Suporte ao Desenvolvimento e Uso de Componentes Flexíveis

Ricardo Pereira e Silva Eng., M.Sc.
Universidade Federal de Santa Catarina - Depto. de Informática e de Estatística
Florianópolis - SC, ricardo@inf.ufsc.br

Roberto Tom Price Eng., M.Sc., D.Phil.
Universidade Federal do Rio Grande do Sul - Instituto de Informática
Porto Alegre - RS, tomprice@inf.ufrgs.br

Abstract

Component-oriented development is being stimulated by the availability of devices allowing component interoperability, regardless of programming language, running platform and executing location. However, there are some unsolved problems concerning search and selection, functionality understanding and adaptation of components.

Flexibility is an important feature for increasing component reusability. Some approaches like wrappers, help in adapting actual components to systems not fully compatible. In this paper we propose to develop components as object-oriented frameworks. This leads to flexible components, much more suitable to reuse. On the other hand, this approach makes more complex the development and the use of components.

SEA is an environment supporting development and use of reusable software artifacts. It employs a describing approach like those used in OOAD methodologies. The environment supports development, semantic checking and semi-automatic translation of design specifications. In this paper we describe how SEA supports the development and the use of flexible components.

1. Introdução

A meta de construir aplicações a partir da composição de artefatos de software tem mais de três décadas. Em 1968 durante a "crise do software", na Conferência de Engenharia de Software da OTAN, McIlroy fala da necessidade da indústria de software produzir famílias de componentes reusáveis. Segundo seu ponto de vista, desenvolvedores de software deveriam poder escolher componentes ajustados às suas necessidades específicas e estes componentes seriam usados em seus sistemas como artefatos caixa-preta [20].

Em 1976 DeRemer propõe um paradigma de desenvolvimento de software em que um sistema é desenvolvido como um conjunto de módulos produzidos separadamente e posteriormente interligados [6]. Este paradigma gerou trabalhos de pesquisa, como o Ambiente ADES, com uma linguagem específica para a produção de módulos e outra para compor aplicações através da interligação de módulos [8]. No entanto, a dependência do uso de linguagens e plataformas específicas, como era o caso do ambiente ADES, limitou a aplicabilidade desta abordagem.

Na década passada a abordagem de orientação a objetos pareceu o mecanismo adequado para promover reuso e chegou-se a considerar a classe como a unidade de reuso [21]. Verificou-se porém, que a reutilização não é característica inerente da orientação a objetos, mas deve ser obtida a partir do uso de técnicas que produzam software reutilizável [12] [15] [28].

A abordagem de desenvolvimento orientado a componentes, semelhante à proposta de DeRemer, determina que uma aplicação seja constituída a partir de um conjunto de módulos (componentes) interligados. Krajnc vê esta abordagem como uma evolução natural da abordagem de orientação a objetos, em que um componente corresponde a um conjunto de classes inter-relacionadas, com visibilidade externa limitada [15]. No evento WCOP 96 definiu-se componente como "uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas. Componentes podem ser duplicados e estar sujeitos a composições com terceiros" [33]. Esta visão foi refinada no WCOP 97: "o que torna alguma coisa um componente não é uma aplicação específica e nem uma tecnologia de implementação específica. Assim, qualquer dispositivo de software pode ser considerado um componente, desde que possua uma interface definida. Esta interface deve ser uma coleção de pontos de acesso a serviços, cada um com uma semântica estabelecida" [35].

Kozaczynski classifica a disponibilidade de mecanismos de interconexão de componentes como um dos principais estímulos à abordagem de desenvolvimento baseado em componentes [14]. CORBA é um destes mecanismos. Permite interconectar componentes independentemente da linguagem, plataforma de execução e localização física dos componentes. As unidades de conexão em CORBA são os objetos CORBA, que possuem uma interface pública definida através de uma linguagem de definição de interfaces (IDL). A disponibilidade de IDL's para a definição de interface para componentes desenvolvidos em diferentes linguagens de programação, como Java, C++ e Smalltalk, possibilita a interação entre componentes desenvolvidos em diferentes linguagens. O Object Request Broker (ORB) implementa o meio de comunicação entre componentes, tornando transparente a localização destes [31].

Além de CORBA, podem ser citados como padrões "*de fato*" de interconexão SOM, COM e JavaBeans. System Object Model (SOM) da IBM, baseado em CORBA e Component Object Model (COM) da Microsoft usam IDL's e possibilitam conexão com independência de linguagem, plataforma de execução e localização física dos componentes [27]. JavaBeans, da Sun Microsystems, em que os componentes são programados em Java, permite independência de plataforma de execução e localização física dos componentes [10]. Os mecanismos de interconexão apesar de solucionarem a questão da heterogeneidade entre elementos que precisam interagir, acarretam queda de desempenho, exigindo a avaliação deste aspecto para sua adoção [16].

Outros problemas associados a componentes estão relacionados à busca e seleção de componentes para reuso. A dificuldade de localização de componentes está associada à inexistência de padrões de repositório e mecanismos de busca que permitam a potenciais usuários selecionar componentes que supram suas necessidades [22]. A dificuldade de seleção está associada às deficiências apresentadas pelos mecanismos de descrição de componentes em especificar o que componentes fazem e como interagem.

Geralmente componentes necessitam ser adaptados antes de serem utilizados [3]. A dificuldade de compatibilizar componentes originalmente incompatíveis constitui outro obstáculo da abordagem de desenvolvimento baseado em componentes.

O presente trabalho trata os aspectos de descrição e adaptação de componentes. Inicialmente são apresentadas abordagens existentes de descrição e de adaptação de componentes. A seguir é proposta uma forma de descrever componentes, suportada pelo ambiente SEA, um ambiente de desenvolvimento e uso de artefatos de software reutilizáveis, atualmente em desenvolvimento [32]. Finalmente é mostrado como este ambiente, a partir da abordagem de frameworks, suporta o desenvolvimento e o uso de componentes flexíveis.

2. Abordagens de descrição da interface de componentes

A forma usual de descrever um componente consiste na descrição de sua interface. Porém, os mecanismos de descrição de interface existentes em geral, são pobres para descrever componentes porque produzem apenas uma visão externa incapaz de descrever suas funcionalidades e como estes interagem [11].

No caso geral, a interação de um componente com o meio externo pode ser bidirecional, ou seja, o componente pode receber invocações de métodos mas também pode originar invocações. Neste tipo de situação uma descrição de interface que apresente o conjunto de métodos fornecidos por um componente não é suficiente para entender como este interage. Ólafsson afirma que além da interface provida, uma descrição de interface de componente também deve estabelecer a interface requerida [24].

Murer estabelece que certas informações de interoperabilidade são comumente publicadas em documentação adicional, porque as linguagens correntes de descrição de interfaces não estão equipadas com a capacidade de descrever interoperabilidade de componentes. Ele propõe três níveis de informação de interoperabilidade [23]:

- Nível de interface: uso de linguagem semelhante a uma IDL para descrever como componentes podem ser agrupados, estruturalmente;
- Nível originador: que tipos e versões de componentes podem trabalhar juntos (restrições do fabricante);
- Nível semântico: uma descrição completa da funcionalidade do componente.

As propostas mais recentes ressaltam que a descrição de uma interface não pode ser feita exclusivamente através da relação de assinaturas de métodos fornecidos e requeridos, mas que também deve definir a dinâmica da interação. *Reuse contract* [18] é uma descrição de interface para um conjunto de componentes participantes que colaboram entre si. Estabelece os participantes que desempenham um papel em um *reuse contract*, suas interfaces, suas relações de conhecimento (componentes referenciados) e a estrutura de interação entre conhecidos. *Reuse contracts* documentam dependências interoperacionais entre um conjunto de componentes. A figura 1 apresenta um exemplo de *reuse contract*, usando notação gráfica.

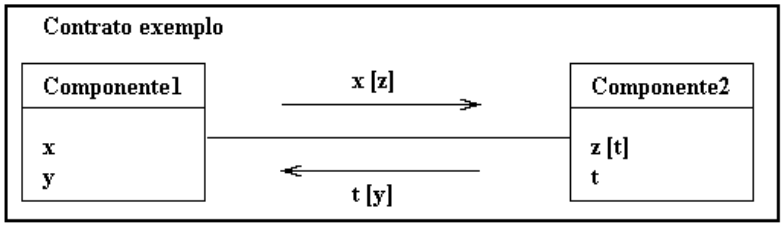

Figura 1 - exemplo de conexão de componentes especificada através de reuse contract

Um *reuse contract* é definido como um conjunto de descrições de participantes (no exemplo, Componente1 e Componente2), onde cada descrição de participante consiste de um nome único, uma cláusula de conhecimento, que estabelece o conjunto de componentes conhecidos de um participante (no caso do exemplo, a linha de conexão entre os dois componentes representa conhecimento mútuo entre eles) e uma descrição de interface. Uma descrição de interface é um conjunto de assinaturas de métodos consistindo de um nome único e uma cláusula de especialização. Uma cláusula de especialização é um conjunto de nomes de componentes conhecidos, cada um com um conjunto de nomes de operações associadas a eles (no exemplo, a execução do procedimento x de Componente1 pode acarretar na execução do procedimento z de Componente2).

Várias das propostas de descrição de interface de componentes carecem de formalismo, o que é necessário para descrever precisamente a interação entre componentes. Formalismos

algébricos têm sido usados para descrever protocolos de comunicação e sistemas distribuídos e segundo Canal, são adequados à abordagem de componentes por permitirem a avaliação de propriedades, como equivalência, inexistência de *deadlock* e outras. Canal propõe o uso de Lambda Cálculo para a especificação de arquiteturas de componentes. Uma desvantagem da proposta é que Lambda Cálculo é uma notação de baixo nível, o que dificulta sua aplicação direta a sistemas complexos. Uma solução para isto seria a incorporação de Lambda Cálculo a uma linguagem de descrição de arquiteturas de componentes [5].

De um modo geral, as propostas existentes de descrição de como componentes interagem são incompletas. Dos casos acima apresentados, Murer identifica a necessidade de descrição funcional "completa" dos componentes, mas não deixa claro como fazê-lo. Lucas se atém aos aspectos estrutural e comportamental dos componentes e propõe um modelo com notação gráfica que facilita o entendimento da interação, porém sua informalidade acarreta deficiências, como a impossibilidade de verificação de propriedades como *deadlock*, o que é crítico. Canal, que se atém ao aspecto comportamental da conexão de componentes, propõe o uso de Álgebra. A notação de baixo nível usada é reconhecida pelo proponente como um obstáculo à adoção da abordagem. A seguir é discutida a necessidade de descrição funcional para possibilitar o reuso de componentes.

3. A necessidade de descrever a funcionalidade de componentes

Dois componentes a serem interligados são estruturalmente compatíveis se o conjunto de métodos invocados através da interface de um está disponível na da interface do outro. A conexão destes componentes porém, pode produzir resultados imprevistos. Seja o exemplo da figura 2, usando notação de *Reuse Contracts*, com dois componentes estruturalmente compatíveis interligados.

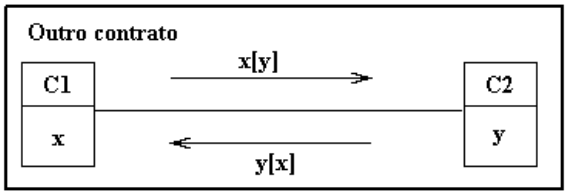

Figura 2 - exemplo de conexão de componentes especificada através de reuse contract

Segundo a abordagem *Reuse Contracts* esta seria uma conexão válida, pois:

- durante a execução de x, C1 pode invocar y de C2 e C2 dispõe deste método e
- durante a execução de y, C2 pode invocar x de C1 e C1 dispõe deste método.

Porém, se a implementação destes métodos nos componentes for tal que C1 aguarde a invocação de x antes de invocar y e C2 aguarde a invocação de y antes de invocar x, estará caracterizado um *deadlock*. Assim, estes componentes seriam estruturalmente compatíveis mas comportamentalmente incompatíveis.

Garantir que dois componentes sejam estrutural e comportamentalmente compatíveis porém, também não assegura operação conjunta sem imprevistos. O caso da explosão da aeronave no vôo 501 do projeto Ariane-5 da Agência Espacial Européia (ESA), segundos após o lançamento, ilustra esta situação.

Segundo o relatório da comissão que analisou o ocorrido [17], foi verificado que:

- um componente de software do sistema de referência inercial da aeronave ficou inoperante (primeiro do sistema *back-up* e em seguida do sistema ativo) em função de uma variável associada à velocidade horizontal ter assumido valor acima do limite previsto;
- estava previsto que o referido sistema de referência inercial seria testado através de simulação, porém, considerou-se este procedimento desnecessário, uma vez que o sistema (*hardware* e *software*) havia sido usado com êxito no projeto Ariane-4;

- tal decisão foi a causa do problema, pois desconsiderou o fato de Ariane-5 assumir uma velocidade horizontal cinco vezes superior à velocidade de Ariane-4 nos segundos posteriores ao lançamento, causa do valor elevado assumido pela variável citada.

O componente reusado era estrutural e comportamentalmente compatível com o sistema a que foi conectado, porém o desconhecimento de aspectos fundamentais da operação deste componente levou a uma situação de reuso indevido, gerando danos irreparáveis. Com isto, verifica-se que além de compreender a estrutura e o comportamento da interface de um componente, é necessário compreender exatamente o que o componente faz, como opera, antes de julgá-lo adequado ao reuso.

4. Abordagens de adaptação de componentes

Dificilmente componentes são reusáveis tal qual foram desenvolvidos. Normalmente precisam ser adaptados para se moldarem aos requisitos do sistema a que serão acoplados. Duas abordagens têm sido usadas para adaptar um componente: alteração ou empacotamento (*wrapping*) do componente [3]. O empacotamento, ao invés de modificar o componente, cria uma visão externa diferente para ele. Outra alternativa para interconectar componentes originalmente incompatíveis consiste em criar um componente intermediário que intermedie a comunicação. Este componente intermediário é genericamente chamado cola (*glue*) [35].

Um problema observado na colagem de componentes é que normalmente a cola está escondida em uma estrutura de programação complexa, que torna obscuro seu propósito. Um melhor entendimento da semântica da colagem de componentes pode ser obtido pelo uso de abstrações para modelar a cola no nível de projeto. Alencar propõe um modelo para especificar colagem de componentes baseado em pontos de vista (*viewpoints*). O relacionamento de cola é isolado e definido separadamente dos componentes [1]. A definição da cola consiste na criação de objetos ponto de vista, que são observadores de componentes existentes, sendo que:

- mais de um ponto de vista pode ser associado a um componente;
- o estado de um ponto de vista se altera quando ocorre a alteração do estado do componente observado;
- um ponto de vista atua como interface entre o componente e o meio com que ele interage, permitindo alteração e adaptação da interface original, bem como extensão de funcionalidades
- pontos de vista podem ser criados, destruídos, acoplados ou desacoplados dinamicamente.

Assmann formulou uma proposta, em que o uso de componentes é baseado em visões abstratas destes (semelhante à abordagem de Alencar). Para desacoplar usos de componentes de suas definições, componentes não fazem referência direta a outros componentes, mas a visões destes. Software é construído por composição. Operadores de composição são usados para combinar múltiplas visões de componentes. O modelo proposto é chamado *software cocktail mixer*. Para construir componentes complexos a partir de componentes mais simples, o modelo usa operadores de composição de definição. Este operador combina duas definições de componentes e produz uma nova [2].

A abordagem de alteração de componentes pode ser uma tarefa árdua, caso não haja uma descrição do componente que permita entender o que deve ser alterado para adaptar o componente. Caso somente se disponha do código fonte, é preciso proceder análise de código para entender o componente e descobrir o que deve ser modificado. Isto corresponde a uma atividade de Engenharia Reversa, em que é procedida uma recuperação de projeto para

possibilitar uma posterior modificação do software original. Em algumas situações esta atividade exige tanto esforço, que pode tornar a adaptação de um componente impraticável.

A alteração de um componente pode se tornar uma tarefa menos árdua se o componente tiver sido desenvolvido para ser alterado. Este é o caso dos arcabouços de componente (*component frameworks*). Arcabouço de componente é um componente que prevê o acoplamento de outros componentes e que é modificado a partir da troca dos componentes a ele acoplados [37] [10].

5. Abordagem proposta de descrição de componentes

A abordagem de descrição de componentes aqui apresentada é utilizada no ambiente SEA, em que o paradigma de orientação a objetos foi adotado para desenvolver frameworks, componentes e aplicações. Assim, componentes são definidos como estruturas de classes. O que caracteriza uma estrutura de classes de componente é que esta reutiliza uma interface, selecionada em uma biblioteca de interfaces de componentes. Uma interface aparece na estrutura do componente como uma das classes, porém possui uma especificação individual que descreve sua estrutura e sua dinâmica comportamental.

A especificação de interfaces em separado da especificação de componentes permite que uma mesma especificação de interface possa ser reusada em vários componentes, produzindo assim uma família de componentes estrutural e comportamentalmente compatíveis.

5.1. Especificação estrutural de interface de componente

Um componente possui uma única interface, que pode prever a conexão a mais de um componente. Cada conexão utiliza um canal de conexão. Uma arquitetura de componentes é produzida a partir da conexão de seus canais, conforme ilustrado na figura 3.

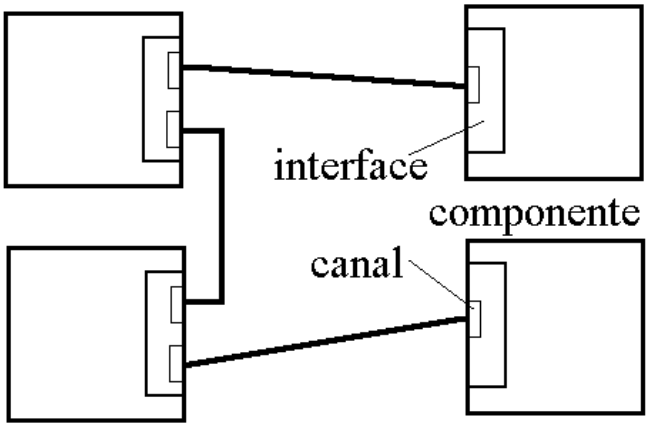

Figura 3 - arquitetura de componentes produzida a partir da conexão de um conjunto de componentes através de seus canais de conexão

A especificação estrutural de uma interface de componente relaciona os métodos fornecidos, os métodos requeridos e a associação destes a cada canal da interface. A figura 4 exemplifica uma situação hipotética de uma interface com três canais, cinco métodos requeridos e cinco métodos fornecidos. Nem todos os métodos fornecidos precisam estar acessíveis em todos os canais (a marca na tabela define a acessibilidade). O mesmo ocorre com os métodos requeridos, possibilitando que a responsabilidade de implementar os métodos requeridos por um componente esteja distribuída entre um conjunto de componentes estrutural, comportamental e funcionalmente diferentes.

A estrutura de uma interface no ambiente SEA é produzida relacionando assinaturas de métodos e canais, e definindo o relacionamento entre estes, conforme o exemplo da figura 4. O ambiente embute esta estrutura nos atributos da classe que implementa a interface.

		Métodos requeridos					Métodos fornecidos				
		mA	**mB**	**mC**	**mD**	**mE**	**mF**	**mG**	**mH**	**mI**	**mJ**
Canais	**c1**	✓		✓	✓		✓	✓	✓		
	c2	✓	✓				✓			✓	✓
	c3		✓		✓	✓	✓		✓	✓	✓

Figura 4 - estrutura de relacionamento de canais e métodos de uma especificação hipotética de interface de componente

5.2. Especificação comportamental de interface de componente

A questão a ser tratada na descrição comportamental da interface de um componente é se há ou não restrições associadas à ordem de invocação de métodos. A inexistência de restrições significa que qualquer método fornecido ou requerido, pode ser invocado a qualquer instante do tempo de existência de um componente. Se existem porém restrições, como a necessidade de invocar determinado método antes de outro, estas devem fazer parte da modelagem da interface. Nas propostas de mecanismos de descrição comportamental anteriormente apresentadas observam-se duas tendências opostas. De um lado mecanismos como *reuse contracts* [18], de fácil compreensão, porém com expressividade limitada, conforme ilustrado no exemplo da figura 2. De outro, mecanismos formais, como a proposta de descrição do comportamento da interface através de Lambda Cálculo [5], porém de difícil compreensão.

Isto ilustra um dilema na escolha da técnica de modelagem adequada à modelagem comportamental de interfaces. Uma técnica de modelagem formal permite produzir modelos sem ambigüidades e que podem ser validados formalmente. A questão da compreensibilidade se reflete na maior ou menor facilidade de entendimento dos modelos produzidos. Os dois aspectos são importantes, porém em geral são antagônicos [7].

Considerando a adequação de uma técnica de modelagem gráfica em relação ao aspecto da compreensibilidade, e a necessidade de formalismo que permita avaliar uma composição de componentes em relação a um conjunto de propriedades, adotou-se o uso de rede de Petri [26] para a modelagem comportamental da interface de componentes. Este modelo é baseado em formalismo algébrico, o que permite a análise de propriedades de uma composição de componentes. Também possui uma notação gráfica correspondente, que o torna de mais fácil compreensão que outros formalismos.

No ambiente SEA está sendo usada a rede de Petri ordinária, que é composta por lugares, transições, arcos que interligam lugares e transições, e uma marcação inicial, caracterizada por uma quantidade de fichas em cada lugar da rede. A única extensão em relação à rede de Petri ordinária é a necessidade de associar cada par *(canal, método)* definido na estrutura da interface a uma ou mais transições da rede. A figura 6 ilustra a descrição comportamental de uma interface hipotética usando este modelo. A figura 5 contém a descrição da estrutura desta interface.

		Métodos requeridos		Métodos fornecidos
		x	**y**	**m**
Canais	**Ca**	✓	✓	✓
	Cb	✓		✓

Figura 5 - estrutura de canais e métodos de uma interface de componente

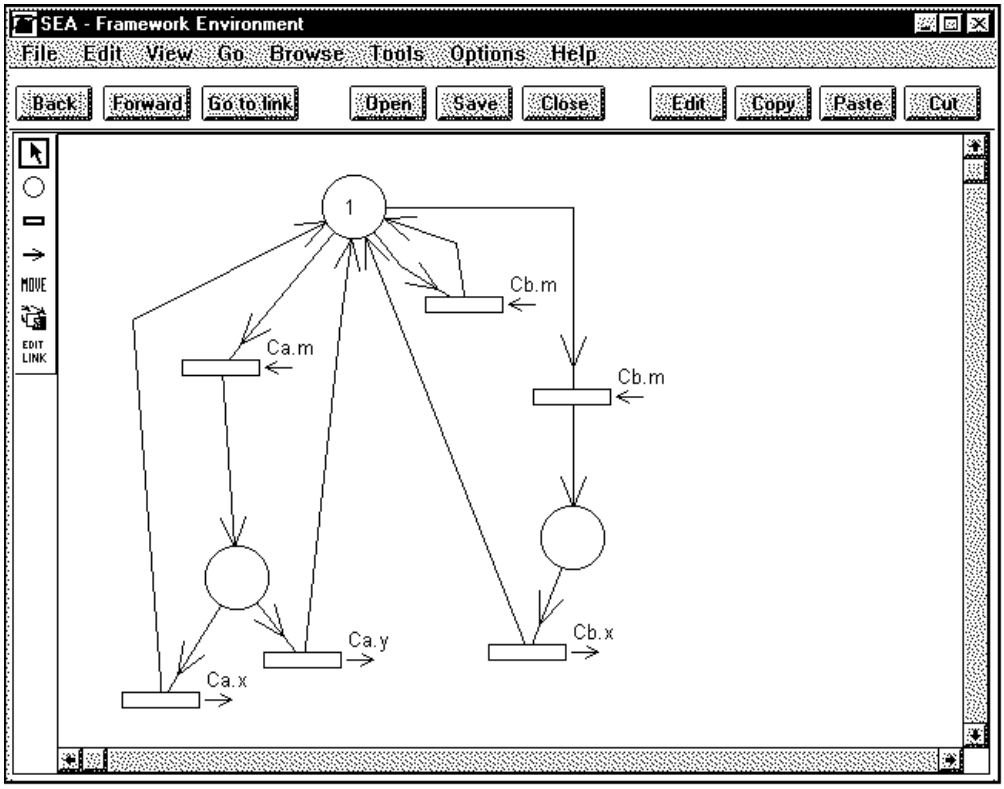

Figura 6 - descrição comportamental de interface de componente usando rede de Petri gerada no ambiente SEA

O comportamento de um conjunto de componentes interligados é obtido a partir da união das redes de Petri que descrevem suas interfaces. No processo de união das redes, lugares, arcos e marcação inicial das redes originais são mantidos, e pares de transições correspondentes são fundidos em uma transição. Duas transições de redes diferentes são correspondentes se estiverem associadas ao mesmo método (requerido por uma interface e fornecido pela outra) e a canais interligados na composição da arquitetura de componentes. A figura 7 ilustra a rede que descreve a interligação de componentes descrita no exemplo da figura 2. As duas redes de Petri à esquerda descrevem a interface dos dois componentes e a rede à direita descreve o comportamento da arquitetura resultante da conexão destes. Os componentes estão interligados a partir dos canais únicos de suas interfaces (Ca e Cb). Observe-se que a marcação inicial da rede resultante não habilita o disparo de nenhuma transição, caracterizando um *deadlock*, como descrito textualmente no exemplo da figura 2.

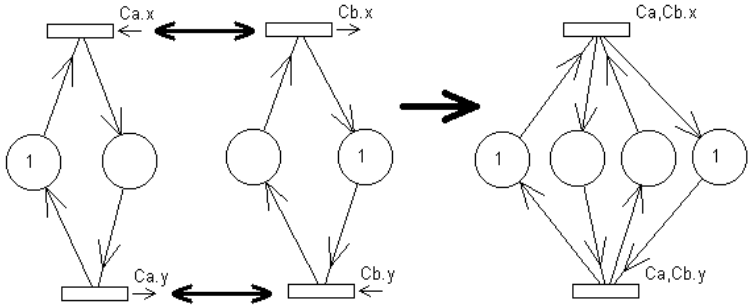

Figura 7 - descrição comportamental de uma arquitetura de componentes

Quando um par *(canal, método)* está associado a N transições em uma das redes, e o par *(canal, método)* correspondente da outra rede (mesmo método, canais interligados) está associado a uma transição, a rede resultante apresentará N transições associadas a estes dois pares. A figura 8 ilustra a situação em que em uma das redes o par correspondente ao método x está associado a uma transição e na outra a duas. A rede resultante apresenta duas transições para x. No caso geral, em que em uma das redes um par *(canal, método)* está associado a N transições e na outra o par correspondente está associado a M transições, a rede resultante apresentará NxM transições associadas a estes pares.

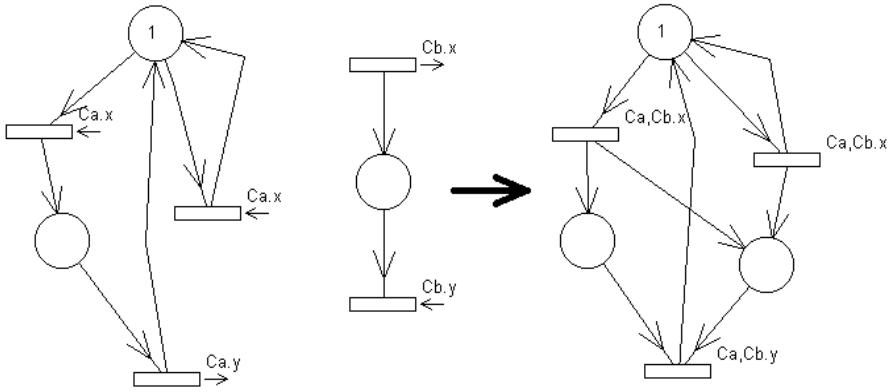

Figura 8 - descrição comportamental de uma arquitetura de componentes

Os lugares da rede de Petri representam pré ou pós-condições das transições a que estão interligados. Assim, uma transição sem pré-condição (que representa um método sem restrições para ser invocado) não possui lugar de entrada. De modo análogo, uma transição que não gera pós-condição não possui lugar de saída. O exemplo da figura 8 apresenta transições com estas características.

A adoção de redes de Petri para descrever o comportamento de interfaces de componentes, além da vantagem da compreensibilidade, permite analisar propriedades de interfaces individuais e arquiteturas de componentes, como a possibilidade de avaliar a existência de *deadlock* ou a possibilidade de verificar se existem transições que nunca são disparadas (ou seja, métodos que nunca são invocados), a partir da análise de invariantes.

6. Aplicação da abordagem de frameworks para a produção de componentes flexíveis

As abordagens de empacotamento e colagem tratam a adaptação de componentes sem alterar componentes existentes. Cria-se uma estrutura de software adicional que mascara um componente existente no primeiro caso, e intermedia-se a interligação de dois componentes no segundo. A abordagem arcabouço de componente propõe a criação de um componente que pode ser alterado a partir da troca de outros componentes a ele acoplados. Em todos os casos não ocorre alteração dos componentes existentes, no sentido de modificar seu código. A modificação interna é evitada basicamente pela complexidade envolvida. Por outro lado, se um componente for projetado prevendo futuras modificações, e se houver suporte para modificá-lo, é possível diminuir a complexidade da alteração da estrutura de componentes, viabilizando a criação de componentes flexíveis. Visando este objetivo, adotou-se a abordagem de frameworks orientados a objetos para o desenvolvimento de componentes flexíveis.

Um framework orientado a objetos é uma estrutura de classes inter-relacionadas, que constitui uma implementação inacabada, para um domínio de aplicações. A geração de

artefatos de software acabados a partir de um framework, consiste na adaptação desta estrutura a necessidades específicas [13]. A figura 9 ilustra uma aplicação desenvolvida a partir de um framework. A parte sombreada corresponde ao framework - uma estrutura de classes que é reutilizada - e parte não-sombreada corresponde ao que é produzido por um usuário do framework.

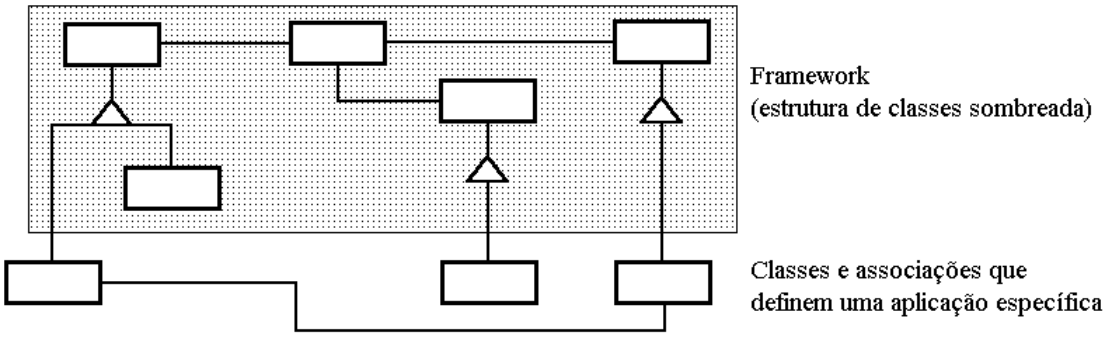

Figura 9 - aplicação desenvolvida utilizando um framework

O principal benefício dos frameworks é a reutilização de projeto: um framework tem definida a estrutura de projeto de componentes ou aplicações a serem desenvolvidos a partir dele. Além disto, há o reuso do código do framework, sendo necessário conhecer apenas parte das classes reutilizadas [36]. A abordagem apresenta dois tipos de desvantagem: complexidade para desenvolver e complexidade para usar. Complexidade de desenvolvimento está relacionada à dificuldade de produzir uma estrutura de classes que descreva um domínio de aplicações e seja dotada de generalidade, flexibilidade e extensibilidade. Complexidade de uso diz respeito ao esforço requerido para um usuário aprender a desenvolver software a partir de um framework [32].

Um framework desenvolvido para produzir componentes corresponde a uma estrutura de classes que apresenta partes mantidas flexíveis (*hot spots* [28]), para possibilitar sua adaptação a diferentes requisitos. Um framework pode corresponder a uma implementação inacabada de componente ou pode conter uma implementação *default*. Neste caso haveria a disponibilidade de um componente sem necessidade de adaptação da estrutura do framework. Nos dois casos a estrutura deve ser adaptada para a obtenção de componentes específicos. Um componente definido como um framework pode conter ou não a definição da interface a ser utilizada. Assim, pode-se obter flexibilidade tanto no aspecto funcional, quanto na definição da interface.

A vantagem da adoção desta abordagem está na possibilidade de obter um conjunto de componentes distintos a partir de um esforço menor que o necessário para desenvolver cada um isoladamente. Isto é possibilitado pelo reuso de projeto e código promovido pela abordagem de frameworks. As dificuldades citadas do desenvolvimento e uso de frameworks podem ser atenuadas por um suporte de desenvolvimento e uso de frameworks, como o suporte fornecido pelo ambiente SEA.

Com a diminuição do esforço necessário para produzir novos componentes promovida pela abordagem dos frameworks, considerando a disponibilidade de suporte ao manuseio de frameworks, altera-se o procedimento de obtenção de um componente ligeiramente diferente de um componente existente (desenvolvido sob um framework). Ao invés de alterar o componente existente, cria-se um novo componente, reutilizando o framework e até mesmo reutilizando parte da especificação de projeto de componentes existentes.

7. Suporte ao desenvolvimento e uso de componentes flexíveis no ambiente SEA

O ambiente SEA, atualmente em construção, dispõe de funcionalidades voltadas a facilitar o desenvolvimento de componentes flexíveis (definidos como frameworks), bem como para a produção de novos componentes a partir de sua alteração.

7.1. SEA, um ambiente para o desenvolvimento e uso de artefatos reutilizáveis de software

SEA é um ambiente que suporta o desenvolvimento e o uso de artefatos reutilizáveis de software (frameworks e componentes). Está sendo desenvolvido considerando os seguintes requisitos.

Suporte à edição gráfica de modelos - o uso de técnicas de modelagem gráficas é capaz de facilitar a compreensão de especificações de software [7], ajudando a tornar o desenvolvimento e uso de frameworks e componentes uma tarefa menos árdua. No ambiente SEA, frameworks, aplicações, componentes, interfaces de componentes e arquiteturas de componentes são especificados como conjuntos de modelos gráficos.

Suporte à edição semântica - a geração de código exige que um ambiente assegure consistência semântica das especificações, ao invés de simplesmente atuar na edição gráfica. As especificações geradas no ambiente SEA podem ser submetidas a verificação de consistência e traduzidas. São adotados os seguintes meios para garantir consistência semântica de especificações de projeto:

- é definido um metamodelo para cada tipo de especificação[1], que estabelece os elementos da especificação (os modelos e os elementos conceituais que os compõem) e as relações semânticas entre eles. Cada metamodelo é implementado na estrutura de classes do ambiente;
- cada editor do ambiente está sujeito às restrições do metamodelo associado.

O ambiente SEA dispõe de um conjunto de funcionalidades de suporte à edição semântica, como:

- ações de refatoração - por exemplo, mover atributos e métodos de uma classe para outra, fundir classes, mudar a ordem, a origem ou o destinatário de mensagens etc.;
- propagação de efeito das ações de edição - a remoção de um classe de uma especificação, por exemplo, produz - entre outros efeitos - a remoção da referência a ela mantida por objetos de diagramas de seqüência, automaticamente.

Suporte ao reuso de artefatos de software - é possível aumentar a produtividade de desenvolvimento por meio do reuso de software. O ambiente SEA deve suportar diferentes abordagens de reuso: reuso de frameworks, de componentes, de interfaces de componentes, de estruturas de projeto pré-elaboradas e mantidas em biblioteca (como *design patterns* [9]), importação de artefatos de software externos ao ambiente por meio de procedimentos de Engenharia Reversa[2].

Flexibilidade - flexibilidade é requerida para possibilitar a evolução de um ambiente, isto é, incluir novas ferramentas e novas funcionalidades com o menor esforço possível, e sem efeitos

---

[1] SEA atualmente trata quatro tipos de especificação: um para a descrição de aplicações, frameworks e componentes, um outro para interfaces de componentes, um terceiro para arquiteturas de componentes e um quarto para hiperdocumentos.

[2] Atualmente o ambiente SEA ainda não dispõe de mecanismos para suportar procedimentos de Engenharia Reversa.

colaterais inesperados sobre a estrutura do ambiente. O ambiente SEA apresenta flexibilidade em diversos aspectos. Permite, por exemplo:

- modificar os tipos de modelo que definem uma especificação (alteração de metamodelo);
- incluir ou alterar ferramentas que atuam sobre especificações;
- usar diferentes mecanismos de armazenamento (SGBD, arquivos etc.);
- alterar as regras de permissão de acesso a especificações

Para obter um ambiente capaz de suportar os requisitos acima, SEA está sendo construído baseado em uma arquitetura de framework. SEA reusa os frameworks MVC [25] e HotDraw [4].

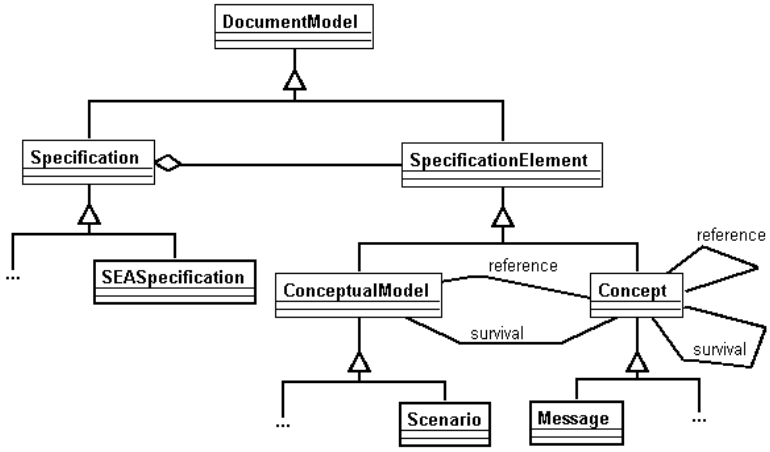

Figura 10 - estrutura de metamodelo suportada pelo ambiente SEA

A figura 10 apresenta a estrutura de metamodelo suportada pelo ambiente. Especificações são compostas por elementos de especificação, que podem ser elementos conceituais, como classe, mensagem (instâncias de subclasses de *Concept*) ou modelos, como diagrama de classes, diagrama de seqüência (instâncias de subclasses de *ConceptualModel*). Pode haver relacionamentos de *referência* ou de *subsistência* entre elementos conceituais ou entre um modelo e um elemento conceitual. Um relacionamento de *referência* entre elementos de especificação denota que parte da definição de um elemento de especificação está apoiada na referência a outro elemento, como por exemplo a referência de uma mensagem (de um diagrama de seqüência) a um método. Um relacionamento de *subsistência* entre elementos de especificação denota que a existência de um elemento de especificação está condicionada à existência de outro elemento, como por exemplo a existência de um método condicionada à existência de sua classe.

Todos os elementos conceituais referenciados por modelos são mantidos em um repositório único. Uma classe que aparece simultaneamente em mais de um diagrama de classes ou em outros diagramas, por exemplo, é um mesmo elemento conceitual. Modelos possuem representação gráfica (definidas como subclasses de *Drawing*, de *HotDraw*). De modo análogo, elementos conceituais são associados a figuras (definidas como subclasses de *Figure*, de *HotDraw*), presentes nos diagramas dos modelos.

7.2. A estrutura de uma especificação de componente

Especificações de projeto de componentes, frameworks e aplicações são desenvolvidas no ambiente SEA a partir do uso de técnicas de modelagem gráficas. Utilizam-se cinco das oito técnicas de modelagem propostas em UML [30] e uma técnica adicional (baseada no diagrama de ação [19]) para descrever o corpo dos métodos, o que não é previsto em UML. SEA utiliza as seguintes técnicas:

- diagrama de casos de uso,

- diagrama de atividades,
- diagrama de classes,
- diagrama de seqüência,
- diagrama de estados e
- diagrama de corpo de método (este não previsto em UML).

Nem todos os elementos sintáticos previstos nas técnicas de UML foram usados nos editores do ambiente SEA, resultando em uma diminuição da expressividade original. Por outro lado, foram introduzidas extensões para:

- representar conceitos do domínio de frameworks, não representáveis nas técnicas de UML, como redefinibilidade de classe;
- suprir lacunas semânticas existentes em UML, como restrições na ordem dos casos de uso e semântica associada aos estados;
- possibilitar a descrição do corpo dos métodos na especificação de projeto;
- expressar a ligação semântica entre elementos de uma especificação, como um caso de uso e um diagrama de seqüência que o refine, e entre especificações distintas, como entre a especificação de um framework e a especificação de um componente desenvolvido sob este framework;
- possibilitar que especificações sejam tratadas como hiperdocumentos, onde os elementos de uma especificação podem ter elos associados que apontem outros elementos de especificação (inclusive de outras especificações).

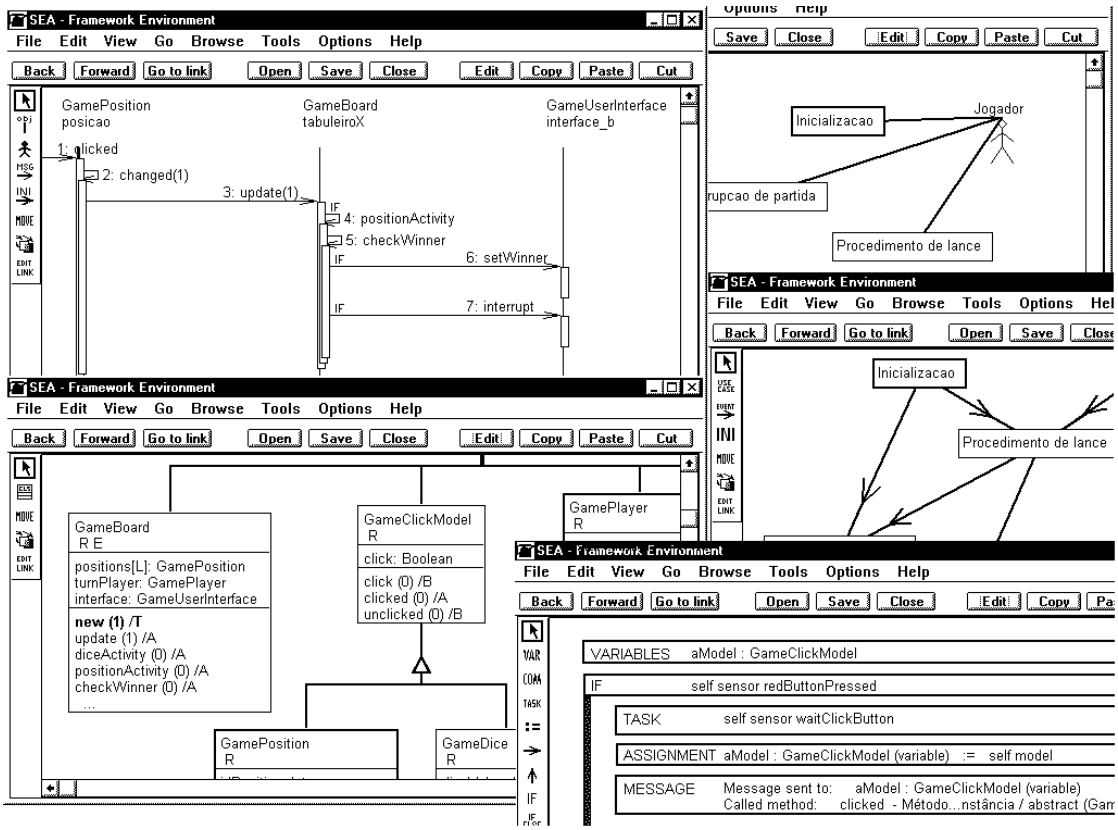

Figura 11 - editores do ambiente SEA

O conjunto de técnicas proposto é voltado à documentação de projeto. A documentação fornecida ao usuário de um componente pode incluir este mesmo conjunto de técnicas de modelagem, porém contendo apenas uma parte da especificação, ou seja, apenas as

informações que um usuário precisa para entender o que o componente faz[3]. Esta especificação que descreve como o componente funciona, juntamente com a especificação da interface, produz uma descrição de componente que cobre os três aspectos a especificar: funcionalidade do componente, estrutura e comportamento da interface. A figura 11 apresenta alguns editores do ambiente SEA.

7.3. O desenvolvimento e uso de componentes flexíveis no ambiente SEA

O ambiente SEA dispõe do conjunto de editores que possibilita especificar componentes como estruturas de classes. Possibilita duas alternativas: uma estrutura de classes acabada ou uma estrutura de classes flexível, ou seja, um framework.

SEA dispõe de suporte à adaptação de frameworks. O mecanismo é baseado na abordagem de cookbooks ativos [29] e orienta o desenvolvimento de especificações de componentes e aplicações baseados em frameworks, através de um hiperdocumento, que estabelece os passos a serem seguidos. Diferentes caminhos podem ser seguidos, de acordo com as necessidades específicas da aplicação ou componente sob desenvolvimento. O hiperdocumento que dirige o desenvolvimento, além de elos convencionais que suportam a navegação através de suas páginas e de especificações de projeto pré-existentes, possui elos ativos que automatizam ações do processo de desenvolvimento de especificações de projeto. O ambiente SEA suporta o desenvolvimento de hiperdocumentos de orientação ao uso de frameworks, bem como o seu manuseio, para a produção de aplicações ou componentes.

Além do desenvolvimento da estrutura de componentes, SEA suporta a especificação de interfaces de componentes, conforme descrito, e de arquiteturas de componentes. Arquiteturas de componentes são obtidas instanciando componentes previamente especificados e interligando os canais correspondentes de suas interfaces. SEA verifica a compatibilidade estrutural dos canais no procedimento de conexão. O ambiente ainda não dispõe de mecanismo de análise de rede de Petri que permita verificar a compatibilidade comportamental.

SEA suporta verificação de consistência de especificações de projeto e geração semi-automática de código. Atualmente especificações podem ser traduzidas em código Smalltalk.

8. Conclusão

Este trabalho apresentou a abordagem adotada no ambiente SEA para o desenvolvimento e uso de componentes. Foi proposta uma estrutura de descrição capaz de especificar componentes como estruturas de classes e capaz de especificar estrutural e comportamentalmente interfaces de componentes. Interfaces possuem especificações distintas das especificações de componentes, o que facilita seu reuso, possibilitando criar diferentes componentes com uma mesma especificação de interface. Arquiteturas de componentes são produzidas instanciando componentes existentes e interligando canais de interface estruturalmente compatíveis.

Foi apresentado o uso da abordagem de frameworks para o desenvolvimento de componentes. Neste caso um framework corresponderia a um componente genérico, alterável para a produção de componentes diferentes. A vantagem desta abordagem é que a reutilização de projeto e código promovida pelos frameworks diminui o esforço necessário para a

---

[3] Há uma razão de ordem prática para que a documentação fornecida ao usuário não contenha toda a especificação de projeto: em termos de produção industrial de software, é inviável esperar que os produtores de software forneçam toda a documentação de projeto de um produto [34].

produção de novos componentes. Isto viabiliza uma alternativa diante da necessidade de adaptação de componentes: ao invés de alterar ou mascarar um componente existente, pode-se produzir um novo componente.

No ambiente SEA componentes e arquiteturas de componentes (assim como frameworks e aplicações) são produzidos como especificações de projeto que são submetidas à verificação de consistência e traduzidas para linguagem de programação. A notação de projeto usa técnicas de modelagem empregadas por UML, com adaptações. O ambiente SEA opera atualmente dispondo de edição e verificação de consistência de modelos, suporte à criação e manuseio de hiperdocumentos que dirigem a aplicação de frameworks, suporte ao reuso de *design patterns* e facilidades de edição semântica. O ambiente está sendo testado e refinado com pequenos exemplos. O mecanismo de descrição de arquiteturas de componentes, que deverá suportar geração automática da rede de Petri resultante da conexão de componentes, ainda não está concluído. Além de análise de redes de Petri, outras funcionalidades previstas para o ambiente SEA, ainda precisam ser incluídas ou completadas. Nos esforços futuros prevê-se viabilizar conexão dinâmica de componentes

O ambiente SEA está sendo desenvolvido com o objetivo de suportar e facilitar o desenvolvimento e uso de artefatos reutilizáveis de software, atividades inerentemente complexas. O suporte ora disponível traz soluções para alguns dos problemas identificados na abordagem de desenvolvimento orientado a componentes, facilitando o desenvolvimento de componentes e arquiteturas de componentes.

9. Referências

[1] ALENCAR, P.S.C. *et al*. **A model for gluing components**. In: Third International Workshop on Component-Oriented Programming (WCOP'98), 1998, Brussels.

[2] ASSMANN, U., SCHMIDT, R. **Towards a model for composed extensible components**. In: Foundations of Component-Based Systems Workshop, 1997, Zurich.

[3] BOSCH, J. **Adapting object-oriented components.** In: Second International Workshop on Component-Oriented Programming (WCOP'97), 1997, Jyväskylä.

[4] BRANT, J. **HotDraw**. Urbana: University of Illinois at Urbana-Champaign, 1995. *Master thesis*.

[5] CANAL, C. *et al*. **On the composition and extension of software components**. In: Foundations of Component-Based Systems Workshop, 1997, Zurich.

[6] DEREMER, F., KRON, H.H. **Programming-in-the-large versus programming-in-the-small**. IEEE Transactions on Software Engineering. v.se-2, n.2. 1976.

[7] FOWLER, M. **Describing and comparing object-oriented analysis and design methods**. In: Carmichael. Object development methods. New York: SIGS Books, 1994. p. 79-109.

[8] FRAGA, J.S. *et al*. **A software environment for distributed applications**. In: 15º IFAC/IFIC, 1988, Valencia.

[9] GAMMA, E. **Design patterns**: elements of reusable object-oriented software. Reading: Addison-Wesley, 1994.

[10] HELTON, D. **The impact of large-scale component and framework application development on business**. In: Third International Workshop on Component-Oriented Programming (WCOP'98), 1998, Brussels.

[11] HONDT, K. *et al*. **Reuse contracts as component interface descriptions**. In: Second International Workshop on Component-Oriented Programming (WCOP'97), 1997, Jyväskylä.

[12] JOHNSON, R. E., FOOTE, B. **Designing reusable classes**. Journal of object-oriented programming. p. 22-35, jun./jul. 1988.

[13] JOHNSON, R. E. **Documenting frameworks using patterns**. In: Object-Oriented Programming Systems, Languages and Applications Conference - OOPSLA, 1992, Vancouver.
[14] KOZACZYNSKI, W., BOOCH, G. **Component-based software engineering**. IEEE Software. sep. 1998.
[15] KRAJNC, M. **What is component-oriented programming?** Oberon Microsystems, 1997, Zurich.
[16] KRAJNC, M. **Why component-oriented programming?** Oberon Microsystems, 1997, Zurich.
[17] LIONS, J. L., **ARIANE 5 failure** - full report. Paris: ESA, 1996 (disponível em http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html)
[18] LUCAS, C. **Documenting reuse and evolution with reuse contracts**. Vrije Universiteit Brussel, 1997. *PhD thesis*.
[19] MARTIN, J. **Técnicas estruturadas e CASE**. São Paulo: Makron Books, McGraw-Hill, 1991.
[20] MCILROY, M. D. **Mass-produced software components**. In North Atlantic Treaty Organization Conference on Software Engineering, 1968, Garmisch-Partenkirchen.
[21] MEYER, B. **Object-oriented software construction**. Englewood Cliffs: Prentice Hall, 1988.
[22] MILI, R. *et al*. **Storing and retrieving software components: a refinement based system**. IEEE Transactions on Software Engineering. v.23, n.7. july 1997.
[23] MURER, T. *et al*. **Improving component interoperability**. In Special issues in object-oriented programming, Workshop reader of the ECOOP'96, 1996, Linz.
[24] ÓLAFSSON, A., DOUG, B. **On the need for "required interfaces" of components**. In Special issues in object-oriented programming, Workshop reader of the ECOOP'96, 1996, Linz.
[25] PARCPLACE. **VisualWorks Cookbook**. ParcPlace Systems Inc.1994.
[26] PETRI, C. A., **Kommunikation mit Automaten**. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962
[27] PFISTER, C. **Object models: SOM and COM**. In: European Conference on Object-Oriented Programming - ECOOP'96, 1996, Linz. Tutorial Notes.
[28] PREE, W. **Design patterns for object oriented software development**. Reading: Addison-Wesley, 1995
[29] PREE, W. *et al*. **Active guidance of framework development**. Software - Concepts and tools v.16, n.3. 1995.
[30] RATIONAL. **UML notation guide**. Rational Software Corporation, 1997. (www.rational.com, uml/ad970805_uml11_Notation2.zip).
[31] SEETHARAMAN, K. **The CORBA connection**. Communications of the ACM. v.33, n.9. sep. 1990.
[32] SILVA, R. P., PRICE, R. T. **Tool support for helping the use of frameworks**. In: XVIII International Conference of the Chilean Computer Science Society (SCCC'98), Nov. 1998, Antofagasta.
[33] SZYPERSKI, C. *et al*. **Sumary of the First International Workshop on Component-Oriented Programming.** In: First International Workshop on Component-Oriented Programming (WCOP'96), 1996, Linz.
[34] SZYPERSKI, C. **Component-oriented programming: a refined variation on object-oriented programming**. In: European Conference on Object-Oriented Programming - ECOOP'96, 1996, Linz. Tutorial Notes.
[35] SZYPERSKI, C. *et al*. **Sumary of the Second International Workshop on Component-Oriented Programming.** In: Second International Workshop on Component-Oriented Programming (WCOP'97), 1997, Jyväskylä.
[36] TALIGENT. **Leveraging object-oriented frameworks**. Taligent Inc. white paper, 1995.
[37] WECK, W. **Independently extensible component frameworks**. In: First International Workshop on Component-Oriented Programming (WCOP'96), 1996, Linz.