do estado do objeto antigo para o novo e a referência para esse novo objeto é registrada no *Location Agent*.

A aplicação cliente executa as requisições normalmente através do ORB (*Object Request Broker*) CORBA. O ORB cliente é responsável por enviar as requisições para o ORB servidor, que repassa as requisições para o objeto alvo.

Mas, no caso de ocorrer uma reconfiguração, o *middleware* enfileira as requisições para o objeto alvo enquanto o *Reconfiguration Manager* executa a reconfiguração. Neste caso, o ORB do servidor notifica o ORB cliente sobre a reconfiguração. Ao final, o *Reconfiguration Manager* notifica o ORB cliente, que refaz as requisições para o novo objeto alvo registrado no *Location Agent*.

Todas as operações obedecem as especificações CORBA e portanto não necessitam de alterações nesse padrão. Esse trabalho se diferencia do nosso arcabouço por utilizar um componente gerenciador (*Reconfiguration Manager*) que centraliza e coordena todas as reconfigurações de cada componente.

2.7 O Sistema Operacional 2k

Pesquisadores da Universidade de Illinois criaram um sistema operacional distribuído chamado 2K [Kon et al., 2000a, Kon et al., 2005, 2K, [sítio](#)]. Essa arquitetura foi criada para resolver os problemas de gerenciamento de recursos em redes heterogêneas, adaptação dinâmica e configuração de aplicações distribuídas baseadas em componentes.

O sistema 2K oferece uma visão orientada a objetos do ambiente computacional distribuído, utilizando objetos CORBA para representar recursos de software e hardware distribuído. Já os serviços do sistema operacional distribuído, como serviço de nomes, são exportados como serviços CORBA. Quando um serviço é instanciado, as entidades que constituem aquele serviço são carregadas.

O 2K segue um modelo “*What You Need Is What You Get*” (*WYNIWYG*), ou seja, o sistema se auto-configura carregando um conjunto minimal de componentes necessárias para executar as aplicações da forma mais eficiente possível.

O Serviço de Configuração Automática (*2K Automatic Configuration Service*) gerencia os pré-requisitos e as dependências dinâmicas entre os componentes. Os pré-requisitos especificam o que é preciso para que um componente consiga ser instanciado e executado corretamente. Configuradores de Componentes [Kon, 2000] são utilizados no 2K para representar as dependências

2 Trabalhos Relacionados

dinâmicas entre os componentes carregados no sistema. Com essa informação o sistema pode referenciar seus próprios requisitos selecionando diferentes componentes que atendam às necessidades do momento.

Uma das principais partes do *2K* é o *dynamicTAO* [Kon et al., 2000b], uma extensão do ORB TAO que permite reconfigurar dinamicamente o motor interno do ORB e aplicações sendo executadas em cima dele.

2.8 *Generic Connector e LuaSpace*

Em [Batista and Carvalho, 1999] é apresentado um mecanismo para reconfiguração dinâmica de sistemas baseados em componentes chamado de *Generic Connector* que dá suporte a tolerância a falhas. Este mecanismo utiliza o ambiente de desenvolvimento de componentes chamado *LuaSpace* [Batista and Rodriguez, 2000], que combina CORBA com a linguagem de *script* procedural Lua [Jerusalimschy et al., 1996].

O mecanismo proposto por seus autores oferece suporte a uma seleção dinâmica de componentes em tempo de execução para tolerância a falhas. Essa seleção dinâmica utilizada pelo *Generic Connector* utiliza como parâmetro de busca a assinatura de métodos dos componentes, tornando possível a configuração da aplicação sem um prévio conhecimento de seus componentes.

É o próprio *Generic Connector* que faz a chamada de métodos dos componentes e devolve seus resultados. O objetivo do suporte a tolerância a falhas é garantir que uma chamada de método feita através do *Generic Connector* seja livre de falhas.

A aplicação é escrita na linguagem Lua e pode ser composta por componentes implementados em qualquer linguagem que tenha suporte a CORBA. É utilizado também o *LuaOrb* [Cerqueira et al., 1999], baseado na DII (*Dynamic Invocation Interface*) do CORBA, que prevê acesso dinâmico aos componentes CORBA como se fossem qualquer outro objeto Lua.

O *Generic Connector* funciona como um proxy para as chamadas de métodos dos componentes. Assim, quando um método é chamado, o interpretador de Lua intercepta a chamada e invoca implicitamente o *Generic Connector*. Este por sua vez faz uma busca para localizar os componentes que possuem o método chamado. Essa busca pode ser feita tanto no repositório padrão (*Naming ou Trading Services*) ou em uma tabela de configuração interna do próprio *Generic Connector*.

Após encontrado o componente que possui o serviço desejado, o *Generic Connector* monta uma seqüência de comandos escritos em Lua para criar um proxy para o componente em questão, ativando o serviço. Por último, o *Generic Connector* registra o método na sua tabela de confi-

guração, executa a chamada e retorna o resultado.

O mecanismo utilizado pelo *Generic Connector* permite configurar dinamicamente a aplicação, pois toda chamada de método é indirecionada, permitindo substituir um componente por outro de implementação diferente, apenas atualizando a tabela de configuração interna.

Deste modo é possível configurar a aplicação como um conjunto de serviços sem ter conhecimento prévio dos componentes que irão implementar o serviço. Esse mecanismo possui um suporte a reconfiguração dinâmica, pois diferentes invocações do mesmo serviço podem resultar na seleção de diferentes componentes.

O diferencial desse trabalho em relação ao nosso está no uso de uma linguagem própria (Lua) para a reconfiguração, além da interceptação de todas chamadas de métodos dos componentes analisadas pelo mecanismo do *Generic Connector*.

2.9 Adaptação Dinâmica de Sistemas Distribuídos

Em [da Silva e Silva, 2003, da Silva e Silva et al., 2003] é apresentado um arcabouço de apoio ao desenvolvimento de aplicações distribuídas adaptáveis dinamicamente. Francisco José da Silva e Silva desenvolveu em seu trabalho de doutorado uma estrutura para dar suporte a adaptação dinâmica, criando módulos que executam a monitoração do sistema, analisam e decidem se é necessário reconfigurar a aplicação, e executam a reconfiguração dinâmica.

O seu trabalho difere da nossa proposta de dissertação pois foca em adaptação dinâmica, enquanto que a nossa pesquisa se limita a desenvolver um arcabouço para reconfiguração dinâmica, que é um dos aspectos da adaptação dinâmica.

O arcabouço descrito em [da Silva e Silva, 2003] possui três partes principais:

- **Monitoração** responsável pelo monitoramento das entidades do sistema,
- **Detecção de Eventos** analisa os dados coletados pela monitoração, identificando os eventos que exijam reconfigurações no sistema,
- **Reconfiguração Dinâmica** verifica as pré-condições de adaptação dinâmica e aplica as ações de reconfiguração no sistema.

A **monitoração** é dividida em quatro partes. A primeira é responsável por interceptar as interações entre objetos, utilizando *interceptadores* descritos no padrão CORBA inseridos entre o objeto cliente e servidor, extraindo informações úteis de cada chamada de método e armazenando-as em *logs* para serem analisadas pela parte de detecção de eventos.

2 Trabalhos Relacionados

A segunda parte da monitoração corresponde a monitorar os recursos distribuídos do sistema, como memória real e virtual, uso do processamento, memória secundária e conexões de rede. Já a terceira parte é responsável por localizar dispositivos móveis, podendo ser, por exemplo, a localização geográfica do dispositivo ou a célula em que ele se encontra. A quarta parte da monitoração corresponde a monitorar serviços e recursos que estarão freqüentemente sendo incluídos, removidos, reconfigurados ou movidos de lugar.

Já a **detecção de eventos** possui um módulo para analisar os dados coletados pela monitoração e, a partir deles, determinar a necessidade ou não de se iniciar uma reconfiguração dinâmica do sistema. A análise é feita através dos *logs* gerados pelos interceptadores de interações de objetos e dos dados enviados pelo monitor de recursos.

Um módulo do pacote de detecção de eventos define expressões de eventos complexos a partir da combinação de outros eventos, criando uma composição de eventos. Além disso, este módulo também notifica a ocorrência de eventos para os objetos que tenham registrado interesse no evento ocorrido. Neste caso é enviada uma notificação ao pacote responsável pela reconfiguração dinâmica desses objetos.

A **reconfiguração dinâmica** foi dividida em três partes. A primeira é a *Tomada de Decisão de Reconfiguração*, que recebe a ocorrência de evento e determina quais ações de reconfiguração serão necessárias para reconfigurar o sistema.

A segunda parte da reconfiguração dinâmica é o *Reconfigurador Dinâmico*, responsável por aplicar as ações decididas na primeira parte. Para isso conta com o auxílio da terceira parte, o *Gerente de Dependências*, que controla as dependências existentes entre os componentes da aplicação. O gerenciamento dessas dependências permite que a reconfiguração seja feita de forma consistente e segura.

O *Gerente de Dependências* utiliza os *Configuradores de Componentes* descritos na Seção 4.2 desta dissertação, sendo possível a representação de pré-requisitos para carregar e ativar um componente no sistema, além das dependências dinâmicas existentes entre os componentes.

2.10 JBoss e *Java Management Extensions* (JMX)

Marc Fleury e Francisco Reverbel [Fleury and Reverbel, 2003] apresentaram a arquitetura principal do JBoss [JBoss, [sítio](#)] e seu modelo de componentes baseado na tecnologia de *Java Management Extensions* (JMX) [JMX, [sítio](#)].

O JBoss é um servidor de aplicação compatível com a plataforma *Java 2 Enterprise Edition* (J2EE) [J2EE, [sítio](#)]. Um servidor de aplicação é um *middleware* que oferece um ambiente de

execução para aplicações baseadas em componentes. Em novembro de 2000, o JBoss foi redesenhado e reescrito para ser uma implementação completa do J2EE baseada em JMX.

O JMX define uma arquitetura para o gerenciamento dinâmico e monitoração de recursos distribuídos pela rede. Com o JMX, um determinado recurso é associado a um ou mais componentes chamados de *Managed Beans* (MBeans) que são registrados em um servidor gerenciador de objetos (MBean *Server*).

Um agente JMX consiste de um MBean *Server* com MBeans registrados e um conjunto de serviços padrões, como carregamento dinâmico de classes, monitoração, *timer* e relação entre MBeans. Esses serviços são geralmente implementados como MBeans, permitindo que os agentes JMX controlem recursos e os disponibilizem para o gerenciamento de aplicações remotas.

O objetivo do JMX é prover um padrão para o gerenciamento e monitoração de toda variedade de *software* e componentes de *hardware* em Java. A arquitetura JMX consiste de três níveis principais:

- O nível de **instrumentação** define como associar recursos de maneira que eles possam ser monitorados e manipulados (remotamente ou não) por aplicações de gerenciamento;
- O nível de **agente** define os agentes controladores do conjunto de recursos associados;
- O nível de **serviços distribuídos** define como as aplicações de gerenciamento interagem com os agentes e seus recursos associados.

Existe um nível de indireção que separa os MBeans de seus clientes. Essa indireção é feita pelo MBean *Server*, que descobre em tempo de execução qual MBean está associado ao nome passado pelo cliente. Isso favorece a reconfiguração dinâmica do sistema, já que a ausência de referências diretas a um MBean facilita a sua substituição e, como o cliente só conhece a interface Java do MBean, é possível alterar tanto a sua implementação quanto a sua interface de gerenciamento.

A interface de gerenciamento de um MBean consiste em seus atributos, operações e notificações que são emitidas pelo MBean. Um MBean pode ser do tipo **estático**, implementando essa interface de gerenciamento estaticamente deixando-a visível ao agente MBean, ou pode ser do tipo **dinâmico**, definindo sua interface de gerenciamento em tempo de execução através de meta-dados de configuração.

O JBoss possui também um gerenciamento de dependência entre os MBeans baseado no padrão de modelagem chamado Configurador de Componente descrito em [Schmidt et al., 2000] que difere do modelo de Configuradores adotado pelo nosso arcabouço descrito no Capítulo 4. Um *ServiceController* é responsável pelo repositório de componentes e rastreia todas as dependências

2 Trabalhos Relacionados

entre os MBeans instalados. Quando um MBean é criado, o *Service Controller* força a criação de todos os MBeans dependentes, assegurando que todos os serviços dos quais um MBean depende já estejam criados. O mesmo ocorre para a destruição de um MBean, destruindo os serviços dos quais um MBean depende antes de sua própria destruição.

O JMX inclui uma interface aberta de forma que um agente JMX e seus recursos possam apresentar informações de gerenciamento consistentes com vários outros modelos de gerenciamento padrão, como:

- *Simple Network Management Protocol* (SNMP) [[SNMP, sítio](#)];
- *Common Information Model and Web Based Enterprise Management* (CIM/WBEM) [[CIM/WBEM, sítio](#)];
- *Lightweight Directory Access Protocol* (LDAP) [[LDAP, sítio](#)].

A especificação de JMX inclui a definição de diversos protocolos de gerenciamento. Isso permite o acesso e comunicação com agentes em outros sistemas através de um protocolo já pré-existente.

2.11 Quadro Comparativo

Resumimos em uma tabela comparativa as características de cada trabalho relacionado ao nosso modelo. A Tabela 2.1 mostra a lista dessas características encontradas em cada trabalho relacionando-as com o nosso modelo.

A transferência de estado é uma característica muito importante na Reconfiguração Dinâmica e é explicada com mais detalhes na Seção 5.2. O Modelo Formal caracteriza o trabalho do autor em demonstrar formalmente que a Reconfiguração Dinâmica funciona.

O Gerenciador de Configuração é um componente central que é acionado para coordenar a reconfiguração da aplicação. A Auto-adaptação caracteriza a capacidade do próprio sistema se monitorar e se adaptar às mudanças que ocorrem em tempo de execução. O uso de metadados mostra a existência de pré-requisitos ou dependências para auxiliar a reconfiguração dinâmica do sistema e a implementação indica se o trabalho foi validado com uma implementação prática do modelo.

A Tabela 2.1 mostra as características que cada trabalho relacionado ao nosso modelo possui. A transferência de estado é encontrada nos trabalhos das Seções 2.3, 2.5, 2.6, 2.7 e 2.9. Nosso trabalho também possui essa característica, descrita na Seção 5.2.

	Transfe- rência de Estado	Modelo Formal	Gerenciador de Confi- guração	Auto- Adaptação	Uso de meta- dados	Imple- mentação
2.1 - Argus		X				X
2.2 - Conic			X			X
2.3 - POLYLITH	X				X	X
2.4 - Bidan			X			X
2.5 - Tang	X	X			X	
2.6 - Almeida	X		X			X
2.7 - $2k$	X		X		X	X
2.8 - LuaSpace					X	X
2.9 - Silva	X		X	X	X	X
2.10 - JBoss					X	X
Nosso Arcabouço	X		X		X	X

Tabela 2.1: Tabela Comparativa

Os trabalhos descritos nas Seções 2.1 e 2.5 formalizaram as características e dificuldades encontradas durante uma reconfiguração, mas este último não possui uma implementação validando a sua proposta, ao contrário do nosso arcabouço, que apesar de não possuir um modelo teórico e formal, estendemos e implementamos um modelo para reconfiguração, descritos nos Capítulos 4 e 5.

A presença de um gerenciador central durante a reconfiguração é encontrada nas Seções 2.2, 2.4 e 2.6. Em particular, os trabalhos das Seções 2.7, 2.9 e o nosso próprio arcabouço possui um componente que não é central mas coordena a reconfiguração chamado de Configurador de Componente, descrito no Capítulo 4.

Apenas o trabalho da Seção 2.9 dão suporte à auto-adaptação. Como essa característica não é o foco de nosso trabalho, os outros trabalhos pesquisados não possuem auto-adaptação.

O uso de metadados é encontrado em 2.3, 2.5, 2.7, 2.8 e 2.9. Nosso trabalho se apóia nas informações das interdependências entre os componentes geradas pelos Configuradores de Componentes durante a reconfiguração para auxiliar todo o processo.

Por último, apenas o trabalho da Seção 2.5 não possui uma implementação de seu modelo proposto. Nosso arcabouço estendeu o modelo de Configurador de Componentes (descrito nos Capítulos 4 e 5) e implementamos uma aplicação gráfica em *Java Swing* descrita na Seção 5.5.

2 Trabalhos Relacionados

Capítulo 3

Reconfiguração Dinâmica de Componentes

Este capítulo descreve as técnicas e mecanismos existentes para a reconfiguração dinâmica de componentes. As etapas para a reconfiguração são mostradas na Seção 3.1, os requisitos necessários para se executar a reconfiguração são listados na Seção 3.2 e os mecanismos existentes são descritos na Seção 3.3.

O objetivo da reconfiguração dinâmica é permitir que um sistema evolua durante seu tempo de execução, trazendo pouco ou nenhum impacto negativo ao desempenho do sistema [Almeida et al., 2001b]. Entretanto, construir uma aplicação que permita uma reconfiguração dinâmica de forma não intrusiva no projeto do sistema não é trivial.

Durante a reconfiguração, certas partes do sistema são afetadas, enquanto que outras continuam sua execução normalmente. Essas partes afetadas são reconfiguradas (por exemplo, substituídas por novas implementações) sem que o sistema precise ser totalmente parado, introduzindo-se apenas a menor interrupção possível de seus serviços, mas garantindo que se preserve a consistência do sistema após a reconfiguração e contribuindo para que haja um aumento na disponibilidade do sistema.

3.1 Etapas da Reconfiguração Dinâmica

Um dos aspectos mais difíceis na reconfiguração dinâmica de componentes é que uma aplicação possui estado, tanto os seus componentes como as interações entre eles. Essa informação do estado deve ser transferida do antigo para o novo componente (no caso de substituição).

3 Reconfiguração Dinâmica de Componentes

Assim, além de fornecer mecanismos para reconfigurar os componentes durante sua execução, o sistema deve também ser capaz de divulgar e até instalar informação de estado dos componentes.

A reconfiguração dinâmica de sistemas baseados em componentes engloba operações como *Adição*, *Remoção*, *Migração* ou *Substituição* dos componentes [Almeida et al., 2001b]:

- **Adição e Remoção.** Simplesmente engloba a entrada de um novo componente ou a remoção de outro do sistema. A remoção é mais trabalhosa, pois geralmente o componente a ser removido possui outros que dependem dos serviços oferecidos por ele. A simples remoção pode até causar uma falha fatal no sistema;
- **Substituição.** Significa substituir um componente por outro, onde o novo componente poderá executar em um outro ambiente e possuir novas funcionalidades e requisitos que diferem do componente antigo. Entretanto, os componentes que mantinham referência ao antigo devem referenciar o novo, e esta nova versão deverá manter ou incluir novas funcionalidades à interface original do componente;
- **Migração.** Significa mover um componente para outro local de execução. Possui os mesmos problemas existentes na Adição e Remoção, visto que estaremos removendo o componente de um ambiente e adicionando-o em outro, mais os problemas da substituição, pois deveremos atualizar as referências dos clientes para o componente no novo local.

A Figura 3.1 ilustra passo a passo um exemplo de substituição de componente. No quadro 1 temos o cenário inicial, com o cliente invocando operações no componente. No quadro 2 surge a necessidade de substituir o componente atual por um novo. O processo de transferência de estado é mostrado no quadro 3 e no final o sistema está reconfigurado com o cliente invocando as operações no novo componente.

3.2 Requisitos para Reconfiguração Dinâmica

Os próximos requisitos são desejáveis idealmente para que ocorra uma reconfiguração dinâmica no sistema [Almeida et al., 2001b]:

- **Preservação da Consistência.** A parte do sistema que interage com a parte que foi reconfigurada deverá continuar funcionando corretamente, ou seja, as entidades do sistema deverão estar em um estado mutuamente consistente após a reconfiguração. Para que isso ocorra, o processo de reconfiguração deverá se iniciar somente após o sistema entrar em um estado seguro para reconfiguração;

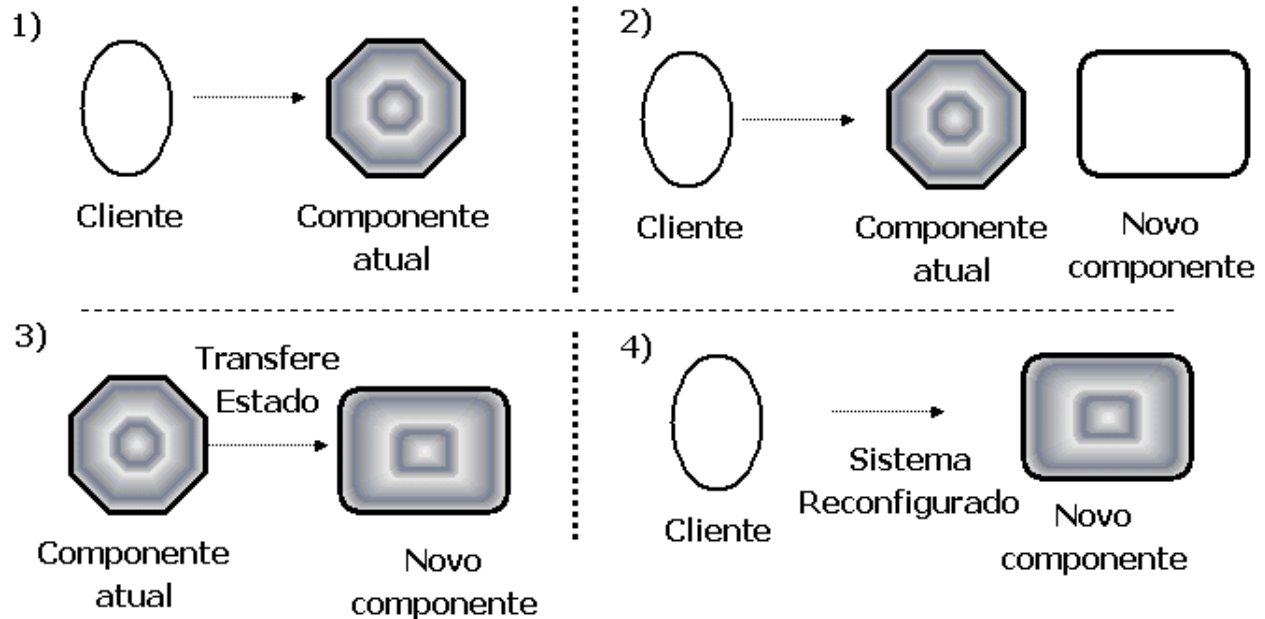


Figura 3.1: Exemplo de substituição de componente com estado.

- **Correção.** A reconfiguração dinâmica do sistema deve ser executada resultando em uma correta evolução do sistema, ou seja, componentes afetados ou não pela reconfiguração deverão continuar funcionando corretamente;
- **Mínimo impacto na execução.** Durante a reconfiguração partes do sistema que não são afetadas deverão continuar disponíveis para execução, causando uma mínima parada nos serviços oferecidos pelo sistema;
- **Máxima transparência.** A reconfiguração dinâmica deverá ser transparente para o desenvolvedor dos componentes do sistema, sendo a menos intrusiva possível.

Além disso, os componentes que interagem no sistema possuem estado que deve ser transferido durante o processo de reconfiguração, mantendo a compatibilidade entre a antiga e a nova configuração do sistema. É necessário que exista uma interação entre as partes afetadas para que esse estado não seja perdido após a reconfiguração. Por esse motivo a reconfiguração dinâmica requer que os componentes possuam um mecanismo de **exportar e importar o seu estado** em tempo de execução.

Essa importação e exportação é também chamada de *Transferência de Estado* do componente [Bidan et al., 1998]. No caso de componentes sem estado (*stateless*) essa transferência não existe,

3 Reconfiguração Dinâmica de Componentes

mas, em outros casos, ela ocorrerá sempre que houver uma substituição de um componente por outro, seja para inserir uma nova implementação de algoritmos, ou para atender novas necessidades que surjam durante o tempo de execução do componente.

Para que a *Migração de Estado* ocorra com sucesso, é necessário que os componentes estejam em um **estado seguro**, preservando assim a consistência do sistema antes e depois de sua reconfiguração. Dizemos que um componente atinge o estado seguro quando:

- não possui nenhuma invocação de métodos por parte de seus clientes que possam alterar o seu estado;
- o componente não está processando nenhum método que possa alterar o seu estado.

Em outras palavras, o componente deve estar em um modo passivo de maneira que não está processando e nem irá processar nenhum método que venha a modificar o seu estado. Atingindo esse estado seguro, podemos tranquilamente substituir o componente por um novo e transferir o seu estado garantindo a correta evolução do sistema.

A Figura 3.2 mostra um componente atingindo o estado seguro. O componente bloqueia as chamadas a métodos de seus clientes, conforme mostrado no quadro 2.1, e aguarda a finalização de qualquer processamento que esteja executando naquele instante. Não existindo mais nenhum processamento, o componente atinge o estado seguro e pode ser reconfigurado, conforme mostra o quadro 3 da figura.

O maior desafio é como fazer para que os componentes atinjam esse estado seguro e permaneçam nesse estado durante o processo de reconfiguração dinâmica. Os mecanismos propostos para resolver esse problema são descritos na Seção 3.3.

A preservação da consistência do sistema é um dos requisitos mais importantes na reconfiguração dinâmica e não pode ser ignorada. O sistema reconfigurado dinamicamente deve estar em um estado correto ao final da reconfiguração. Os aspectos de estado correto e preservação da consistência foram identificados em [Moazami-Goudarzi, 1999]. Um sistema está em um estado correto se:

1. A integridade estrutural do sistema é preservada. Ou seja, após a reconfiguração a estrutura do sistema é afetada com clientes deixando de acessar os componentes antigos e realizando requisições aos novos componentes instalados. Essa nova versão instalada deve satisfazer os requisitos da versão original e seus clientes devem continuar acessando seus serviços preservando-se assim a integridade estrutural do sistema;

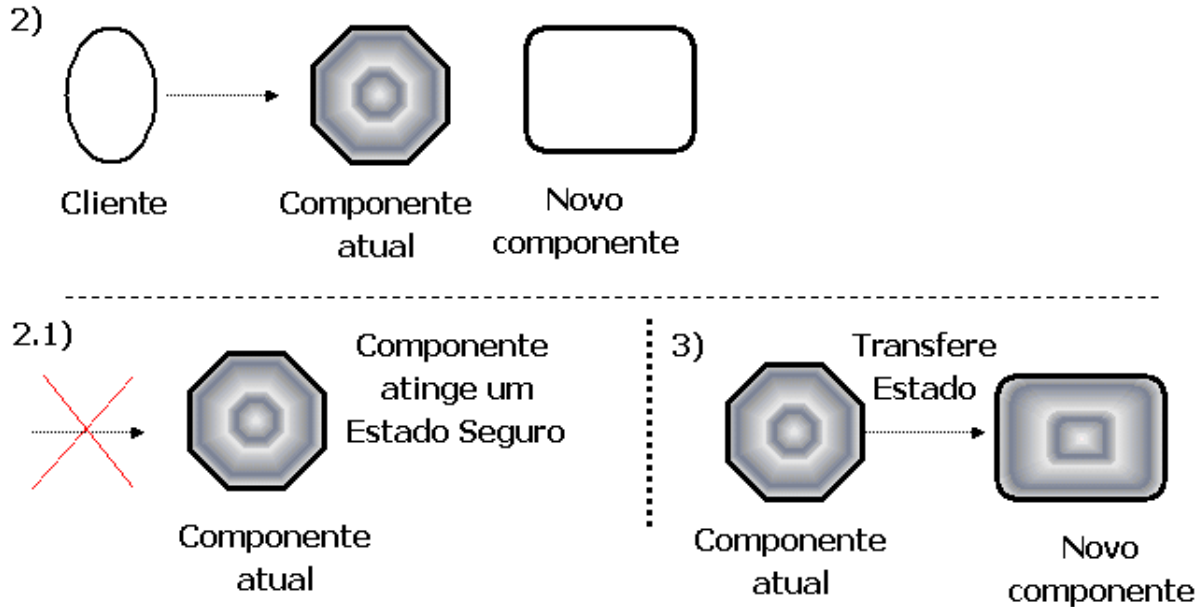


Figura 3.2: Componente atingindo estado seguro.

2. As partes afetadas do sistema estão em um estado mutuamente consistente, ou seja, se elas conseguem e continuam interagindo entre si com sucesso. As partes afetadas do sistema são aquelas que passaram pelo processo de reconfiguração, e é durante essas interações que o estado de cada uma é afetado;
3. As invariantes do estado da aplicação são preservadas. Por exemplo, se existisse uma invariante do estado da forma “toda gravação da hora do sistema em arquivo deve ser única”, o sistema terá a sua hora gravada em arquivo a cada t minutos. Para preservar essa invariante após a reconfiguração, a nova versão instalada deve ser inicializada em um estado que evite a gravação de alguma hora já gravada pela aplicação.

3.3 Mecanismos de Reconfiguração

A reconfiguração dinâmica de um sistema deve ser realizada sem que haja a necessidade de uma parada total de seus serviços. Esse requisito é vital em sistemas cuja disponibilidade deva ser alta e longas paradas de seus serviços são inaceitáveis devido a razões financeiras ou de segurança, como é o caso de aplicações críticas.

Devido a esses motivos, muitos mecanismos de reconfiguração dinâmica de sistemas foram

3 Reconfiguração Dinâmica de Componentes

propostos para que seja possível evoluir esses sistemas durante a sua execução (por exemplo substituindo componentes por outros de melhor algoritmo), introduzindo o mínimo possível de interrupção de seus serviços. Esses mecanismos foram desenhados de modo a garantir que os requisitos de reconfiguração dinâmica descritos na Seção 3.2 sejam cumpridos.

3.3.1 Mecanismo de Bloqueio de Chamadas

Para que ocorra a reconfiguração dinâmica é necessário que as partes envolvidas cheguem a um estado seguro e permaneçam nele durante a reconfiguração. Isso só será possível se as chamadas a métodos das partes envolvidas fiquem bloqueadas de modo que não se alterem os seus estados garantindo assim a consistência do sistema após a reconfiguração. Os sistemas propostos em [Kramer and Magee, 1985, Bidan et al., 1998, Almeida et al., 2001b, Almeida et al., 2001a] seguem essa abordagem. Esse bloqueio de chamadas a métodos pode ser feito nos seguintes locais:

- Na aplicação do lado **cliente**. Esse mecanismo é ilustrado na Figura 3.3. Quando o bloqueio é feito pelo cliente, a reconfiguração é facilitada pois basta o cliente aguardar a sua finalização e continuar as chamadas já no novo componente. Mas essa é uma abordagem mais intrusiva pois é necessário adaptar o código do cliente (além do código do componente reconfigurado) para realizar a reconfiguração do sistema. Um caso típico seria o cliente invocar um método no componente no momento da reconfiguração, recebendo uma exceção abortando a chamada (pois uma reconfiguração está em andamento) e então ele se bloquearia para poder invocar novamente a chamada ao método mais tarde. A vantagem é que esse tipo de bloqueio é mais simples em relação aos outros (explicados nos próximos itens), mas a desvantagem é ser necessário uma alteração no código do cliente, que em muitos casos isso não é possível;
- Bloquear as chamadas no **componente** sendo reconfigurado tem a vantagem de ser transparente para o cliente, mas possui a desvantagem de ser necessário migrar para o novo componente não só o estado atual do componente sendo reconfigurado como também o estado local do método chamado (suas variáveis locais, ou seja, aquelas criadas e utilizadas pelo método chamado) para que o novo componente dê continuidade à chamada. A Figura 3.4 mostra esse caso. Essa abordagem insere pontos de reconfiguração no meio da implementação dos métodos, que são pontos onde as chamadas são bloqueadas e é descrita com mais detalhes na Seção 3.3.3;

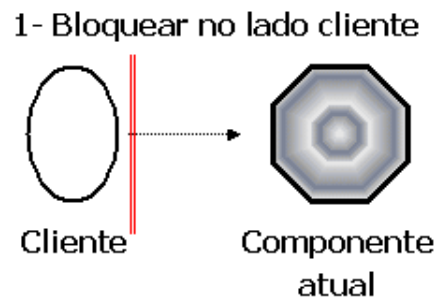


Figura 3.3: Mecanismo de bloqueio no lado do **cliente**.

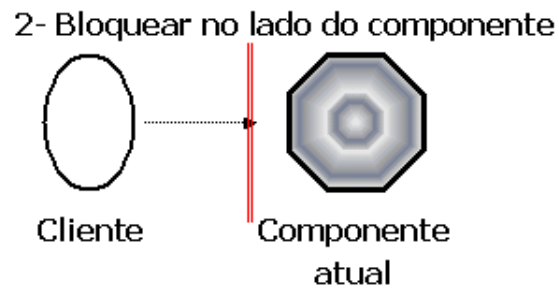


Figura 3.4: Mecanismo de bloqueio no lado do **componente**.

3- Bloquear entre o cliente e o componente

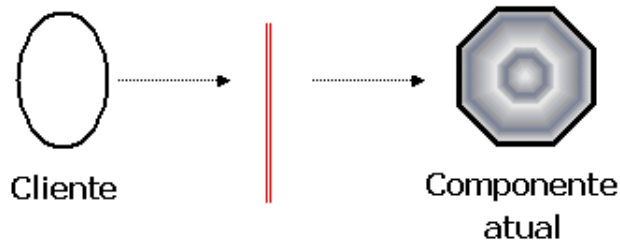


Figura 3.5: Mecanismo de bloqueio **entre o cliente e o componente**.

- Bloquear as chamadas **entre o cliente e o componente** sendo reconfigurado exige uma indireção nas chamadas por um serviço intermediário entre os dois, conforme Figura 3.5. Esse bloqueio é realizado durante a configuração até que o novo componente seja instalado e as chamadas redirecionadas para esse novo componente. Essa abordagem se apóia em mecanismos existentes no próprio *middleware* em que o sistema está executando para redirecionar as chamadas entre o cliente e componente, conforme descrito na Seção 3.3.2.

Como se percebe, o bloqueio de chamadas existe em todo mecanismo de reconfiguração dinâmica, só alterando o local em que é feito esse bloqueio, além de existir um sincronismo de comunicação entre as partes bloqueadas. Bloquear chamadas é muito importante para se garantir a consistência do sistema após a reconfiguração. No entanto se as partes envolvidas não possuem estado (*são stateless components*) é possível reconfigurar dinamicamente sem bloquear chamadas.

Uma alternativa para bloqueios de chamadas seria a criação de uma interface intermediária entre o cliente e o componente. Essa interface e as formas como os clientes se conectam (*bindings*) aos componentes têm como exemplo o sistema *Adapt* [Fitzpatrick et al., 1998]. Os autores chamam essa interface intermediária de interface externa, que atua como um *proxy* entre as chamadas do cliente e o componente. Desta forma, a interface pode ser usada para se reconfigurar ou adaptar o sistema dinamicamente. Esse mecanismo se assemelha ao de indireção de chamadas, explicado na Seção 3.3.2.

3.3.2 Mecanismo de Indireção de Chamadas

Sistemas mais recentes utilizam mecanismos do próprio *middleware* em que estão instalados (CORBA, Enterprise JavaBeans [Java/EJB, sítio], Microsoft .NET [Microsoft/.NET, sítio])

para redirecionar cada chamada do cliente para um componente, ou seja, existe um serviço ou outro componente intermediário que intercepta toda chamada entre o cliente e o componente, redirecionando-a ao seu destino. Esse indireccionamento existe para prover transparência e flexibilidade às aplicações clientes.

Sistemas como o *Lua Space* [Batista and Carvalho, 1999, Batista and Rodriguez, 2000] utilizam esse mecanismo também para poder selecionar dinamicamente, em tempo de execução, qual componente provê o serviço requisitado pelo cliente, configurando a aplicação como um conjunto de serviços sem que o cliente saiba *a priori* qual componente irá executar o serviço. A reconfiguração seria complementar ao modelo deles para se tratar aspectos como transferência de estado do antigo para o novo componente.

Durante o processo de reconfiguração dinâmica, as chamadas são bloqueadas por esse serviço intermediário, para que seja possível migrar o estado (se necessário) ao novo componente. Ao final, é feita a atualização do serviço para que toda nova chamada seja redirecionada a esse novo componente.

A vantagem desse mecanismo é a transparência para os clientes pois não há a necessidade de se alterar os seus códigos para suportar a reconfiguração dinâmica da aplicação. Mas a desvantagem de se usar esse mecanismo é o processamento a mais causado pela interceptação de toda chamada de método entre o cliente e componente.

3.3.3 Mecanismo de Pontos de Reconfiguração

O uso de pontos de reconfiguração para indicar os pontos no código que são seguros para se realizar a reconfiguração caracteriza esse mecanismo. O termo “seguro para se reconfigurar” indica pontos no qual se sabe com certeza que naquele exato instante, mesmo que a execução do método não tenha terminado, o componente não está executando nenhuma operação que possa vir a alterar o seu estado e comprometer a reconfiguração.

A idéia principal é inserir pontos de reconfiguração em partes estratégicas do código determinando os momentos ideais para evoluir a aplicação. Sempre que o componente em execução atingir um ponto de reconfiguração, irá checar se existe uma reconfiguração a ser realizada naquele instante. Em caso positivo, ele irá parar a sua execução naquele ponto e realizar a reconfiguração. Caso contrário, o componente continua sua execução normalmente.

Vários trabalhos [Purtilo, 1990, Hofmeister, 1994, Tang, 2000] adotam essa abordagem e argumentam que nem sempre é possível esperar que os componentes alcancem um estado seguro (por exemplo bloqueando novas chamadas e aguardando que aquelas em execução terminem) pois podem existir métodos de longa execução, atrasando a reconfiguração, ou métodos que só terminem

3 Reconfiguração Dinâmica de Componentes

junto com o término da aplicação.

Utilizando pontos de reconfiguração, a execução do método é interrompida para que, naquele instante, o componente verifique se é preciso iniciar uma nova reconfiguração. Em caso positivo, a execução é parada naquele ponto e o estado do componente é transferido para a nova implementação.

O diferencial desse mecanismo é a interrupção durante a execução do método para executar a reconfiguração antes mesmo que ele termine. Nos casos em que ocorre a reconfiguração, as chamadas a métodos pelo cliente iniciam em um componente e terminam em um novo componente reconfigurado sem que o cliente perceba. Por isso, além de se preservar o estado do componente é necessário preservar o estado local do método em que ocorre a reconfiguração. As variáveis locais existentes antes do ponto de reconfiguração devem ser mantidas e transferidas ao novo componente para que sejam utilizadas no processamento do método após a reconfiguração.

Esse mecanismo preserva os detalhes da execução do método em andamento de modo que consiga processar e retornar normalmente a chamada ao cliente mesmo que ocorra a reconfiguração do componente. A invocação de um método pelo cliente procede da seguinte forma:

- inicia em um componente onde a chamada é parcialmente executada;
- é feito o mapeamento da chamada para um ponto de reconfiguração equivalente localizado em um outro componente;
- o método finaliza sua computação nesse novo componente e retorna a chamada ao cliente.

Isso permite evoluir o sistema sem que haja a necessidade de se esperar que o método termine, ou forçar a sua finalização para então iniciar a reconfiguração do sistema. O uso de pontos de reconfiguração permite considerar um componente em estado seguro antes mesmo que as execuções de métodos terminem.

Em [Tang, 2000] é especificado formalmente o uso desse mecanismo, adicionando-se novas palavras-chave à linguagem Java. Por exemplo, um método que possua um ponto de reconfiguração deve indicar na sua assinatura a palavra-chave `reconfigurables` *Identificador* e na sua implementação, o ponto de reconfiguração é indicado com a palavra-chave `reconfigurable` *Identificador*.

A Listagem 3.1 mostra um exemplo de uso de pontos de reconfiguração. A linha 3 indica a existência de um ponto de reconfiguração chamado `PartialSendBuffer`, e a sua implementação se encontra entre as linhas 5 a 8.

Listing 3.1: Uso de Pontos de Reconfiguração

```
1 void sendBuffer(byte[] buffer, String host)
2     throws IOException
3     reconfigurables PartialSendBuffer {
4         ...
5         reconfigurable PartialSendBuffer {
6             // código da reconfiguração
7             ...
8         };
9         ...
10 }
```

O mapeamento do estado é feito utilizando as novas palavras-chave `encode` (exportar o estado) e `decode` (importar o estado). É utilizado também uma nova palavra-chave `that`, representando o mapeamento do estado entre `this` (componente atual) e `that` (novo componente).

A Listagem 3.2 mostra o uso dessas palavras-chave. A linha 1 declara uma classe `NetworkStackImpl` que implementa o componente `SimpleNetworkStack` usando a nova palavra-chave `fulfills`. Na linha 3 está o início do trecho do `encode`, que mostra o uso do `this` e `that` para exportar o estado. A linha 17 inicia o trecho de `decode` para importar o estado do componente.

Listing 3.2: Mapeando estado com Pontos de Reconfiguração

```
1 class NetworkStackImpl fulfills SimpleNetworkStack {
2     ...
3     encode {
4         Enumeration keys = this.privateConnections.keys();
5         that.currentConnections = new java.util.Vector();
6
7         while (keys.hasMoreElements()) {
8             String currentHost = (String) keys.nextElement();
9             String currentBuffer = (String) keys.get(currentHost);
10            CurrentPendingSends newSend = new CurrentPendingSends();
11            newSend.host = currentHost;
12            newSend.buffer = currentBuffer;
13            that.currentConnections.addElement(newSend);
```

3 Reconfiguração Dinâmica de Componentes

```
14     }
15 }
16 ...
17 decode {
18     this ();
19     Enumeration e = that.currentConnections.Elements ();
20     while (e.hasMoreElements ()) {
21         CurrentPendingSends send = (CurrentPendingSends) e.nextElement ();
22         this.privateConnections.put(send.host, send.buffer);
23     }
24 }
25 ...
26 }
```

Apesar de ter especificado formalmente cada operação utilizando pontos de reconfiguração, Tang não validou o seu trabalho com uma implementação em Java, devido à falta de um compilador e uma máquina virtual que permitisse o desenvolvimento de aplicações baseadas no seu modelo.

Uma desvantagem do uso desse mecanismo é a ocorrência de casos nos quais a nova implementação é muito diferente da original, tornando impossível o mapeamento de contexto e a continuação da execução do método no novo componente. Por isso é desejável que exista sempre um ponto de reconfiguração no início ou fim de cada método, onde todas as implementações compartilham o mesmo comportamento.

Outra desvantagem é como lidar com aplicações multi-processadas, com vários processos acessando o componente a ser reconfigurado. Tang [Tang, 2000] propõe que, nesses casos, só seja feita a reconfiguração se cada processo estiver passivo em relação ao componente, ou seja, bloqueado em algum ponto de reconfiguração. Feita a reconfiguração, todos os processos são ativados e continuam a execução normalmente.

Capítulo 4

Configuradores de Componentes

Configuradores de Componentes são objetos responsáveis por manter uma representação explícita das dependências dinâmicas entre os componentes. Neste capítulo será apresentada a definição de componentes na Seção 4.1 e o modelo dos Configuradores de Componentes é mostrado na Seção 4.2.

4.1 O que são Componentes

No contexto desta dissertação, componentes são objetos que encapsulam funcionalidades e interagem com outros componentes através de interfaces bem-definidas [Grannon, 1997]. Szyperky [Szyperki, 2002] define componentes de software como unidades independentes que interagem para formar um sistema. São definidos por interfaces públicas que especificam suas operações, assim como protocolos que utilizam para se comunicarem com outros componentes.

Sistemas implementados neste modelo são chamados de sistemas baseados em componentes. Uma definição precisa de componente depende do ambiente em que ele é utilizado, pois esse ambiente é quem definirá toda a arquitetura e funcionalidades que os componentes deverão prover.

O modo como as interfaces dos componentes serão apresentadas (visualmente ou não) é definido pela arquitetura dos componentes. Exemplos de arquiteturas de componentes são CCM (*CORBA Component Model*) [OMG, 2002, CORBA/CCM, sítio], Microsoft .NET [Microsoft, 2000] e *Enterprise JavaBeans* [Sun Microsystems, 2003]. Existem dois modelos muito comuns de integração de componentes [Grannon, 1997]:

- **Comunicação Cliente/Servidor.** O cliente da aplicação executa requisições através das funções públicas definidas pelas interfaces dos componentes. Microsoft .NET

4 Configuradores de Componentes

[[Microsoft, sítio](#)], *CORBA Component Model* e *Enterprise JavaBeans* são exemplos baseados neste modelo, mas CORBA é flexível o bastante para permitir a utilização de outros modelos;

- **“Circuito Integrado de Software”**. Analogia a um circuito eletrônico que possui *buffers* de entrada e portas de saída de dados. Assim, é possível conectar qualquer porta de saída a uma entrada de dados correta, formando um fluxo de mensagens. O tipo de uma porta seria a interface do componente descrevendo as mensagens que ele pode receber. Além desse tipo de comunicação, os componentes também podem se comunicar através de eventos ou exceções geradas por outro componente para todos os outros que estejam “ouvindo” eventos daquele tipo. *Enterprise JavaBeans* [[Java/EJB, sítio](#)] utiliza JMS (*Java Message Service*) com *Message-Driven Beans* e o *CORBA Component Model*, com os conceitos de faceta e receptáculos, oferecem suporte para esse modelo.

Uma arquitetura de componentes possui também o conceito de contêiner. Um contêiner é um sistema que contém os componentes e é utilizado para criar um ambiente de execução para os componentes e conectá-los entre si, provendo funcionalidades como suporte a transações, persistência e segurança, como é o caso do contêiner de *Enterprise JavaBeans*.

4.2 Modelo dos Configuradores de Componentes

O modelo de Configuradores de Componentes [[Kon and Campbell, 1999](#), [Kon, 2000](#)] foi criado com o objetivo de possibilitar a representação das dependências dinâmicas entre os componentes em execução. Um Configurador de Componentes é um objeto associado a um componente e é responsável pela reificação das dependências dinâmicas daquele componente. Reificação é a transformação de informações sobre a execução de um programa em dados disponíveis ao próprio programa. A Figura 4.1 ilustra essa associação entre o componente e o seu Configurador de Componente.

As partes do sistema a serem reconfiguradas dinamicamente deverão possuir um Configurador de Componente associado. O modelo de Configuradores de Componentes associa um Configurador a cada componente sendo executado. Existindo uma dependência de um componente *A* em relação a um componente *B*, esta dependência será representada através de uma referência para o Configurador de Componente de *A* armazenada no Configurador de Componente de *B* e vice-versa, conforme é mostrado na Figura 4.2.

Configuradores de Componentes também são responsáveis por distribuir eventos de reconfiguração entre os componentes interdependentes. Exemplos de eventos podem ser a falha ou

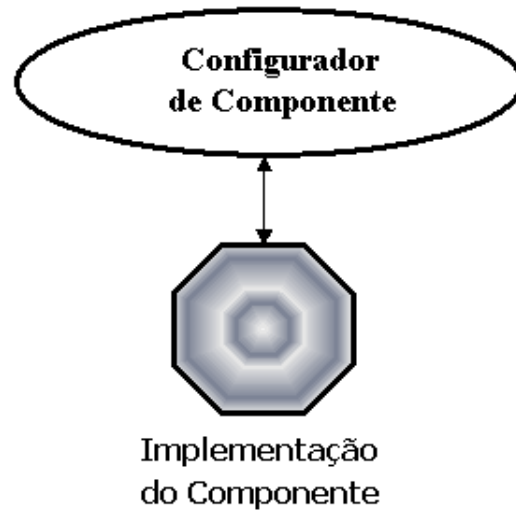


Figura 4.1: Configurador de Componente associado ao seu Componente

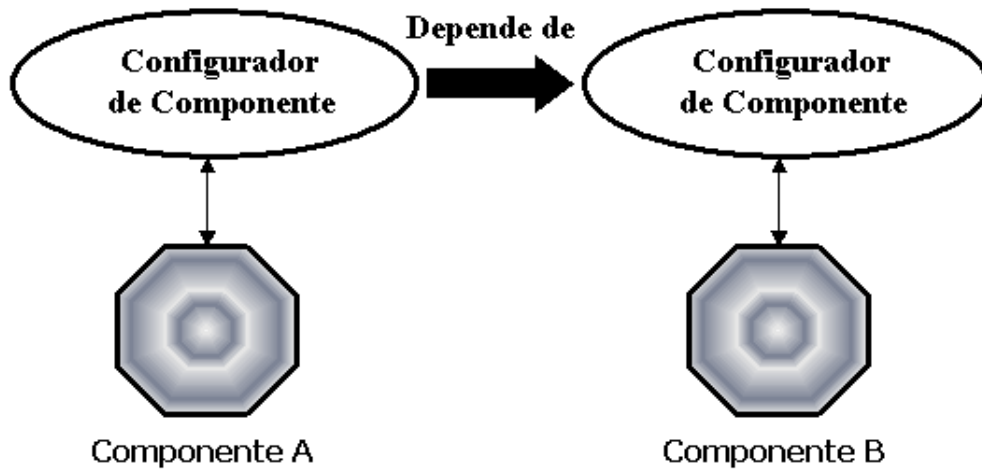


Figura 4.2: Dependência entre Configuradores de Componentes

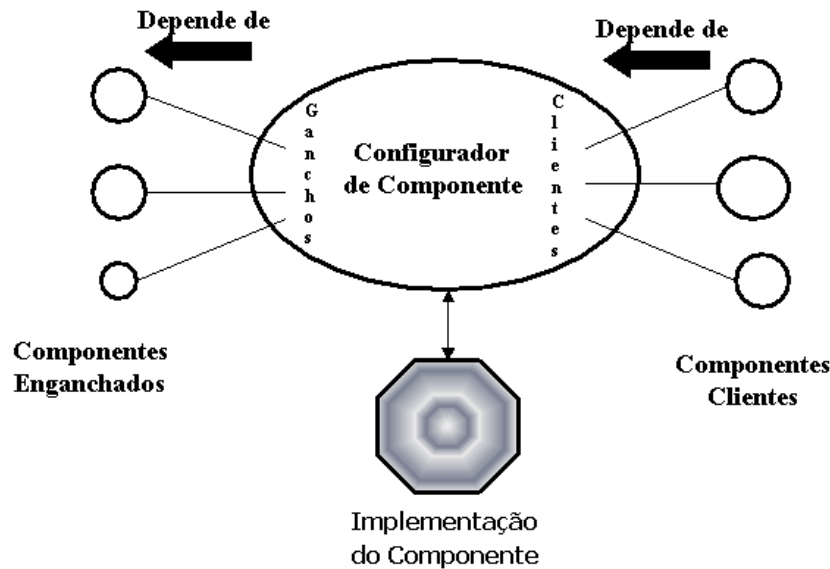


Figura 4.3: Reificação das Dependências da Componente

destruição de um cliente, reconfigurações do sistema ou uma substituição de um componente.

Deste modo aspectos não-funcionais do sistema como tolerância a falhas e reconfiguração dinâmica podem ser implementados pelos Configuradores de Componentes, deixando que as funções principais sejam implementadas pelos componentes do sistema, existindo uma clara separação entre aspectos não-funcionais e funcionais do sistema.

Com as informações das interdependências dinâmicas entre os componentes, forma-se um grafo dirigido distribuído do sistema que poderá ser referenciado em tempo de execução e manipulado pela aplicação.

É possível com esse grafo das interdependências criar sistemas reflexivos que possam utilizar-se dessas informações otimizando seu desempenho ou adaptando-se às mudanças do ambiente.

Uma forma de otimizar o desempenho seria selecionar componentes diferentes de acordo com as necessidades existentes manipulando as dependências entre componentes para adaptar-se às mudanças em tempo de execução.

Um Configurador de Componentes gerencia a reconfiguração dinâmica de cada componente, sendo responsável por armazenar as dependências dinâmicas entre os componentes da aplicação [Kon and Campbell, 2000]. A Figura 4.3 demonstra as dependências que um Configurador de Componente reifica, utilizando um conjunto de ganchos (*hooks*) e clientes para representar as interdependências.

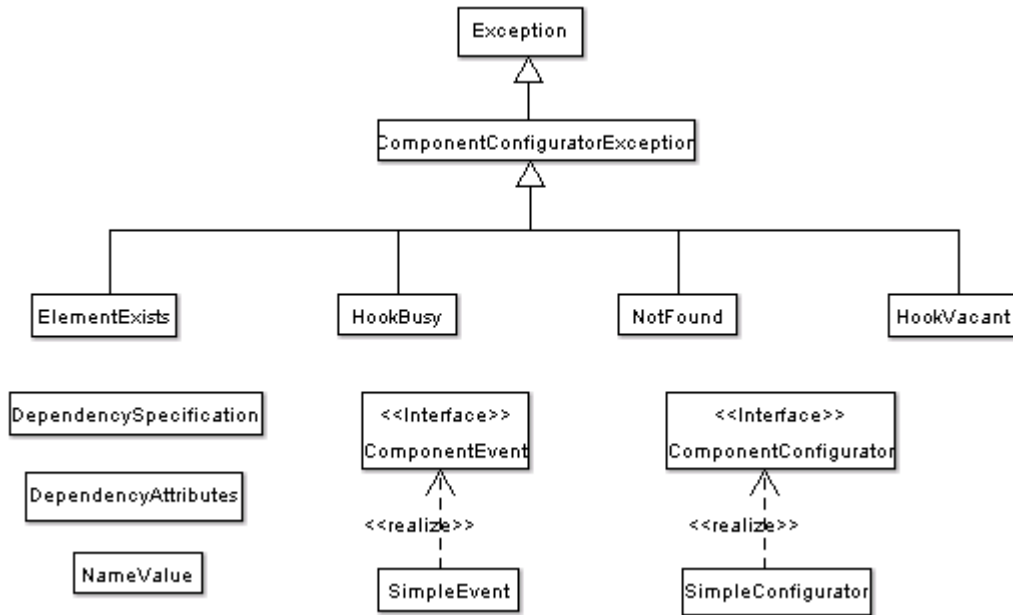


Figura 4.4: Diagrama de Classes do Modelo de Configuradores de Componentes

Cada componente C possui seu próprio Configurador de Componente. Este por sua vez possui um conjunto de ganchos utilizados para se engancharem a outros Configuradores de Componentes. Esses Configuradores de Componentes enganchados (*hooked components*) são os componentes dos quais C depende. Os componentes que dependem de C são chamados de clientes.

Quando um componente C_1 depende de outro componente C_2 o sistema executa duas ações:

1. Anexar o Configurador de Componentes de C_2 a um dos ganchos do Configurador de Componentes de C_1 .
2. Adicionar o Configurador de Componentes de C_1 à lista de clientes do Configurador de Componentes de C_2 .

A Figura 4.4 mostra o diagrama de classes do Modelo de Configuradores de Componentes. Nesse diagrama a interface `ComponentConfigurator` e sua classe de implementação de referência `SimpleConfigurator` foram estendidas em nosso arcabouço para tratar das interdependências entre os componentes, conforme descrito no Capítulo 5. A seguir descrevemos brevemente o papel de cada classe:

- *ComponentConfiguratorException*: é a classe pai de todas as exceções desse modelo;

4 Configuradores de Componentes

- *ElementExists*: exceção lançada quando adiciona-se ao Configurador de Componente um gancho ou cliente já existente;
- *HookVacant*: exceção lançada quando foi pedido ao componente para retirar um gancho mas este não existe;
- *HookBusy*: exceção lançada indicando que o gancho já possui um Configurador de Componente associado a esse gancho;
- *NotFound*: exceção lançada quando o gancho ou cliente não é encontrado;
- *DependencySpecification*: classe que encapsula uma tripla de valores: o nome do gancho, um ponteiro ao Configurador de Componente e os atributos de um gancho;
- *DependencyAttributes*: representa os atributos de uma dependência;
- *NameValue*: encapsula o nome e valor de um atributo de dependência;
- *ComponentEvent*: interface que representa os eventos de reconfiguração dinâmica (finalização, reconfiguração, substituição de componente etc);
- *SimpleEvent*: implementa os eventos de reconfiguração;
- *ComponentConfigurator*: interface que representa as interdependências entre os componentes e as suas operações;
- *SimpleConfigurator*: implementa os métodos de ComponentConfigurator.

O modelo utilizado por esta proposta de dissertação foi baseado na implementação em Java dos Configuradores de Componentes [[ComponentConfigurator, sítio](#)]. O código na listagem 4.1 mostra os métodos da interface de um Configurador de Componentes.

Listing 4.1: Interface Java do Configurador de Componentes

```
1 package configuration ;
2
3 import java.util.Vector ;
4
5 public interface ComponentConfigurator {
6
7     public void destroyComponentConfigurator ( ) ;
```

```
8
9  public void addHook(String hookName, DependencyAttributes attributes)
    throws ElementExists;
10
11 public void deleteHook(String hookName) throws NotFound;
12
13 public void hook(String hookName, ComponentConfigurator cc,
    DependencyAttributes attributes) throws HookBusy, NotFound;
14
15 public void hook(String hookName, ComponentConfigurator cc) throws
    HookBusy, NotFound;
16
17 public void unhook(String hookName) throws HookVacant, NotFound;
18
19 public void registerClient(ComponentConfigurator client, String
    hookNameInClient, DependencyAttributes attributes) throws
    ElementExists;
20
21 public void unregisterClient(ComponentConfigurator client, String
    hookNameInClient) throws NotFound;
22
23 public void eventFromHookedComponent(ComponentConfigurator
    hookedComponent, ComponentEvent e);
24
25 public void eventFromClient(ComponentConfigurator client,
    ComponentEvent e);
26
27 public Vector listHooks();
28 public int numberOfClients();
29 public Vector listClients();
30 public void name(String s);
31 public String name();
32 public void info(String s);
33 public String info();
```

4 Configuradores de Componentes

```
34 public Object implementation();
35 public void implementation(Object implementation);
36 public void printHooks(java.io.PrintStream out);
37 public void printClients(java.io.PrintStream out);
38 ComponentConfigurator getHookedComponent(String hookName) throws
    NotFound;
39 }
```

Esse modelo é flexível o bastante para ser capaz de referenciar componentes executando em um único espaço de endereçamento de memória, diferentes processos ou até mesmo em máquinas diferentes de um sistema distribuído.

Os métodos *addHook()* (linha 9) e *deleteHook()* (linha 11) são utilizados para criar e remover os ganchos do Configurador de Componentes. Cada gancho é representado por uma *String* representando o seu nome.

O método *hook()* (linhas 13 e 15) é usado para especificar que este componente depende de outro, e *unhook()* (linha 17) destrói essa dependência. Já *registerClient()* (linha 19) registra um componente que depende deste Configurador de Componentes, e *unregisterClient()* (linha 21) remove esse registro. Note que os ganchos e clientes sempre referenciam outros Configuradores de Componentes.

O método *eventFromHookedComponent* (linha 23) é chamado para anunciar que um evento ocorreu no componente que está enganchado (por exemplo, se o componente foi removido). Diferentes comportamentos podem ser definidos por diferentes implementações desta interface.

A classe *DependencyAttributes* (usada como parâmetro nos métodos *addHook()* e *registerClient()* nas linhas 13 e 19) representa uma lista de atributos de dependência. Esses atributos guardam informações sobre as características de uma dependência possibilitando diferentes tipos de tratamento durante a reconfiguração. Cada atributo é um par <nome, valor> onde **nome** é um identificador e **valor** é um objeto Java.

Um exemplo de uso de *DependencyAttributes* [Kon, 2000] seria um servidor de arquivos com dois tipos de clientes: programas simples de usuários e dispositivos de *backup* em fita. Quando o servidor precisar ser desligado, é preciso esperar que os dispositivos terminem a cópia para a fita, e essa característica pode ser registrada com atributos de dependência.

Usando atributos o servidor de arquivos consegue diferenciar seus diferentes tipos de clientes para saber se deve notificá-los (dispositivos de *backup*) ou não (programas de usuários) no momento de ser desligado.

Quando ocorre um evento no cliente do Configurador de Componentes, deverá ser chamado

o método *eventFromClient()* (linha 25). Ele poderá ser usado por exemplo para disparar as reconfigurações de componentes para adaptarem-se às novas condições de seus clientes.

No Capítulo 5 mostramos como estendemos esse modelo para usá-lo em nosso arcabouço. O *ComponentConfigurator* é usado para gerenciar as dependências entre os componentes e a interface *ComponentEvent* e sua implementação *SimpleEvent* são utilizadas para representar os eventos de reconfiguração.

Dessa forma, a substituição de um componente (Seção 5.3) utiliza a classe *SimpleEvent* para enviar um evento de reconfiguração aos seus clientes através do método *eventFromHookedComponent()* do *ComponentConfigurator*.

Capítulo 5

Extensão do Modelo

Foi desenvolvido em nossa pesquisa um arcabouço para objetos reconfiguráveis a fim de gerenciar a tarefa de reconfiguração dinâmica. Estendemos o modelo original de Configuradores de Componentes explicado na Seção 4.2 para a utilização em nosso arcabouço, e aplicamos essa extensão em uma aplicação gráfica que mostra a reconfiguração dinâmica de componentes representados por figuras geométricas.

Desenvolvemos esse arcabouço em Java para ser portátil para outras plataformas e esse arcabouço será uma das contribuições para a área de reconfiguração dinâmica. Um dos objetivos de nossa pesquisa é torná-lo genérico o suficiente para que possa ser reutilizado por outros arcabouços dessa área.

Este Capítulo está dividido nas seguintes partes: a Seção 5.1 explica o modelo que construímos de *Objetos Reconfiguráveis*; as Seções 5.2 e 5.3 explicam o que ocorre durante a transferência de estado e substituição de componentes; a Seção 5.4 descreve as políticas de manipulação de dependências; finalmente na Seção 5.5 mostramos a aplicação gráfica desenvolvida.

5.1 Objetos Reconfiguráveis

Estendendo o modelo original de Configuradores de Componentes, foi criado em nossa pesquisa um arcabouço em Java a fim de implementar algumas políticas de reconfiguração dinâmica. Esse arcabouço possui objetos chamados de **Objetos Reconfiguráveis** que definem uma interface própria para aspectos da reconfiguração.

A Figura 5.1 mostra as principais interfaces do arcabouço de Reconfiguração Dinâmica, explicadas a seguir. A interface `ReconfigurableObject` define apenas três métodos principais:

5 Extensão do Modelo

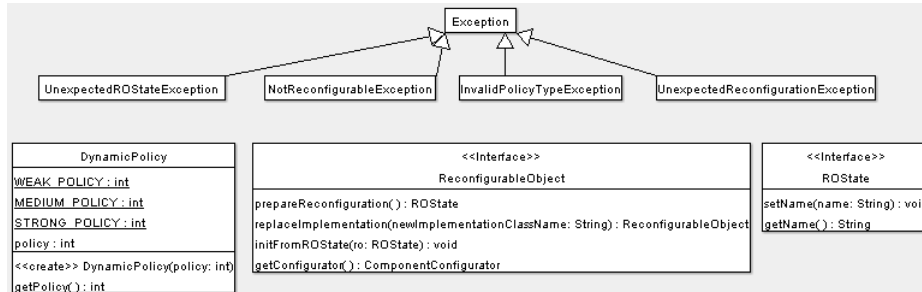


Figura 5.1: Visão do Modelo de Objetos Reconfiguráveis

1. `public ROState prepareReconfiguration()` - primeira etapa da reconfiguração, exporta o estado atual do `ReconfigurableObject`, preparando-o para ser reconfigurado,
2. `public ReconfigurableObject replaceImplementation(String newImplementationClassName)` - executa a substituição deste `ReconfigurableObject` por um novo, instância da classe descrito por `newImplementationClassName`,
3. `public void initFromROState(ROState ro) throws UnexpectedROStateException` - inicia o objeto a partir do estado contido em um `ROState`.

O método `prepareReconfiguration()` é responsável por iniciar o processo de reconfiguração, devendo copiar o estado atual do objeto e exportá-lo para uma estrutura do tipo `ROState`. É esse estado que é devolvido por `prepareReconfiguration()` e utilizado na chamada a `replaceImplementation()`.

O método `initFromROState()` é responsável por inicializar uma instância de um `ReconfigurableObject` através de um estado armazenado em um `ROState`. Desta maneira é possível realizar a transferência de estado de um `ReconfigurableObject` para outro.

A interface `ROState` define apenas um método para se obter o nome do objeto. Ela deve ser estendida para atender cada caso de exportação de estado, é uma instância do padrão *Memento* [Gamma et al., 1995].

O padrão *Memento* possibilita capturar e armazenar o estado interno de um objeto de forma que o objeto possa ser restaurado posteriormente, sem que se viole o seu encapsulamento. Em [Gamma et al., 1995] é dado o exemplo de um editor gráfico que possui a opção de desfazer a operação de um usuário. Uma operação do tipo mover um retângulo faria a aplicação armazenar o estado do objeto para que, se necessário, seja possível voltá-lo à sua posição original.

A exceção `UnexpectedROStateException` é lançada em `initFromROState()` caso o `ROState` recebido não seja do tipo esperado. Por exemplo, o `ReconfigurableObject` poderá dar suporte

para importar dados apenas de um `DummyROState` (que estende `ROState`). Caso não seja do tipo esperado, a importação do estado não poderá ocorrer, e a exceção será lançada.

A exceção `NotReconfigurableException` é lançada durante o `replaceImplementation()` caso a nova implementação a ser instanciada não seja do tipo `ReconfigurableObject`.

A exceção `UnexpectedReconfigurationException` é lançada caso ocorra algum problema durante a substituição de implementação, como por exemplo quando a nova implementação não puder ser carregada.

O método `replaceImplementation()` é responsável por substituir a instância atual pela nova implementação em `newImplementationClassName`. Ele primeiro salva o estado atual através do `prepareReconfiguration()`, cria uma nova instância de `newImplementationClassName` e transfere o `ROState` atual para a nova implementação através do método `initFromROState()`. É essa implementação que será devolvida pelo método, que será então utilizada substituindo a implementação antiga.

Neste momento, o novo `ReconfigurableObject` possui referência para o mesmo Configurador de Componentes que o antigo `ReconfigurableObject` possuía. Essa referência foi atualizada durante a exportação do estado no método `initFromROState()`. Na Seção 5.3 é mostrada a implementação do método `replaceImplementation()`.

O objeto cliente que iniciou a reconfiguração chamando `replaceImplementation()` recebe como retorno o novo `ReconfigurableObject` instanciado com o mesmo estado do `ReconfigurableObject` antigo. Portanto, o objeto cliente deve apenas substituir a antiga referência para o `ReconfigurableObject` pela nova do retorno de `replaceImplementation()`. Não existindo mais nenhuma referência para a referência antiga, ela deverá ser coletada pelo coletor de lixo do sistema.

5.2 Transferência de Estado

A transferência de estado durante a reconfiguração dinâmica ocorre quando um componente é substituído por um mais novo, sendo possível assim atender a novas necessidades que possam existir durante o tempo de execução do sistema.

Essa etapa da reconfiguração dinâmica é muito importante e também a mais delicada, pois se existir algum erro durante a transferência, a consistência e correção do sistema não será preservada, podendo resultar em até uma queda geral da aplicação.

No arcabouço criado, cada implementação específica da interface `ReconfigurableObject` é responsável por exportar e importar o estado através de um `ROState`. A vantagem é que deste

modo não se quebra o encapsulamento dos dados de cada componente. A desvantagem é o prévio conhecimento do tipo de `RState` que cada componente irá exportar/importar. Caso os tipos sejam incompatíveis, uma exceção do tipo `UnexpectedRStateException` será lançada.

5.3 Substituição de Implementação

A substituição de implementação na reconfiguração dinâmica envolve desde a divulgação do estado atual da componente até a instanciação da nova implementação e substituição do componente em execução no sistema. Além desses passos, é importante também que todas as referências à antiga implementação sejam atualizadas para o novo componente.

Conforme descrito em 5.1, o método `replaceImplementation()` é responsável pela substituição durante a reconfiguração, executando a transferência do estado, instanciando a nova implementação e devolvendo-a para que esta seja utilizada no lugar do antigo componente.

As referências para o antigo componente que foi substituído são tratadas através dos Configuradores de Componentes, já que são eles os responsáveis por manter as interdependências dinâmicas entre os componentes. Isso significa que quando um componente precisa referenciar outro, ele o fará através dos Configuradores de Componentes, atualizando-se assim o grafo dirigido das dependências. Deste modo consegue-se separar claramente a parte funcional do sistema, implementada pelos componentes, e a parte não-funcional, tratada pelos Configuradores de Componentes.

Na Listagem 5.1, apresentamos a implementação em Java do método `replaceImplementation()` mostrando a execução da substituição de implementação de um componente pelo existente na classe `newImplementationClassName`. Essa implementação faz parte do nosso arcabouço de objetos reconfiguráveis e está disponível para toda aplicação que desejar ter suporte à reconfiguração dinâmica. Um exemplo disso é a nossa aplicação gráfica, desenvolvida para demonstrar o uso do nosso arcabouço e explicada na Seção 5.5.

Listing 5.1: Implementação do método `replaceImplementation()`

```
1 public Object replaceImplementation(String newImplementaionClassName)
2     throws NotReconfigurableException ,
3         UnexpectedReconfigurationException , UnexpectedRStateException {
4     System.out.println("--> Replacing this " + getClass().getName() + "
5         with " + newImplementaionClassName);
```

```
6 // Guarda estado atual em currentState
7 ROState currentState = prepareReconfiguration();
8
9 try {
10     // Carrega nova classe
11     Class newClass = Class.forName(newImplementaionClassName);
12     System.out.println("---> Loaded new class " + newClass.getName()
13         );
14
15     Object newOb = newClass.newInstance();
16     if (newOb instanceof ReconfigurableObject) {
17         ReconfigurableObject ro = (ReconfigurableObject) newOb;
18
19         // Importa o currentState no novo objeto reconfigurável
20         // (pode lançar UnexpectedROStateException)
21         System.out.println("---> Loading new State");
22         ro.initFromROState(currentState);
23
24         System.out.println("**** Implementation replaced ****\n");
25         return ro;
26     } else {
27         throw new NotReconfigurableException("Object from class " +
28             newImplementaionClassName + " is not a ReconfigurableObject
29             !");
30     }
31 } catch (ClassNotFoundException cnfe) {
32     cnfe.printStackTrace();
33     throw new UnexpectedReconfigurationException(cnfe.getMessage());
34 } catch (InstantiationException ie) {
35     ie.printStackTrace();
36     throw new UnexpectedReconfigurationException(ie.getMessage());
37 } catch (IllegalAccessException iae) {
38     iae.printStackTrace();
39     throw new UnexpectedReconfigurationException(iae.getMessage());
40 }
```

5 Extensão do Modelo

```
37     }  
38 }
```

A linha 7 exporta o estado atual do componente e armazena esse estado em uma variável `currentState` do tipo `RState`. Esse estado será importado pelo novo componente na linha 21 ao chamar o método `initFromRState(currentState)`.

A nova implementação é carregada nas linhas 11 a 14. Caso essa nova instância carregada não seja um `ReconfigurableObject` (linha 15) será lançada a exceção `NotReconfigurableException`. Já a exceção `UnexpectedRStateException` poderá ser lançada na linha 21 caso o estado a ser importado não implemente a interface `RState`.

A exceção `ClassNotFoundException` poderá ser lançada na linha 11 caso a implementação não seja encontrada. As exceções `InstantiationException` e `IllegalAccessException` poderão ser lançadas na criação da nova instância na linha 14.

A aplicação gráfica que desenvolvemos utiliza essa implementação (Seção 5.5) para substituir diferentes figuras geométricas entre si. A Listagem 5.3 mostra um exemplo de uso do método `replaceImplementation()` para executar a substituição do componente e como utilizamos os Configuradores de Componentes para notificar os clientes do evento de substituição.

5.4 Políticas de Reconfiguração Dinâmica

A manipulação de dependências é controlada pelos Configuradores de Componentes utilizando os mecanismos já existentes de ganchos e clientes. É através desses mecanismos que se torna possível criar um grafo das interdependências entre os componentes, que será bastante útil durante a fase de reconfiguração dinâmica, pois facilita a tarefa de atualização dos clientes após a reconfiguração.

A manutenção da consistência do estado do componente é de suma importância. Mecanismos foram propostos (Seção 3.3) para garantir que a reconfiguração dinâmica do sistema preserve a sua consistência. Mas existem determinados casos nos quais esses mecanismos são desnecessários de acordo com a lógica do sistema em questão, como por exemplo no caso de reconfiguração de componentes sem estado (*stateless*). Deste modo, o uso de mecanismos para garantir a consistência de estado acaba resultando em uma maior sobrecarga e perda de desempenho do sistema, pois existiriam bloqueios de chamadas e sincronismos desnecessários durante a reconfiguração, fato que poderia ser evitado caso existisse alguma política de manipulação de reconfiguração dinâmica determinando como proceder em cada caso.

Nosso trabalho propõe a criação de políticas a serem desenvolvidas para a reconfiguração

dinâmica de componentes. Essas políticas são implementadas com o auxílio dos Configuradores de Componentes e determinam o comportamento desses Configuradores de Componentes durante a fase de reconfiguração, procurando trazer melhorias e não impactar negativamente no desempenho do sistema.

Manter o estado consistente dos componentes após a reconfiguração é apenas um dos requisitos necessários para a reconfiguração dinâmica, sendo necessário o uso de políticas que garantam esses requisitos. Definimos como Políticas de Reconfiguração Dinâmica o conjunto de regras que irão determinar o comportamento dos componentes durante a reconfiguração.

Em nosso arcabouço definimos três tipos de políticas em relação às garantias de consistência providas: fraca, média e forte. Nas Seções 5.4.1, 5.4.2 e 5.4.3 explicamos em mais detalhes cada uma. Na Seção 5.4.4 mostramos as diferenças entre elas durante a implementação e reconfiguração do sistema e na Seção 5.4.5 discutimos as vantagens e desvantagens de cada uma. A forma como implementamos essas políticas em nosso arcabouço é descrita na Seção 5.4.6.

5.4.1 Política de Consistência Fraca

A política de consistência fraca define a reconfiguração dinâmica para componentes que não possuem estado ou possuem estado imutável (somente leitura). Nesses casos, a política de reconfiguração é definida como fraca, pois não é necessário o uso de mecanismos que garantam a consistência dos dados do componente sendo reconfigurado. Até mesmo componentes com estado mas que não necessitam manter a sua consistência durante a reconfiguração dinâmica podem também usar esse tipo de política.

A reconfiguração dinâmica a ser feita utilizando essa política de consistência fraca será executada normalmente sem que haja a necessidade de se bloquear as chamadas de métodos de clientes ou utilizar qualquer outro mecanismo para se garantir a consistência.

O componente que não possuir estado não terá nada para se manter consistente e, portanto, pode utilizar essa política tranquilamente. Já o componente que possuir estado, estará implicitamente determinando que não necessita que o seu estado seja mantido consistente durante a reconfiguração.

Um exemplo de uso dessa política seria para componentes que executam cálculos matemáticos, como determinar se um dado número é primo. Esse componente não possui estado e, caso exista a necessidade de substituí-lo por outro de melhor algoritmo, basta atualizar os clientes para acessarem esse novo componente.

5.4.2 Política de Consistência Média

A política de consistência média define um tratamento da consistência da aplicação de forma mais flexível criando a possibilidade do componente escolher quais partes de seu código são necessários para garantir a sua consistência durante a reconfiguração dinâmica.

O componente poderá ter alguns métodos que devem ser bloqueados e outros que não precisam de nenhum mecanismo de bloqueio, tornando possível selecionar as partes que são realmente críticas para a sua reconfiguração e não bloquear desnecessariamente métodos que não precisam de consistência.

Como o componente não é totalmente bloqueado durante a reconfiguração temos um aumento da disponibilidade de seus serviços pois ele continua atendendo às chamadas dos clientes para métodos que não estejam sendo bloqueados.

Essa política se aplica para os componentes que possuem algum estado durante a sua existência e apenas parte dele deve ser mantido consistente. Mesmo que a outra parte desse estado se torne inconsistente, o próprio sistema é responsável por sincronizar o seu estado novamente. A política de consistência fraca não seria adequada para esse componente já que existe uma necessidade de consistência.

Um exemplo de cenário que se encaixa nessa política seria componentes com dados consultados por clientes esporadicamente, sendo que esses dados são atualizados pelo sistema com uma determinada frequência, por exemplo de hora em hora. Um cenário típico seria um componente armazenando o número de pessoas que atravessam a avenida Paulista por hora, uma das mais movimentadas da cidade de São Paulo. Esse número seria uma estimativa do número de pessoas com uma margem de erro pois dificilmente se conseguiria o número exato de pessoas. Assim, uma breve inconsistência nesses dados não traria graves conseqüências ao sistema, pois a atualização dos dados do componente é constante.

5.4.3 Política de Consistência Forte

A política de consistência forte define uma reconfiguração dinâmica que garante a consistência total do estado dos componentes durante e após a reconfiguração. Toda chamada de método do cliente ao componente será bloqueada no momento em que o componente estiver sendo reconfigurado. Após o processo de reconfiguração, essas chamadas são liberadas e retornadas pelo novo componente.

Deste modo faz-se necessário o uso de mecanismos de bloqueio de acesso ao componente sendo reconfigurado, de modo que ele esteja em um estado seguro e tenhamos a certeza de que seu estado não será alterado durante o processo de reconfiguração.

Essa política se aplica a aplicações críticas cujos dados (ou estado dos componentes) não podem de forma alguma ficar inconsistentes, como é o caso de sistemas onde vidas de pessoas estão em jogo ou até mesmo sistemas financeiros que exigem que dados não se corrompam durante as suas operações. Pode também ser utilizada em aplicações não tão críticas mas onde a perda da consistência traria a diminuição da qualidade de serviço oferecida pelo sistema.

5.4.4 Diferenças entre as Políticas

Cada política apresentada se aplica a domínios específicos de sistemas, dependendo da lógica de negócio de cada um. Devido a isso, existem muitas diferenças na implementação e durante a aplicação de cada política na reconfiguração dinâmica do sistema.

A implementação de cada política em nosso arcabouço utiliza os Configuradores de Componentes para determinarem o que fazer durante o processo de reconfiguração. Cada Configurator de Componente possui a implementação necessária para a política adotada caso ocorra alguma reconfiguração, conforme descrito na Seção 5.4.6.

Caso o sistema utilize a política de consistência fraca, os Configuradores de Componentes não bloqueiam as chamadas de métodos dos clientes e a reconfiguração procede normalmente. Caso seja adotada uma política de consistência média, os Configuradores de Componentes podem bloquear apenas as chamadas provenientes de determinados clientes (por exemplo, aqueles que desejam ter consistência, aqueles que acessavam versões antigas do componente) e, após a reconfiguração, desbloqueá-los.

Uma outra abordagem para a política de consistência média seria apenas bloquear determinados métodos do componente liberando aqueles que não deixariam o componente inconsistente. Para o caso de se utilizar uma política de consistência forte, os Configuradores de Componentes devem bloquear todas as chamadas dos clientes para poder iniciar o processo de reconfiguração. Podemos resumir essas diferenças nos seguintes itens:

- A política de consistência fraca não necessita bloquear chamadas de métodos;
- A política de consistência média é mais flexível, podendo bloquear algumas chamadas e outras não, dependendo do contexto de cada aplicação;
- A política de reconfiguração forte bloqueia todas as chamadas de modo a garantir a consistência total do sistema.

5.4.5 Vantagens e Desvantagens

Utilizando essas políticas em nosso arcabouço, temos como vantagem principal a possibilidade de escolher a melhor maneira de se executar a reconfiguração dinâmica do sistema de acordo com as necessidades de cada aplicação. Sem essas políticas poderíamos estar bloqueando clientes desnecessariamente resultando em uma sobrecarga maior e perda de desempenho do sistema que poderiam ser evitados.

Outra vantagem é a possibilidade de se utilizar diferentes políticas durante o tempo de vida do sistema, caso ele altere sua lógica de negócio, ou seja, a cada reconfiguração, é possível escolher qual política seguir. Essa escolha de política é mostrada na nossa aplicação de exemplo criada como demonstração de nosso arcabouço, descrita na Seção 5.5. Com isso, é possível escolher em tempo de execução qual política será adotada para o processo de reconfiguração.

No entanto, permitir ao usuário da aplicação escolher qual a política utilizar para a reconfiguração tem a desvantagem de estarmos confiando nas ações desse usuário, ou seja, ele pode fazer uma escolha incorreta e com isso tornar o sistema inconsistente após a reconfiguração.

5.4.6 Implementação das Políticas

Nosso arcabouço utiliza as políticas de consistência fraca, média e forte para determinar o comportamento da reconfiguração dinâmica de um componente. Implementamos essas políticas de forma que ao ser executada uma reconfiguração, o componente já sabe se comportar da forma determinada pela escolha de uma determinada política.

Para tanto, foi criada uma classe chamada `DynamicPolicy` responsável por armazenar qual política de consistência será usada pelo componente no momento da reconfiguração. Sua implementação está na Listagem 5.2.

Listing 5.2: Implementação da classe `DynamicPolicy`

```
1 package br.usp.ime.riko.dynamic.reconfiguration;
2
3 public class DynamicPolicy {
4
5     public static final int WEAK_POLICY = 0;
6     public static final int MEDIUM_POLICY = 1;
7     public static final int STRONG_POLICY = 2;
8
9     private int policy;
```

```
10
11 public DynamicPolicy(int policy) throws InvalidPolicyTypeException {
12     final boolean isWeakPolicy = policy == WEAK_POLICY;
13     final boolean isMediumPolicy = policy == MEDIUM_POLICY;
14     final boolean isStrongPolicy = policy == STRONG_POLICY;
15     final boolean policyExists =
16         isWeakPolicy || isMediumPolicy || isStrongPolicy;
17
18     if (!policyExists) {
19         throw new InvalidPolicyTypeException(
20             "Invalid policy value: " + policy);
21     }
22     this.policy = policy;
23 }
24
25 public int getPolicy() {
26     return policy;
27 }
28 }
```

O construtor da linha 11 cria uma instância de `DynamicPolicy` que será imutável e definida com uma política recebida como parâmetro desse construtor. Caso a política não exista, será lançada a exceção `InvalidPolicyTypeException` (linha 20).

A classe `DynamicPolicy` atua em conjunto com o processo de reconfiguração dinâmica, servindo para armazenar o valor da política a ser utilizada quando for necessário reconfigurar o componente.

5.5 Aplicação Gráfica

Foi desenvolvida em nossa pesquisa uma aplicação gráfica para ilustrar o uso do arcabouço de reconfiguração dinâmica estendendo o modelo de Configuradores de Componentes. Utilizou-se as interfaces mostradas na Seção 5.1 que foram estendidas também para atender às funcionalidades criadas por essa aplicação.

A Figura 5.2 mostra o diagrama de classes da aplicação gráfica. A interface `ShapeR0` estende do nosso arcabouço a interface `ReconfigurableObject` para prover métodos específicos ao formato

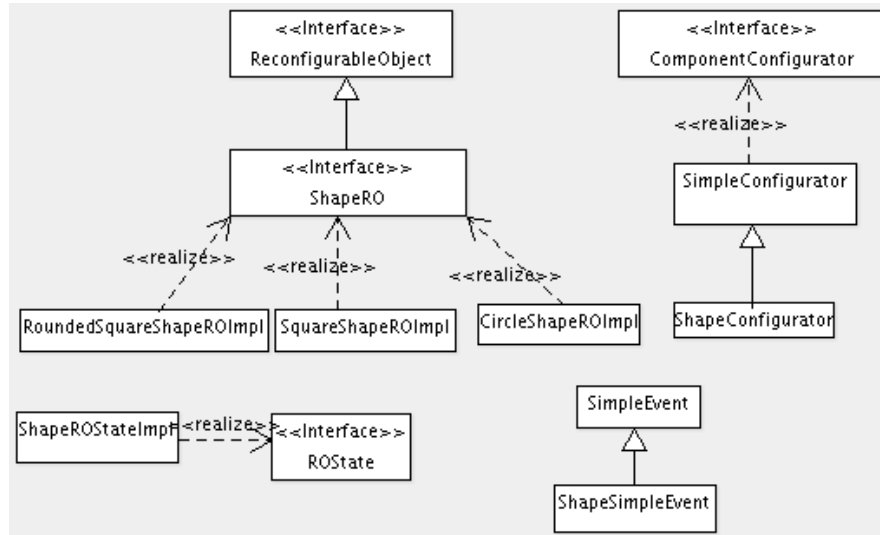


Figura 5.2: Diagrama de Classes da Aplicação Gráfica.

de figuras geométricas da nossa aplicação. Criamos três implementações de figuras geométricas: um círculo, um quadrado e um retângulo de cantos arredondados.

A aplicação mostra diversas figuras geométricas (círculos, quadrados etc) que possuem tamanho, nome e cores definidas em uma animação diminuindo e aumentando de tamanho. Ela foi desenvolvida utilizando *Java Swing* [JFC/Swing, [sítio](#)] de modo a torná-la portátil para vários sistemas operacionais. Veja na Figura 5.3 um exemplo com seis figuras geométricas sendo cinco círculos e um quadrado desenhados na aplicação.

Cada figura geométrica é um componente. Cada componente possui uma implementação na forma de uma classe Java que implementa a interface `ReconfigurableObject` e que é representada pela forma geométrica da figura (círculo, quadrado etc) diferenciando uma implementação da outra. O estado de um componente é o seu tamanho, cor, nome e ID da figura.

O número dentro de cada forma geométrica é o seu ID, e é incrementado na medida em que cada forma é substituída por outra através de uma chamada ao método `replaceImplementation`. A atribuição de ID a uma forma geométrica segue as seguintes regras:

1. Cada ID cresce seqüencialmente dentro de uma mesma implementação. Em outras palavras, não deverá existir um quadrado com o mesmo ID de outro quadrado;
2. É possível que exista formas geométricas distintas com o mesmo ID. Por exemplo, no início um quadrado e um círculo possuem o mesmo ID zero;

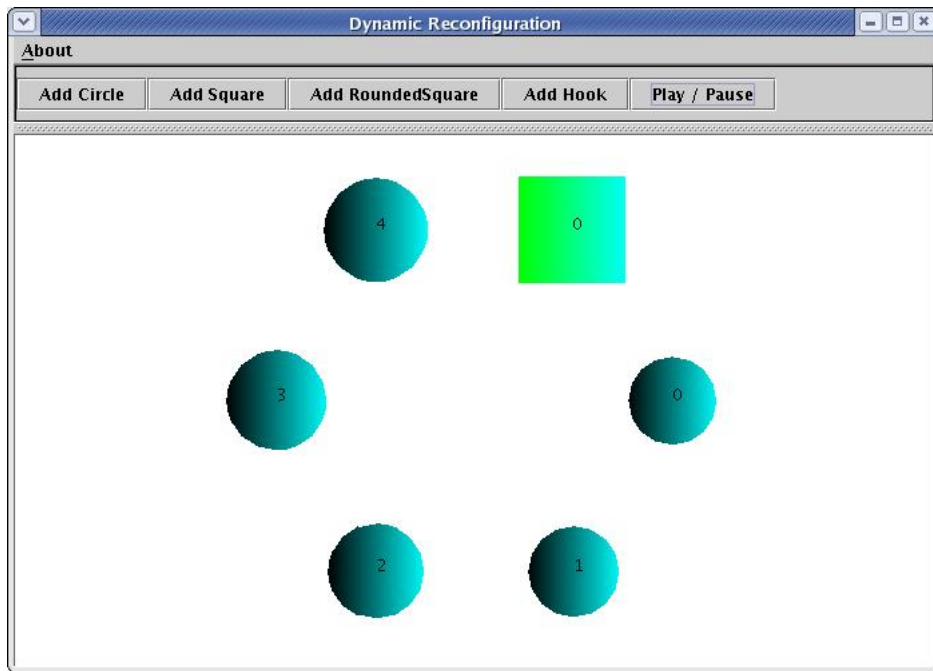


Figura 5.3: Aplicação Gráfica no seu estado inicial.

5 Extensão do Modelo

3. A reconfiguração (substituição) do componente não mantém o ID da figura original, o novo componente terá o seu ID gerado conforme a regra 1.

A regra 3 é necessária para se garantir que figuras geométricas iguais não possuam IDs repetidos após a reconfiguração. Por exemplo, seja um cenário inicial com um quadrado e um círculo, ambos com ID zero. Após a substituição do quadrado por um outro círculo e se mantivéssemos o ID, teríamos dois círculos com IDs iguais a zero, violando a primeira regra.

A Figura 5.4 mostra a substituição dinâmica de um círculo por um quadrado. O círculo possui ID 4 e será substituído por um outro quadrado. Ao selecionar no menu da figura a substituição dinâmica do componente círculo por uma implementação de quadrado, o componente passa por todas as fases de reconfiguração, exportando o seu estado (no caso a sua cor, tamanho) e importando no novo componente (no caso o quadrado).

A Figura 5.5 mostra o resultado final da reconfiguração dinâmica, com o novo quadrado no lugar do círculo. Como o ID é seqüencial dentro da mesma implementação, o quadrado novo terá ID 1, e mantém o estado do componente anterior, no caso a cor do círculo.

O objetivo da aplicação é ilustrar visualmente as possíveis reconfigurações dinâmicas que possam ocorrer durante o tempo de execução da aplicação. Por exemplo, é possível substituir dinamicamente a implementação de uma figura de círculo para quadrado, mantendo o estado do componente anterior (por exemplo a cor e o tamanho) e suas interdependências dinâmicas, auxiliado por um Configurador de Componente.

O papel do Configurador de Componente é manter as dependências entre os componentes, de modo que ocorrendo uma reconfiguração, o Configurador de Componente será responsável por atualizar as referências que o antigo componente possuía.

Suponha que os componentes círculos possuam uma dependência do primeiro componente quadrado de ID zero na aplicação. Portanto os círculos precisam se enganchar no quadrado utilizando os seus Configuradores de Componentes. A Figura 5.6 mostra um círculo adicionando um gancho utilizando o botão da aplicação **Add Hook**. Através de setas desenhadas na aplicação indicamos que um componente possui uma dependência de outro componente.

Um terceiro tipo de componente na nossa aplicação é o retângulo arredondado, cuja cor inicial é azul. Suponha que inserimos três retângulos na nossa aplicação e adicionamos ganchos entre os componentes para indicar as suas dependências.

Esse cenário é mostrado na Figura 5.7, com os círculos enganchados no quadrado de ID zero, e os quadrados enganchados nos retângulos arredondados. Existe apenas um retângulo arredondado que depende do círculo de ID zero.

Implementamos na nossa aplicação a possibilidade de um componente poder notificar os seus

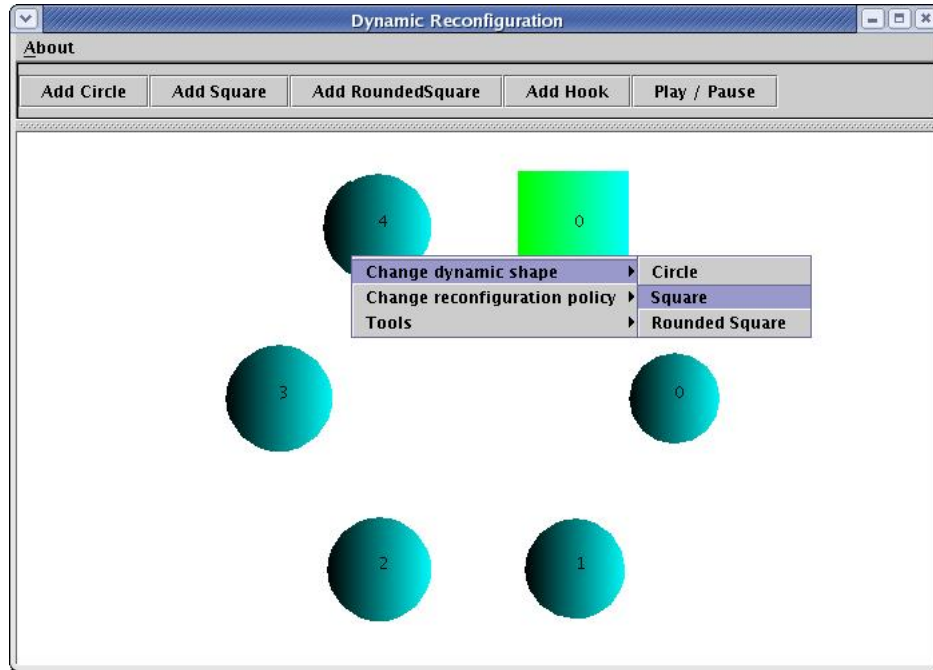


Figura 5.4: Substituição dinâmica do componente círculo por um quadrado.

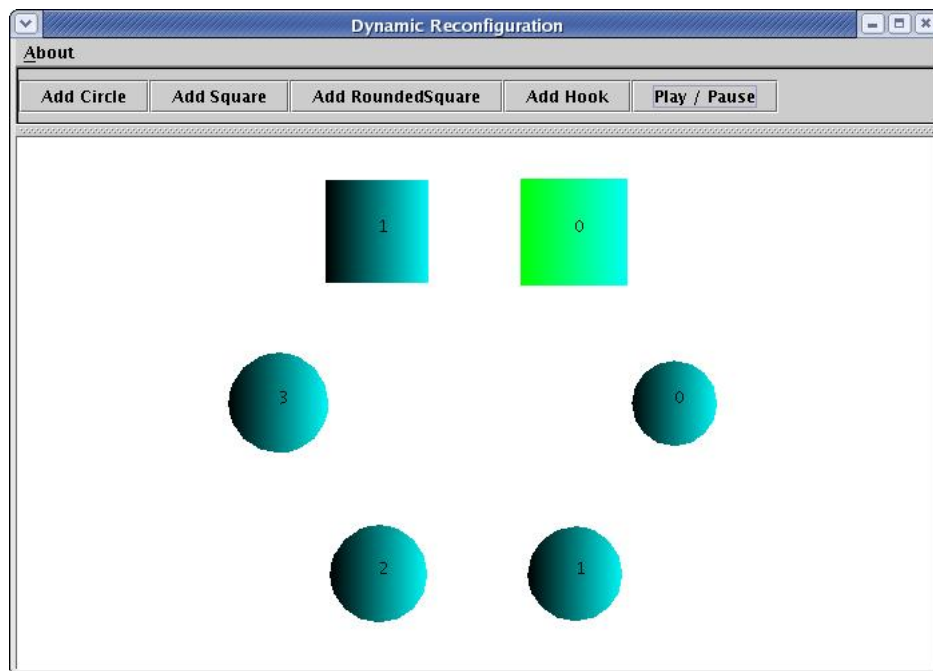


Figura 5.5: Resultado da substituição do componente círculo por um quadrado.

5 Extensão do Modelo

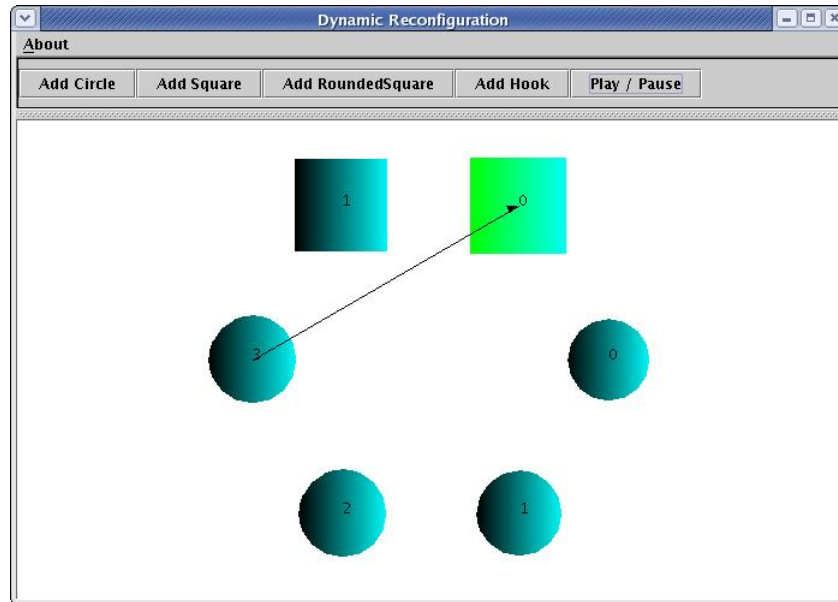


Figura 5.6: Adicionando ganchos para indicar dependências.

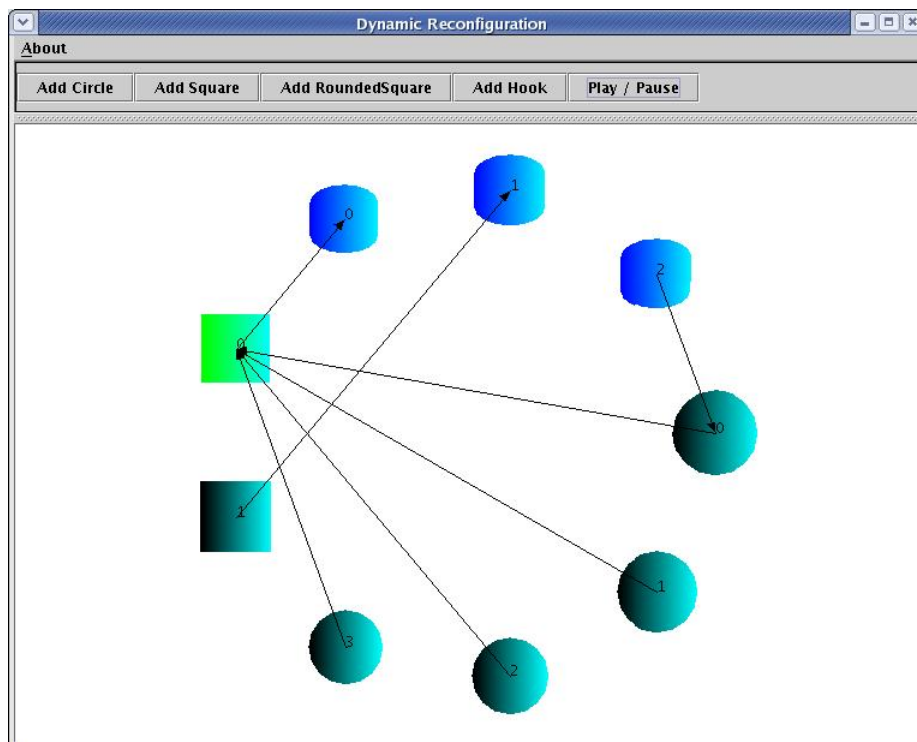


Figura 5.7: Adicionando novo componente retângulo arredondado e dependências.

clientes no caso de alguma alteração de sua cor. Deste modo os componentes deverão notificar os seus clientes utilizando o grafo de interdependências e notificá-los com a cor nova a ser utilizada.

A Figura 5.8 mostra o momento em que o quadrado irá notificar seus clientes para alterarem as suas cores. Essa alteração será válida tanto para os seus clientes diretos quanto indiretos, ou seja, os círculos que são seus clientes diretos terão as suas cores alteradas e o retângulo arredondado que é um cliente indireto (via componente círculo) também terá a sua cor alterada.

O resultado final da alteração de cores dos clientes do quadrado é mostrado na Figura 5.9. Para que a notificação aos clientes tenha êxito, cada componente possui um Configurator de Componente gerenciando as suas interdependências (no caso a cor), e recebe um evento de reconfiguração através de seus ganchos e clientes. Esse evento possui atributos específicos que ajudam o Configurator de Componentes como proceder no momento da reconfiguração.

A dependência entre os componentes é gerenciada pelo Configurator de Componente, e a aplicação gráfica utiliza a visualização das setas para indicar essas interdependências. A cor tem um papel importante na nossa aplicação para mostrar o que ocorre no evento de substituição de uma figura geométrica por uma outra.

A Listagem 5.3 mostra como ocorre a chamada do método `replaceImplementation()` para substituir a implementação de uma figura geométrica. Feita a substituição, utiliza-se os Configuradores de Componente para notificar os clientes que houve um evento, que deverão ter as suas cores alteradas após o recebimento desse evento.

Listing 5.3: Exemplo de chamada do método `replaceImplementation()` na aplicação gráfica

```

1 public void changeShape(String newShapeImplementation) {
2     try {
3         // faz a substituição pelo newShapeImplementation
4         System.out.println("changing shape to " + newShapeImplementation);
5         shapeRO = (ShapeRO) shapeRO.replaceImplementation(
6             newShapeImplementation);
7
8         // notifica clientes
9         System.out.println("**** Implementation replaced - notifying
10            clients ****\n");
11         ShapeConfigurator newShapeConfigurator = (ShapeConfigurator)
12             shapeRO.getConfigurator();
13         newShapeConfigurator.notifyClients(new SimpleEvent("Evento
14             REPLACED", ComponentEvent.REPLACED));

```

5 Extensão do Modelo

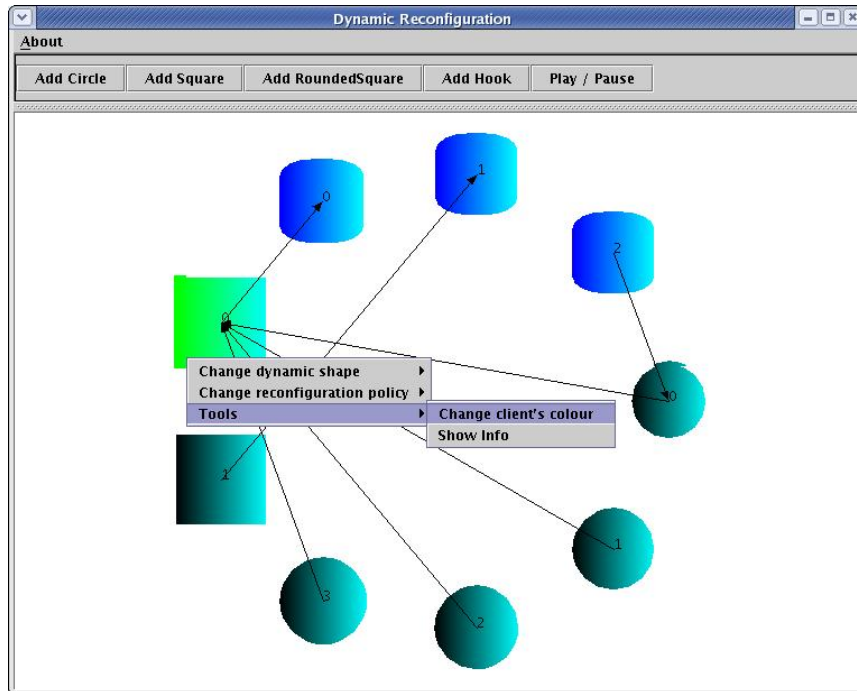


Figura 5.8: Notificando clientes para alteração de cor.

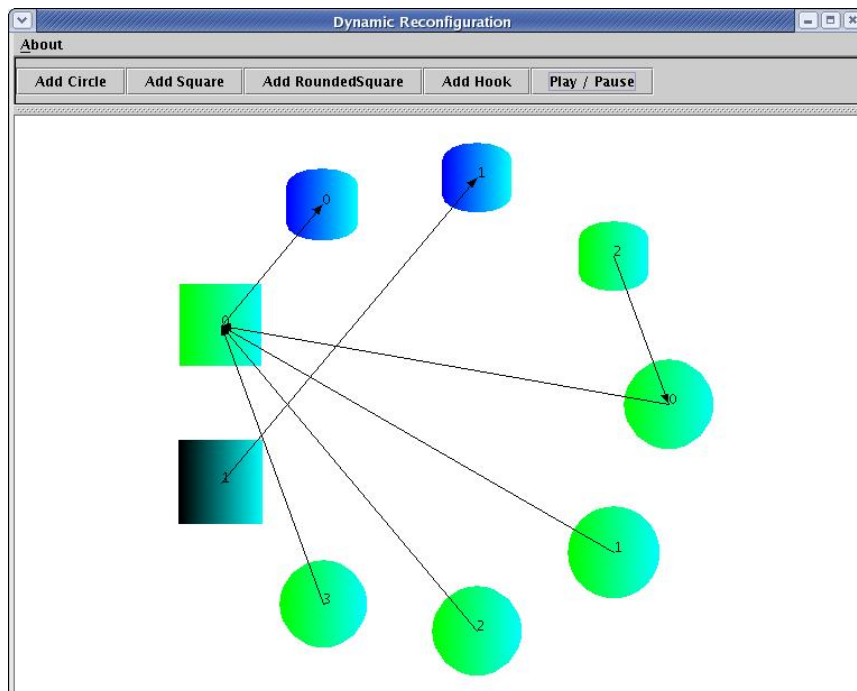


Figura 5.9: Resultado final da notificação de alteração de cor.

```
11
12     System.out.println("==== Implementation replaced successfully.
13         Calling garbage collector");
14
15     repaint();
16     System.gc();
17 } catch (Exception e) {
18     e.printStackTrace();
19 }
```

A linha 5 invoca o método `replaceImplementation()` passando como parâmetro o nome da classe da nova implementação (`newShapeImplementation`). Esse método retorna um `ReconfigurableObject` e é feito o *cast* para um `ShapeRO`.

A partir do novo `shapeRO`, obtém-se o seu Configurador de Componente (linha 9) implementado pela classe `ShapeConfigurator`. Essa classe é uma extensão da implementação de referência do modelo explicado no Capítulo 4 e é usada para notificar os Configuradores de Componentes dos clientes de que houve um evento de substituição de componente (linha 10).

O evento de substituição é recebido pelo Configurador de Componente do cliente e este altera a sua cor para a mesma do componente no qual o cliente está enganchado.

O código fonte e documentação do arcabouço e da aplicação de exemplo estão disponíveis no endereço [[riko, sítio](#)].

5.6 Análise de desempenho

Utilizamos a aplicação gráfica para avaliar o tempo necessário para se executar a reconfiguração de um componente. Criamos uma bateria de testes com cinquenta substituições de cada componente e medimos o tempo de duração de cada reconfiguração.

Como existem três tipos de componentes em nossa aplicação gráfica (quadrado, círculo e quadrado arredondado) cada bateria de teste consumiu cento e cinquenta reconfigurações. Ao final de cada bateria obtivemos a média final do tempo consumido para os testes.

Em nosso arcabouço, durante a reconfiguração, o Configurador de Componentes notifica seus clientes enviando um evento de substituição. Cada tempo medido engloba a operação de reconfiguração mais o tempo que cada Configurador precisou para notificar todos os seus clientes.

A Tabela 5.1 mostra os tempos médios em milissegundos obtidos para cada bateria de teste.

5 Extensão do Modelo

Inicialmente, medimos a reconfiguração de apenas um componente sem nenhum cliente e fomos adicionando mais clientes até chegarmos ao total de dez clientes. Executamos três baterias de testes para cada quantidade de clientes, obtendo a média dos tempos na última coluna.

	Média 1	Média 2	Média 3	Média Final
0 Cliente	32,94	32,00	32,95	32,63
1 Clientes	34,07	33,78	32,88	33,57
2 Clientes	33,53	33,23	34,51	33,76
3 Clientes	36,17	35,96	35,50	35,88
4 Clientes	35,78	39,57	38,75	38,04
5 Clientes	39,88	38,28	38,73	38,96
10 Clientes	48,36	48,38	50,61	49,11

Tabela 5.1: Tabela de Tempo Médio de Reconfiguração (em milissegundos)

É interessante notar que, mesmo para um componente com dez clientes precisando ser notificados durante a reconfiguração, o tempo médio ficou por volta de 50ms, ou seja, a aplicação precisou esperar esse pequeno tempo para voltar a ser operável.

Os tempos foram obtidos em um computador com sistema operacional Linux Fedora Core3 com processador Pentium4 HT 2.6Ghz e 1GB de memória RAM. A versão do Java utilizada foi a j2sdk1.4.2_08.

5.7 Considerações Finais

A aplicação gráfica demonstra as funcionalidades de reconfiguração do nosso arcabouço mas não é um exemplo de cenário de uma aplicação mais realista. No entanto, é possível utilizar nosso modelo em diversas aplicações críticas.

Um exemplo de caso real poderia ser um sistema hospitalar que centralizasse todas as informações de pacientes a serem consultadas pelos médicos a qualquer hora. Esse sistema teria uma interface possibilitando o médico efetuar sua consulta, que se comunicaria com componentes críticos do sistema contendo a informação desejada.

Caso surja a necessidade de atualizar esses componentes (por exemplo uma correção de falha na busca de pacientes ou adição de novas funcionalidades a essa busca), o sistema não poderia ser parado pois vidas de pacientes estariam em perigo. Utilizando nosso arcabouço seria possível executar a reconfiguração do sistema dinamicamente com uma interrupção parcial de seus serviços.

Outro exemplo de uso seria em sistemas cuja disponibilidade de seus serviços seja 24 x 7

(24 horas por dia, 7 dias por semana). Um sistema de Autoridade Certificadora de certificados digitais se encontra nesse caso. Uma Autoridade Certificadora [[ICP-Brasil, sítio](#)], [[ITI, sítio](#)] tem a responsabilidade principal de emitir e revogar (cancelar) certificados digitais, além de emitir uma lista atualizada dos certificados revogados, chamada de LCR. Dependendo da política definida pela Autoridade Certificadora, a LCR normalmente é atualizada e emitida a cada uma hora e deve estar sempre disponível no esquema 24 x 7.

No caso da necessidade de manutenção de uma Autoridade Certificadora, ela não poderia ficar mais de uma hora sem o serviço de emissão de LCR. Possuindo suporte à reconfiguração dinâmica, esse serviço não seria interrompido totalmente e a LCR não correria o risco de ficar inválida.

Capítulo 6

Conclusões

Nós apresentamos um arcabouço em Java para dar suporte a reconfiguração dinâmica de sistemas baseados em componentes. Estendendo o modelo de Configuradores de Componentes, criamos um arcabouço em Java implementando mecanismos para a reconfiguração de sistema.

O arcabouço foi construído de forma a ser genérico o suficiente para que outros possam criar aplicações reconfiguráveis dinamicamente, bastando implementar as interfaces criadas no nosso modelo. Nós desenvolvemos uma aplicação gráfica a partir de nosso arcabouço, mostrando como é possível reconfigurar dinamicamente componentes em tempo de execução.

Definimos também políticas de reconfiguração dinâmica que servem para determinar a maneira que será feita a reconfiguração dos componentes, mostrando o seu uso na nossa aplicação de exemplo. Com isso, estamos contribuindo com a definição de um mecanismo adicional para controlar a reconfiguração, tornando esse processo um pouco mais flexível em relação a trabalhos anteriores.

6.1 Contribuição

Nosso trabalho tem como principal contribuição a implementação de um modelo para reconfiguração dinâmica utilizando as interdependências entre os componentes. Estendendo o modelo de Configuradores de Componentes, mostramos como é possível notificar todos os clientes dependentes do componente a ser reconfigurado.

Realizamos também um estudo dos principais trabalhos relacionados na área de reconfiguração dinâmica e detalhamos quais os principais mecanismos utilizados.

As políticas de reconfiguração dinâmica definidas pelo nosso arcabouço também contribuem de forma a se criar uma nova maneira de controle do comportamento dos componentes envolvidos

6 Conclusões

na reconfiguração.

Nosso trabalho criou uma implementação de referência para mostrar o uso do nosso arcabouço para reconfiguração dinâmica. A aplicação gráfica desenvolvida com Java Swing possui como índice LOC (*Lines Of Code*) 1728 linhas, e como índice NLOC (*Non-Comment Lines Of Code*) o valor de 1241 linhas.

Esses valores foram obtidos utilizando um software *Lines Of Codes Counter* [LOC, [sítio](#)] e os resultados detalhados estão nas Tabelas 6.1 (valores para LOC) e 6.2 (valores para NLOC).

```
[riko@yuka-linux src]$ find . -name '*.java' | java -cp ../lib/LOC.jar textui.LOC
00008 ./br/usp/ime/riko/dynamic/reconfiguration/NotReconfigurableException.java
00011 ./br/usp/ime/riko/dynamic/reconfiguration/ROState.java
00028 ./br/usp/ime/riko/dynamic/reconfiguration/ReconfigurableObject.java
00009 ./br/usp/ime/riko/dynamic/reconfiguration/UnexpectedROStateException.java
00018 ./br/usp/ime/riko/dynamic/reconfiguration/DynamicPolicy.java
00009 ./br/usp/ime/riko/dynamic/reconfiguration/InvalidPolicyTypeException.java
00009 ./br/usp/ime/riko/dynamic/reconfiguration/UnexpectedReconfigurationException.java
00152 ./br/usp/ime/riko/dynamic/shape/CircleShapeROImpl.java
00153 ./br/usp/ime/riko/dynamic/shape/RoundedSquareShapeROImpl.java
00107 ./br/usp/ime/riko/dynamic/shape/ShapeConfigurator.java
00027 ./br/usp/ime/riko/dynamic/shape/ShapeRO.java
00048 ./br/usp/ime/riko/dynamic/shape/ShapeROStateImpl.java
00147 ./br/usp/ime/riko/dynamic/shape/SquareShapeROImpl.java
00009 ./br/usp/ime/riko/dynamic/shape/ShapeSimpleEvent.java
00232 ./br/usp/ime/riko/swing/CircleLayout.java
00147 ./br/usp/ime/riko/swing/JPanelAllShapes.java
00099 ./br/usp/ime/riko/swing/JPanelPopupMenu.java
00179 ./br/usp/ime/riko/swing/MainFrame.java
00271 ./br/usp/ime/riko/swing/PaintJPanel.java
00023 ./br/usp/ime/riko/swing/InfoJDialog.java
00042 ./br/usp/ime/riko/swing/DrawArrow.java
—
01728
```

Tabela 6.1: Tabela de Linhas de Código (LOC)

```
[riko@yuka-linux src]$ find . -name '*.java' | java -cp ../lib/LOC.jar textui.LOC -n
00005 ./br/usp/ime/riko/dynamic/reconfiguration/NotReconfigurableException.java
00004 ./br/usp/ime/riko/dynamic/reconfiguration/ROState.java
00008 ./br/usp/ime/riko/dynamic/reconfiguration/ReconfigurableObject.java
00005 ./br/usp/ime/riko/dynamic/reconfiguration/UnexpectedROStateException.java
00016 ./br/usp/ime/riko/dynamic/reconfiguration/DynamicPolicy.java
00005 ./br/usp/ime/riko/dynamic/reconfiguration/InvalidPolicyTypeException.java
00005 ./br/usp/ime/riko/dynamic/reconfiguration/UnexpectedReconfigurationException.java
00113 ./br/usp/ime/riko/dynamic/shape/CircleShapeROImpl.java
00111 ./br/usp/ime/riko/dynamic/shape/RoundedSquareShapeROImpl.java
00067 ./br/usp/ime/riko/dynamic/shape/ShapeConfigurator.java
00014 ./br/usp/ime/riko/dynamic/shape/ShapeRO.java
00031 ./br/usp/ime/riko/dynamic/shape/ShapeROStateImpl.java
00111 ./br/usp/ime/riko/dynamic/shape/SquareShapeROImpl.java
00005 ./br/usp/ime/riko/dynamic/shape/ShapeSimpleEvent.java
00157 ./br/usp/ime/riko/swing/CircleLayout.java
00121 ./br/usp/ime/riko/swing/JPanelAllShapes.java
00080 ./br/usp/ime/riko/swing/JPanelPopupMenu.java
00142 ./br/usp/ime/riko/swing/MainFrame.java
00184 ./br/usp/ime/riko/swing/PaintJPanel.java
00017 ./br/usp/ime/riko/swing/InfoJDialog.java
00040 ./br/usp/ime/riko/swing/DrawArrow.java
—
01241
```

Tabela 6.2: Tabela de Linhas de Código Sem Comentários (NCLOC)

6.2 Trabalhos Futuros

O arcabouço apresentado nessa dissertação é apenas um modelo inicial de extensão dos Configuradores de Componentes. É possível melhorar a nossa idéia em vários aspectos:

- Em nossa pesquisa não abordamos os aspectos transacionais existentes durante a reconfiguração dinâmica dos componentes. Caso um componente se reconfigure mas ocorra uma falha em algum de seus clientes, é necessário que exista o conceito de voltar atrás para o estado original dos componentes (executar um “*rollback*” da reconfiguração);
- A tolerância a falhas é uma área bem vasta e interessante para se aplicar na reconfiguração dinâmica. É preciso estudar os diversos casos de tratamento no caso de falhas de componentes durante ou não o processo de reconfiguração;
- A implementação de sincronização entre os bloqueios das chamadas dos clientes e a operação de reconfiguração do componente não é trivial pois existe o risco de deixar o sistema em *deadlock*;
- Nossa pesquisa apresentou uma aplicação gráfica de exemplo, mas faltou uma implementação de caso mais real que nos permitisse estudar com mais detalhes o uso de nosso arcabouço;
- O nosso modelo de execução se restringiu a componentes e chamadas de métodos locais. Migrar o nosso arcabouço para dar suporte a sistemas distribuídos representa uma evolução natural do nosso trabalho, pois aplicações desse domínio distribuído se beneficiariam muito mais com uma reconfiguração dinâmica transparente aos seus clientes.

Referências Bibliográficas

- [1stICCDS, 1992] 1stICCDS (1992). *Intl. Workshop on Configurable Dist. Systems*, London, England. 2, 70
- [2ndICCDS, 1994] 2ndICCDS (1994). *2nd Intl. Workshop on Configurable Dist. Systems*, Pittsburgh, PA. 2
- [3rdICCDS, 1996] 3rdICCDS (1996). *3rd Intl. Workshop on Configurable Dist. Systems*, Annapolis, MD. 2
- [4thICCDS, 1998] 4thICCDS (1998). *4th Intl. Conf. on Configurable Dist. Systems*, Annapolis, MD. 2, 69
- [Almeida et al., 2001a] Almeida, J. P. A., Wegdam, M., Pires, L. F., and Sinderen, M. V. (2001a). An approach to dynamic reconfiguration of distributed systems based on object middleware. In *19o. Simpósio Brasileiro de Redes de Computadores*. 12, 26
- [Almeida et al., 2001b] Almeida, J. P. A., Wegdam, M., van Sinderen, M., and Nieuwenhuis, L. (2001b). Transparent Dynamic Reconfiguration for CORBA. In *3rd International Symposium on Distributed Object and Applications*, Rome, Italy. 2, 12, 21, 22, 26
- [Batista and Rodriguez, 2000] Batista, T. and Rodriguez, N. (2000). Dynamic reconfiguration of component-based applications. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 32. IEEE Computer Society. 14, 29
- [Batista and Carvalho, 1999] Batista, T. V. and Carvalho, M. G. (1999). Component-Based Applications: A Dynamic Reconfiguration Approach with Fault Tolerance Support. *Eletronic Notes in Theoretical Computer Science*, 65(4):9. 14, 29
- [Bidan et al., 1998] Bidan, C., Issarny, V., Saidakis, T., and Zarras, A. (1998). A Dynamic Reconfiguration Service for CORBA. In [4thICCDS, 1998], pages 35–42. 2, 10, 23, 26
- [Bloom, 1983] Bloom, T. (1983). *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT. Also as MIT LCS Tech. Report 303. 7

Referências Bibliográficas

- [Cerqueira et al., 1999] Cerqueira, R., Cassino, C., and Ierusalimschy, R. (1999). Dynamic Component Gluing Accross Different Componentware Systems. *DOA 99*, pages 362–371. 14
- [da Silva e Silva, 2003] da Silva e Silva, F. J. (2003). *Adaptação Dinâmica de Sistemas Distribuídos*. PhD thesis, Instituto de Matemática e Estatística da Universidade de São Paulo. 2, 15
- [da Silva e Silva et al., 2003] da Silva e Silva, F. J., Endler, M., and Kon, F. (2003). Developing adaptive distributed applications: a framework overview and experimental results. In *Proceedings of the International Conference on Distributed Objects and Applications (DOA'03)*, LNCS 2888, pages 1275–1291, Catania. Springer-Verlag. 15
- [Fitzpatrick et al., 1998] Fitzpatrick, T., Blair, G. S., Coulson, G., Davies, N., and Robin, P. (1998). A software architecture for adaptive distributed multimedia systems. In *IEE Proceedings on Software*, volume 145, pages 163–171. 28
- [Fleury and Reverbel, 2003] Fleury, M. and Reverbel, F. (2003). The JBoss extensible server. In Endler, M. and Schmidt, D., editors, *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of LNCS, pages 344–373. Springer-Verlag. 16
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA. 44
- [Grannon, 1997] Grannon, D. (1997). Transparent Dynamic Reconfiguration for CORBA. *Workshop on Compositional Software Architectures*. 33
- [Hofmeister et al., 1992] Hofmeister, C., White, E., and Purtilo, J. (1992). Surgeon: A packager for dynamically reconfigurable distributed applications. In [1stICCDs, 1992], pages 164–175. 9
- [Hofmeister, 1994] Hofmeister, C. R. (1994). *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Department of Computer Science, University of Maryland. 9, 29
- [Hofmeister and Purtilo, 1993] Hofmeister, C. R. and Purtilo, J. M. (1993). A framework for dynamic reconfiguration of distributed programs. Technical Report CS-TR-3119, University of Maryland. 7
- [Ierusalimschy et al., 1996] Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. (1996). Lua - an extensible extension language. *Software, Practice and Experience*, 26(6):635–652. 14
- [Kon, 2000] Kon, F. (2000). *Automatic Configuration of Component-Based Distributed Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign. 2, 13, 34, 40

- [Kon and Campbell, 1999] Kon, F. and Campbell, R. H. (1999). Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 175–187, San Diego, CA. 34
- [Kon and Campbell, 2000] Kon, F. and Campbell, R. H. (2000). Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, 8(1):26–36. 4, 36
- [Kon et al., 2000a] Kon, F., Campbell, R. H., Mickunas, M. D., Nahrstedt, K., and Ballesteros, F. J. (2000a). 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh. 7, 13
- [Kon et al., 2005] Kon, F., Marques, J. R., Yamane, T., Campbell, R. H., and Mickunas, M. D. (2005). Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems. *Software: Practice and Experience*. John Wiley & Sons, Inc. Publisher. 13
- [Kon et al., 2000b] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L. C., and Campbell, R. H. (2000b). Monitoring, Security, and Dynamic Configuration with the dynamic-TAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York. Springer-Verlag. 14
- [Kramer and Magee, 1985] Kramer, J. and Magee, J. (1985). Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436. 8, 26
- [Kramer and Magee, 1990] Kramer, J. and Magee, J. (1990). The Evolving Philosophers Problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306. 8
- [Magee et al., 1989] Magee, J., Kramer, J., and Sloman, M. (1989). Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6). 8
- [Microsoft, 1998] Microsoft (1998). DCOM: A Technical Overview. Technical report, Microsoft Corporation. 2
- [Microsoft, 2000] Microsoft (2000). Microsoft .NET. Technical report, Microsoft Corporation. 2, 33
- [Moazami-Goudarzi, 1999] Moazami-Goudarzi, K. (1999). *Consistency Preserving Dynamic Re-configuration of Distributed Systems*. PhD thesis, Imperial College London. 24
- [OMG, 1998] OMG (1998). *CORBA services: Common Object Services Specification*. Object Management Group, Framingham, MA. OMG Document 98-12-09. 10

Referências Bibliográficas

- [OMG, 2002] OMG (2002). *CORBA Component Model*. Object Management Group. Version 3.0, OMG Document formal/2002-06-65. 2, 33
- [OMG, 2004] OMG (2004). *CORBA v3.03 Specification*. Object Management Group. OMG Document formal/2004-03-01. 10
- [Purtilo, 1990] Purtilo, J. M. (1990). *The POLYLITH Software Toolbus*. Technical Report CS-TR-2469, University of Maryland. 9, 29
- [RM2000, 2000] RM2000 (2000). *Workshop on Reflective Middleware*, New York, USA. 3
- [RM2003, 2003] RM2003 (2003). *The 2nd Workshop on Reflective and Adaptive Middleware*, Rio de Janeiro, Brazil. 3
- [RM2004, 2004] RM2004 (2004). *The 3rd Workshop on Reflective and Adaptive Middleware*, Toronto, Ontario, Canada. 3
- [Schmidt et al., 2000] Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture*, volume Vol. 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons. 17
- [Sun Microsystems, 2003] Sun Microsystems (2003). *Enterprise Java Beans Specification*. Sun Microsystems. Version 2.1 Final Release. 2, 33
- [Szyperski, 2002] Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2, 33
- [Tang, 2000] Tang, Z. (2000). *Dynamic reconfiguration of component-based applications in Java*. Master's thesis, MIT. 10, 11, 29, 30, 32

Referências a Sítios na Internet

- [ComponentConfigurator, sítio] ComponentConfigurator. <http://choices.cs.uiuc.edu/2k/ComponentConfigurator>. Último acesso em fevereiro/2005. 38
- [JBoss, sítio] JBoss. <http://www.jboss.org>. Último acesso em fevereiro/2005. 16
- [JMX, sítio] JMX. <http://java.sun.com/products/JavaManagement/index.jsp>. Último acesso em fevereiro/2005. 16
- [Java, sítio] Java. <http://java.sun.com>. Último acesso em fevereiro/2005. 5
- [J2EE, sítio] J2EE. <http://java.sun.com/j2ee>. Último acesso em fevereiro/2005. 16
- [SNMP, sítio] SNMP. <http://www.ietf.org/rfc/rfc1157.txt>. Último acesso em fevereiro/2005. 18
- [CIM/WBEM, sítio] CIM/WBEM. http://www.snia.org/tech_activities/SMI/cim. Último acesso em fevereiro/2005. 18
- [CORBA, sítio] CORBA. <http://www.corba.org>. Último acesso em fevereiro/2005.
- [LDAP, sítio] LDAP. <http://www.ietf.org/rfc/rfc3377.txt>. Último acesso em fevereiro/2005. 18
- [riko, sítio] riko. <http://www.ime.usp.br/~riko>. Último acesso em fevereiro/2005. 61
- [JFC/Swing, sítio] JFC/Swing. <http://java.sun.com/products/jfc/index.jsp>. Último acesso em fevereiro/2005. 54
- [Microsoft/.NET, sítio] Microsoft/.NET. <http://www.microsoft.com/net>. Último acesso em fevereiro/2005. 28
- [Java/EJB, sítio] Java/EJB. <http://java.sun.com/products/ejb>. Último acesso em fevereiro/2005. 28, 34
- [CORBA/CCM, sítio] CORBA/CCM. <http://www.omg.org/technology/documents/formal/components.htm>. Último acesso em fevereiro/2005. 33

Referências a Sítios na Internet

[Microsoft, sítio] Microsoft. <http://www.microsoft.com>. Último acesso em fevereiro/2005. 34

[2K, sítio] 2K. <http://srg.cs.uiuc.edu/2k>. Último acesso em fevereiro/2005. 13

[LOC, sítio] LOC. <http://www.csc.calpoly.edu/~jdalbey/SWE/PSP/LOChelp.html>. Último acesso em junho/2005. 66

[ICP-Brasil, sítio] ICP-Brasil. <http://www.icpbrasil.gov.br/>. Último acesso em junho/2005. 63

[ITI, sítio] ITI. <http://www.iti.br/>. Último acesso em junho/2005. 63

Índice Remissivo

- 2K, [13](#)
 - Automatic Configuration Service, [13](#)
 - dynamicTAO, [14](#)
- adaptação, [3](#)
 - adaptação dinâmica, [15](#)
- aplicação gráfica, [5](#), [53](#)
- arcabouço, [5](#), [53](#), [61](#)
 - aplicação gráfica, [46](#)
 - extensão do modelo, [43](#)
 - interfaces, [45](#)
 - políticas, [49](#), [52](#)
- Argus, [7](#), [8](#)
- arquitetura, [12](#), [17](#)
- aspectos transacionais, [68](#)
- atributos, [40](#)
- atributos de dependência, [40](#)
- auto-configuração, [3](#)
- barramento de dados, [7](#)
- CCM, *veja* CORBA Componente Model, [33](#)
- chamadas reentrantes, [12](#)
- CIM, [18](#)
- classe pai, [37](#)
- complexidade, [1](#)
- componentes
 - arquitetura, [34](#)
 - definição, [33](#)
 - interdependentes, [34](#)
 - sistema, [3](#)
- computação móvel, [1](#), [2](#)
- Configuradores de Componentes, [4](#), [5](#)
 - 2K, [13](#)
 - clientes, [36](#)
 - definição, [33](#)
 - extensão, [43](#)
 - ganchos, [36](#)
 - modelo, [34](#)
- Conic, [8](#)
- consistência, [4](#), [45](#)
 - manutenção do estado, [48](#)
- construtores, [11](#)
- contêiner, [34](#)
- CORBA, [10](#), [12–14](#), [33](#), [34](#)
 - CORBA Component Model, [2](#), [33](#)
 - COS, [10](#)
 - reconfiguração dinâmica, [12](#)
- dependência, [17](#)
- dependências dinâmicas, [4](#), [13](#)
- deployment, [2](#)
- diagrama de classes, [53](#)
- dinamismo, [1](#)
- disponibilidade, [1](#), [2](#)
- diversidade, [1](#)
- dynamicTAO, [14](#)
- estado, [11](#)
 - transferência, [45](#)
- evento, [40](#)
- evolução, [4](#)
- gancho, [37](#), [38](#)
- gerenciamento, [17](#)
- guardião, [8](#)
- guardião, *veja* guardiões
- guardiões

- Argus, 7
- heterogeneidade, 1
- Hofmeister, 9
- Illinois, 13
- Imperial College, 8
- interceptadores, 15
- interdependências, 5, 7, 36
- interface, 2, 8, 11, 17
- interrupção, 3
- intervenção manual, 3
- J2EE, 16
- Java, 11, 12, 17, 38
 - EJB, 28
 - Enterprise JavaBeans, 2, 33
 - J2EE, 16
 - JBoss, 16
 - JMX, 16, 18
 - máquina virtual, 12
 - Swing, 54
- JBoss, 17
- Kramer, 8
- LDAP, 18
- LuaOrb, 14
- Magee, 8
- MBean, 17
 - agentes, 17
 - MBean Server, 17
- mecanismo, 11
- mecanismos, 5, 48
- Microsoft
 - .NET, 2, 28, 33
 - DCOM, 2
- middleware, 13, 16
- MIT, 7
- mobilidade, 1
- monitoração, 17
- monitoramento, 15
- Object Request Broker, 10, 13
- objeto, 40
- objetos, 4, 13, 33
 - reconfiguráveis, 43
- Objetos Reconfiguráveis, 5
- off-the-shelf components, 2
- ORB, *veja* Object Request Broker, 10, 12–14
- Paradas em sistemas, 3
- perda de desempenho, 3
- políticas, 4, 5, 65
 - manipulação, 4
 - políticas de consistência, 5
- POLYLITH, 9
- ponteiro, 38
- pontos de reconfiguração, 10
- POO, 1, *veja* Programacao Orientada a Objetos
- Programação Orientada a Objetos, 1
- protocolos, 18
- Purtilo, 9
- QoS, *veja* Quality of Service, 2
- Qualidade de Serviço, 3
- Quality of Service, *veja* Qualidade de Serviço
- quiescence, 8, 9
- reconfiguração, 41
- reconfiguração dinâmica
 - definição, 3
 - eventos, 34
 - mecanismos, 5
 - quiescence, 8
 - substituição, 46
 - workshops, 2
- Reificação, 34
- SimpleConfigurator, 37
- sincronização, 68
- sistemas baseados em componentes, 2, 33
- sistemas distribuídos, 68
- sistemas reflexivos, 36
- SNMP, 18

Surgeon, [9](#)

tempo de execução, [17](#)

tempo de execução, [5](#)

tempo de vida, [3](#)

Toby Bloom, [7](#)

tolerância a falhas, [68](#)

usuário, [8](#)