# 2
# Conceitos Básicos

**Resumo**

*Esta seção define os conceitos básicos utilizados neste trabalho. Em um primeiro momento, são apresentadas definições sobre componentes de software. Em seguida, questões de composição e coordenação de componentes são discutidas. Por fim, uma conceituação sobre arquiteturas de software é elaborada, incluindo a apresentação do padrão arquitetural Layer (Buschman, 1996).*

## 2.1.
## Componentes de Software

O desenvolvimento de software baseado em componentes é uma abordagem utilizada para estruturar soluções, organizar o desenvolvimento de aplicações e promover a convivência entre tecnologias (Heineman & Councill, 2001). Especificar e projetar unidades de software altamente coesas e fracamente acopladas pode não ser uma tarefa trivial. Mas lidar adequadamente com estas questões pode reduzir, e muito, o custo do entendimento dos problemas e do desenvolvimento das soluções. O sucesso do desenvolvimento de software baseado em componentes é dependente de boas técnicas que auxiliem a responder como os problemas devem ser decompostos em componentes de software e como as unidades de software, já existentes, podem ser reunidas (Staa, 2000).

Existem diversas definições para o que vem a ser um componente de software, algumas tão antigas quanto a própria área de Engenharia de Software (McIlroy, 1968). Neste trabalho, os componentes são tratados como artefatos de software, sendo utilizados como blocos de construção de uma solução. Neste sentido, componentes podem ser definidos como unidades de composição com interfaces contratualmente especificadas e com dependências explícitas de contexto (Szyperski, 1999). Ainda segundo Szyperski (1999), componentes são unidades binárias, disponibilizadas de forma independente, e passíveis de composição por terceiras partes.

Uma outra definição bastante interessante e sucinta apresenta componentes de software como unidades reutilizáveis que precisam ser integradas como um todo para que desempenhem um papel na solução (Sametinger, 1997). Idealmente, a utilização de componentes de software deve ser verbatim*,* isto é, sem alterar o componente para facilitar a sua adequação a resolução de determinado problema.

O pleno uso de componentes pode ter como conseqüências uma melhora na qualidade das soluções geradas, um aumento da produtividade da equipe e do processo de desenvolvimento utilizado, além de um aumento da confiança e da qualidade dos artefatos gerados. Isto ocorre, pois ao isolar estas unidades, elas podem ser individualmente mais bem testadas e validadas (Sametinger, 1997).

Da busca constante por maiores níveis de reutilização, foram desenvolvidas diversas tecnologias, cada uma com diferentes peculiaridades. Estas tecnologias incluem desde funções, classes, bibliotecas estáticas ou dinâmicas, pacotes, subsistemas, componentes CORBA (OMG CORBA, 1997), processos distribuídos (Andrews, 1991), *Web Services* (Kreger, 2001) até agentes de software (Genesereth & Ketchpel, 1994; Wooldrige & Jennings, 1995). A complexidade e os serviços oferecidos pelas diversas tecnologias variam bastante. Este fato acaba contribuindo para que não exista uma tecnologia integradora que possibilite a unificação entre as diversas propostas existentes, e é a grande motivação para se desenvolver uma abordagem que separe o conceito de composição dos demais conceitos relevantes ao desenvolvimento de software.

A próxima seção aborda os conceitos de composição, padrões de composição, coordenação e padrões de coordenação visando entender algumas das principais variáveis envolvidas no desenvolvimento de soluções baseadas em componentes de software. Ao final deste capítulo, são discutidas em mais detalhes questões relativas à arquitetura de software, sendo apresentado o padrão arquitetural *Layer* (Buschman, 1996).

## 2.2. Composição e Coordenação de Componentes de Software

Ao se distribuir funcionalidades e responsabilidades entre unidades de software, aparece uma questão de suma importância: como adequadamente compor e coordenar as unidades para resolver um problema?

A resposta para esta pergunta não é trivial. Em um primeiro momento, ao se analisar um problema e seus requisitos, e ao se expressar todas as funcionalidades necessárias em componentes de software, é possível identificar um cenário de uso que aparentemente é capaz de resolver o problema, contendo unidades já desenvolvidas ou em vias de desenvolvimento. Porém, será que este cenário realmente resolve o problema? Existem cenários equivalentes ou melhores do que este que é utilizado? É possível configurar a infra-estrutura existente para prestar um serviço melhor?

Para facilitar o processo de análise e auxiliar a formulação de respostas para estas perguntas, os conceitos de composição e coordenação de componentes de software tornam-se extremamente relevantes. Separar as questões relativas à coordenação de componentes, de outras questões como comunicação, interface, persistência, pode vir a ser uma excelente abordagem para a reutilização de soluções (Malone & Crowstone, 1994; Arbab, 1998). A existência de uma grande variedade de tecnologias e a possibilidade de compor unidades de software de diferentes formas são as motivações para separar o conceito de composição dos demais conceitos relevantes ao desenvolvimento de software.



### 2.2.1. Composição de Componentes de Software

Ao longo dos tempos, foram desenvolvidas diferentes propostas para a composição de componentes. Cada uma delas aborda o problema em diferentes níveis de abstração (Weinreich, 1997). Componentes são compostos utilizando diferentes unidades de software, isto é, diferentes conjuntos de instruções descritas em linguagens de programação, linguagens de script, *binding*, *design patterns*, *frameworks*, bibliotecas, técnicas de programação visual, ferramentas de composição, infra-estruturas de componentes, dentre outras opções.

Segundo Heineman e Councill (2001), o processo de composição de componentes é a combinação de dois ou mais componentes de software para produzir um novo componente em outro nível de abstração. As características do novo componente são determinadas pelas unidades que estão sendo combinadas e pela forma como elas são reunidas.

Porém, uma composição só é possível se houver algum grau de interoperabilidade entre as unidades (Sametinger, 1997). Isto é, as partes devem

estar aptas a trocar informações, com entendimento semântico e sintático, além de compartilhar alguma forma de controle por meio de canais de comunicação. Tanto a integração de dados, quanto a integração de controle, ou coordenação da composição, são tópicos de suma importância para o desenvolvimento de software baseado em componentes.

Independentemente da forma como as unidades são compostas, é importante observar que uma composição somente é considerada correta tecnicamente quando as atuais propriedades funcionais e não-funcionais do composto atendem as especificações previstas para a composição (Szyperski, 2002b).

### 2.2.2. Padrões de Composição

Existem diferentes modelos de composição que podem ser estudados e aplicados ao tratamento adequado da integração de software (Nierstrasz & Tsichritzis, 1995; Szyperski, 2002a, 2002b). Cada modelo apresenta vantagens e desvantagens que precisam ser cuidadosamente estudadas e avaliadas antes de sua aplicação.

Um modelo de composição é capaz de refletir um espaço de soluções de integração. Modelos podem apresentar características comuns e recorrentes entre si, o que torna possível a sua classificação segundo algum padrão de composição. Um padrão de composição impõe uma série de restrições e oferece um conjunto de alternativas visando facilitar o aprendizado e o entendimento de questões relativas à reunião de artefatos de software.

Já existem alguns padrões de composição dominantes e bastante difundidos, alguns exemplos destes padrões são discutidos nas próximas subseções. No entanto, diversas abordagens alternativas estão em desenvolvimento e se mostram relevantes na prática, principalmente para a resolução de problemas específicos (Szyperski, 2002b).

#### 2.2.2.1. Composição Funcional

O mecanismo de composição mais usual é a composição funcional (Nierstrasz & Tsichritzis, 1995). A partir deste paradigma, uma entidade é encapsulada e parametrizada como uma abstração funcional. Esta entidade é

ativada (ou instanciada) ao receber argumentos, restritos aos seus parâmetros (Figura 2).

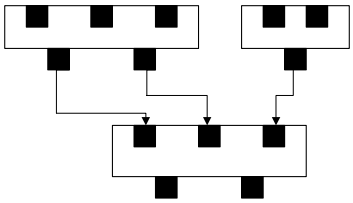

Figura 2 – Ilustração de composição funcional

O paradigma funcional oferece suporte para composições de alta-ordem, isto é, os componentes por si só são dados. Em conseqüência disto, tarefas de composições podem ser encapsuladas como componentes e parte do processo de composição pode ser automatizado. Finalmente, a composição funcional tem a propriedade de ser facilmente verificável desde que funções sejam vistas externamente como caixas pretas. Isto é, a partir de algumas suposições sobre parâmetros de uma função é possível deduzir algumas propriedades de seu resultado (Nierstrasz & Tsichritzis, 1995). A partir da análise destas propriedades, é possível saber se este resultado pode ser seguramente repassado a uma outra função.

**2.2.2.2.**
**Composição Orientada a Dados**

Em composições orientadas a dados, ambientes compartilhados são tipicamente utilizados como um mecanismo de troca de informações (Nierstrasz & Tsichritzis, 1995). Um exemplo deste mecanismo é um *blackboard* (Figura 3). Um *blackboard* é um espaço compartilhado, conhecido por todos os componentes, no qual é possível inserir, atualizar e obter informações. Esta forma de composição suporta a reunião n-ária de componentes.

Além de permitir acesso ao espaço de tuplas, outra conseqüência desta abordagem é que menos restrições precisam ser estabelecidas na interface dos componentes. Por este mesmo motivo, é muito difícil verificar a corretude de sistemas que trabalham com espaço de tuplas, porque as interações entre os componentes não estão localizadas precisamente (Nierstrasz & Tsichritzis, 1995). Como solução parcial para este problema, sistemas de composição podem oferecer mecanismos de encapsulamento que configuram fronteiras dentro do espaço de tuplas, onde a interferência fica restrita a um número bem conhecido de componentes. Neste sentido, este modelo pode impor algumas restrições aos

dados compartilhados, ao invés de impor restrições às interfaces dos componentes.

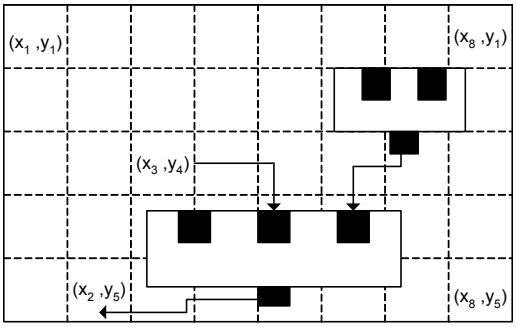


Figura 3 – Ilustração de composição orientada a dados

Mediante esta forma de composição, é possível compor interfaces dinamicamente para acomodar a natureza dos dados utilizados (Szyperski, 2002b). Esta abordagem pode fazer com que uma solução funcione como um sistema orientado a mensagens, que ativa os componentes baseando-se no tipo de informações que estão disponíveis. Neste tipo de sistema, uma composição pode ser vista como o resultado do fluxo de dados corrente.

**2.2.2.3.**
**Composição Orientada a Objetos por Meio de Extensões**

Um paradigma de composição bem difundido em orientação a objeto está associado ao conceito de extensibilidade (Figura 4). Extensibilidade é a possibilidade de adicionar funcionalidades a um componente que permanece compatível com seus usos tradicionais. Por exemplo, a extensibilidade é obtida em *OO* por meio de herança ou de delegação. A extensibilidade é considerada um fator importante para amortecer a evolução de configurações de software (Nierstrasz & Tsichritzis, 1995).

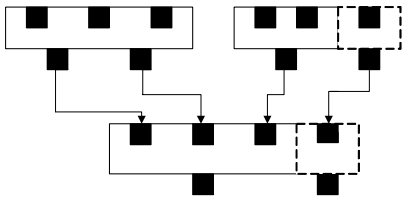

Figura 4 – Ilustração de composição por meio de extensões

Uma observação é que o uso de extensões não deve ser visto como mecanismo para a correção de erros de projeto. Tais erros devem ser tratados através de uma reestruturação, ou *refactoring*. Neste sentido, extensões devem ser utilizadas para instanciar membros específicos dentro de uma família de soluções, mesmo quando essa família não seja conhecida a priori.

### 2.2.2.4. Composição Orientada a Conexão

Objetos não necessariamente formam composições (Szyperski, 2002a). A combinação de objetos e de seus relacionamentos pode apresentar uma série de problemas causados por efeitos colaterais (Szyperski, 2002a). O primeiro passo para resolver este problema é explicitar todas as dependências existentes. Isto é, ao invés de criar dependências entre classes, é preferível criar e manter interfaces que definem todas as dependências entre as unidades de composição (Figura 5).

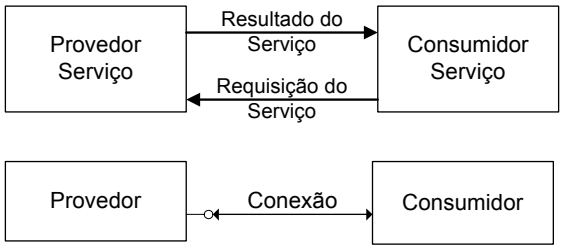


Figura 5 – Ilustração de composição orientada a serviço ou orientada a conexão

A composição orientada a conexão é o extremo da utilização de interfaces para explicitar dependências entre entidades (Szyperski, 2002a). Ela ocorre quando é possível basear toda a interação entre os componentes a partir destas conexões. Este tipo de composição também pode ser caracterizado por conexões do tipo *event source* → *event listener*. Exemplos de tecnologias que impõe este mecanismo são *JavaBeans* ou *.NET Windows Forms*.

A composição orientada a serviço (Figura 5) é outra solução similar a esta, sendo que *Web Services* é um exemplo de tecnologia que se baseia fortemente neste tipo de composição.

### 2.2.2.5. Composição Baseada em Contexto

Diferentemente das abordagens apresentadas até o momento, a composição baseada em contexto se caracteriza por ser assimétrica, isto é, a composição só ocorre por meio da intermediação de uma outra parte (Szyperski, 2002a).

Nesta abordagem, um contexto (Figura 6) tem como objetivo isolar, de forma controlada, instâncias de componentes. Implicitamente, um contexto está relacionado a todas as instâncias de componentes residentes nele, estando estas restritas a todas as políticas impostas e ao uso compartilhado de uma série de serviços oferecidos por ele.

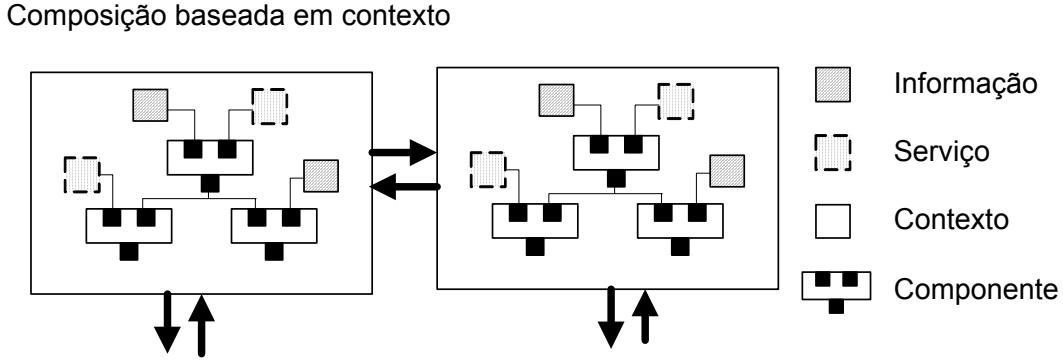


Figura 6 – Ilustração de composição orientada a contexto

Um exemplo de intermediário neste processo de composição é um contêiner de EJB. Uma instância de um componente em um contêiner se beneficia dos serviços oferecidos e da abstração mantida por ele. Esta abstração criada corresponde a uma simplificação do mundo exterior. Ainda sob esta perspectiva, é possível associar o processamento de transações a um grupo de componentes ou restringir o acesso a um conjunto de informações. Um outro exemplo de tecnologia que impõe este padrão de composição é *enterprises services context* do *.NET* (Szyperski, 2002b).

### 2.2.2.6. Resumo de Padrões de Composição

Além dos padrões apresentados, existem outras abordagens como a composição por geração (Batory, 1995), ou a implementação de componentes a partir de interpretadores especializados (por exemplo, interpretadores de diálogo). No caso da utilização de interpretadores especializados, componentes podem ser adaptados por meio de scripts redigidos em uma linguagem que o componente é capaz de interpretar.

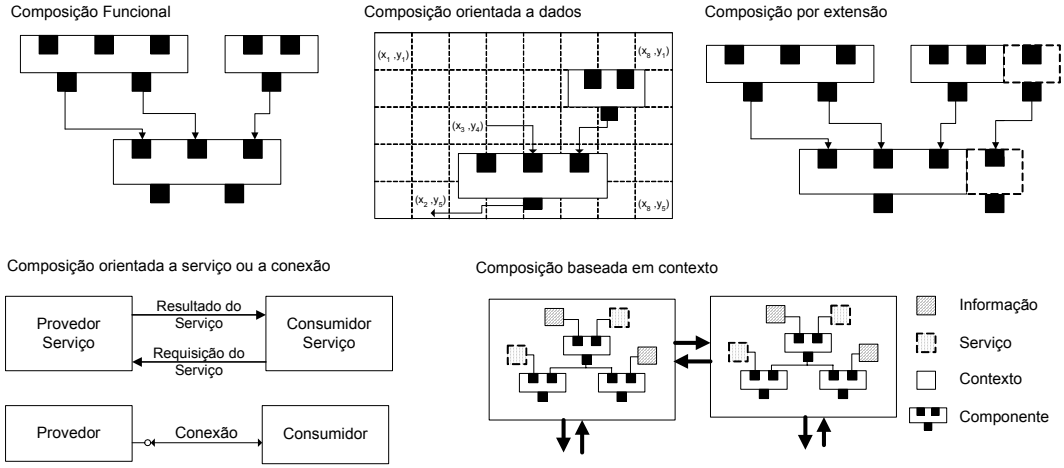


Figura 7 – Ilustração dos padrões de composição apresentados

Como foi descrito, o espectro de abordagens (Figura 7) para a composição é bem rico e a seleção de uma abordagem apropriada para a resolução de um problema pode não ser uma atividade trivial. Dois critérios importantes para auxiliar a seleção de um modelo de composição são o grau de adequação da técnica de composição aos requisitos necessários à solução do problema e o grau de estabilidade que uma composição pode apresentar sob a alteração de suas partes (Szyperski, 2002b). Quando adequado, é possível desenvolver soluções onde são combinados diferentes padrões de composição para resolver determinado problema.

### 2.2.3. Coordenação de Componentes de Software

Malone e Crowston (1994) definem coordenação como a gerência de interdependências entre as atividades executadas por um grupo de entidades para atingir um objetivo.

Em Cruz e Tichelaar (1998) são descritas algumas destas dependências que podem ser consideradas em um processo de coordenação, dentre elas, dependências de fluxo, dependências de compartilhamento e dependências de convivência. Dependências de fluxo ocorrem quando entidades produzem algum resultado que é utilizado por outras entidades. Esta dependência pode ser vista como a combinação de restrições de pré-requisitos (um item precisa ser produzido antes de ser utilizado), restrições de acessibilidade (um item que é produzido precisa estar disponível para ser utilizado) ou restrições de usabilidade (um item que é produzido precisa ser utilizável por outras atividades). A dependência de compartilhamento ocorre quando múltiplas atividades competem pela utilização de um mesmo recurso, e a dependência de convivência ocorre quando múltiplas atividades produzem resultados que precisam ser utilizados em conjunto por um grupo de componentes de software.

Tradicionalmente, componentes são utilizados em uma infra-estrutura de software fixa que invoca cada unidade e fornece tratamento e coordenação adequados para o funcionamento entre as partes. A princípio, esta abordagem pode não facilitar, ou até dificultar, a evolução de uma solução. Necessariamente, ao modificar determinado cenário de uso de um componente é preciso alterar ou configurar a infra-estrutura existente.

Atualmente, a infra-estrutura de coordenação vem sendo reconhecida como um componente independente que pode ser disponibilizado de forma pré-configurada (Heineman & Councill, 2001). Por meio da organização de conceitos de coordenação, Garlan e Shaw (1993) desenvolveram a base para o estudo desta infra-estrutura em seu trabalho que cataloga diferentes estilos arquiteturais. Um estilo arquitetural é determinado por um conjunto de tipos de componentes, pelos mecanismos de interação previstos, que determinam como os componentes são coordenados, e por uma configuração topológica, indicando os relacionamentos e os padrões de interação possíveis entre estes componentes.

### 2.2.4. Padrões de Coordenação

Assim como em composição, existem diferentes modelos de coordenação que podem ser estudados e aplicados ao controle de atividades de entidades de uma aplicação (Hollingsworth, 1995; Cruz & Tichelaar, 1998; Lano et al., 2002; IBM BPEL, 2002). Cada modelo apresenta vantagens e desvantagens que precisam ser cuidadosamente estudadas e avaliadas antes de sua aplicação.

Um modelo de coordenação é capaz de oferecer uma estrutura básica para explicitar e gerir um conjunto de dependências entre as entidades. Estes modelos podem apresentar características comuns e recorrentes entre si, o que torna possível a sua classificação segundo algum padrão de coordenação.

#### 2.2.4.1. Padrão Baseado na Teoria da Superposição

Este padrão de coordenação, proposto por Andrade et al. (2000), é baseado nos conceitos de superposição (Katz, 1993). A partir dele, as chamadas explícitas a um método de *Subject* (Figura 8) correspondem a um evento e são desacopladas da execução do serviço, permitindo que as regras definidas em um contrato interceptem a chamada. Após esta interceptação, é possível encadear ou sobrepor o comportamento necessário para gerar a solução desejada. Este mecanismo é especificado por meio de regras do tipo evento → reação. Este conjunto de regras pode incluir ações antes ou depois da execução de determinado serviço, além de permitir a substituição do comportamento a ser executado.

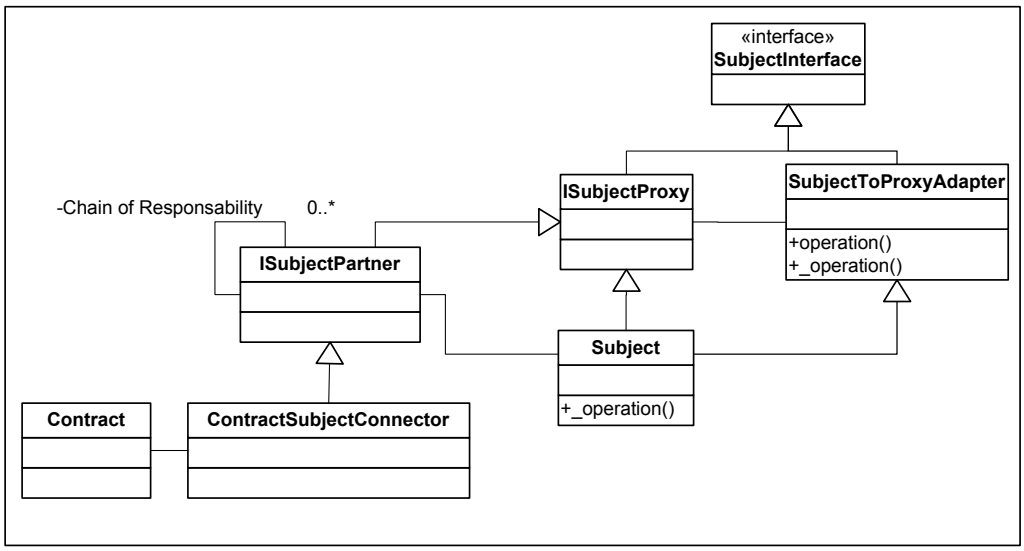


Figura 8 – Padrão de coordenação proposto por Andrade et al. (2000)

Neste projeto de padrão de coordenação (Figura 8), a execução de operações em um *Subject* está associada a uma cadeia de ações especificadas em um contrato. Esta cadeia de ações é estruturada pelo padrão de projeto *Chain of Responsability* (Gamma et al., 1995). Cada nó desta cadeia é implementado por instâncias de extensões da classe *ISubjectPartner*. O acesso a um *Subject* é feito via *proxies* criados especificamente para esta tarefa.

### 2.2.4.2. Padrão Baseado em Políticas de Controle a Recursos

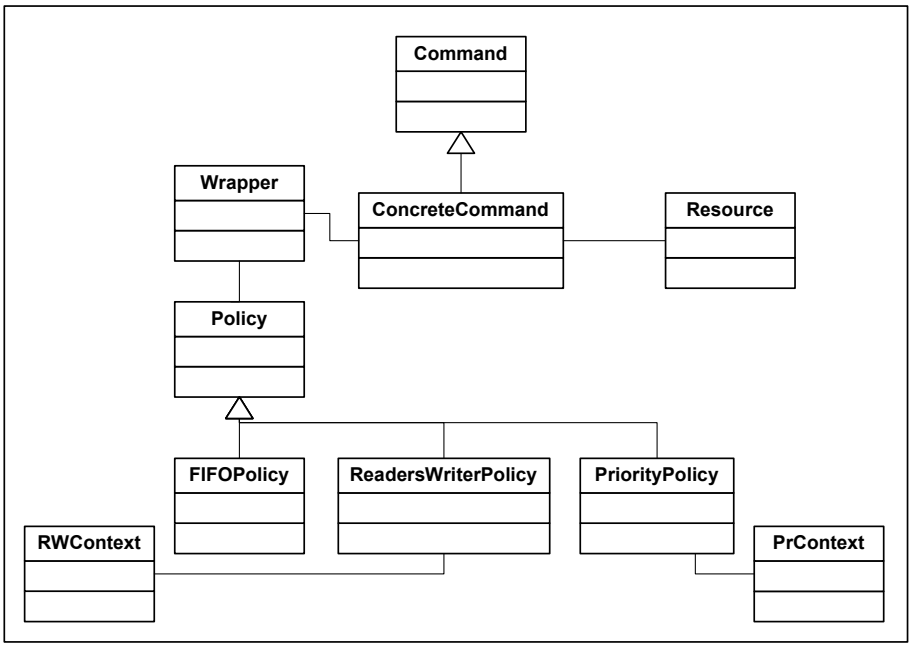


Figura 9 – Padrão de coordenação proposto por Cruz e Tichelaar (1998)

No padrão de coordenação proposto por (Cruz & Tichelaar, 1998), a classe *Wrapper* (Figura 9) funciona como uma interface, recebendo as requisições de manipulação ou execução de operações da classe *Resource*. A forma de acesso a um recurso é estabelecida por um tratamento implementado por um *Command* (Gamma et al., 1995) concreto. Esta representação explícita de comandos possibilita a organização da execução de requisições, como por exemplo, a mudança da ordem em que as requisições são tratadas, a possibilidade de sua execução em paralelo, ou qualquer restrição que precise ser imposta por uma política de coordenação.

Toda esta organização é feita por meio de políticas (extensões de *Policy*) associadas à classe *Wrapper*. A estas políticas pode ser associado um contexto a partir do qual informações são utilizadas para auxiliar o processo de tomada de decisão.

#### 2.2.4.3. Padrão Baseado na Execução Condicional de Tratamentos a Contextos

Neste padrão de coordenação (Figura 10), existe um mecanismo de controle da execução de uma tarefa, representada pela classe Tratamento. Este controle se baseia em condições associadas entre o Processo e um Tratamento específico. Isto é, existem processos associados em um contexto, que por sua vez possuem uma série de informações estruturadas que podem ser consultadas para a realização das tarefas. Um processo dispõe de uma série de tratadores que são responsáveis por executar as ações e manipular as informações. A execução de um Tratamento só ocorrerá caso a condição associada a ele seja válida.

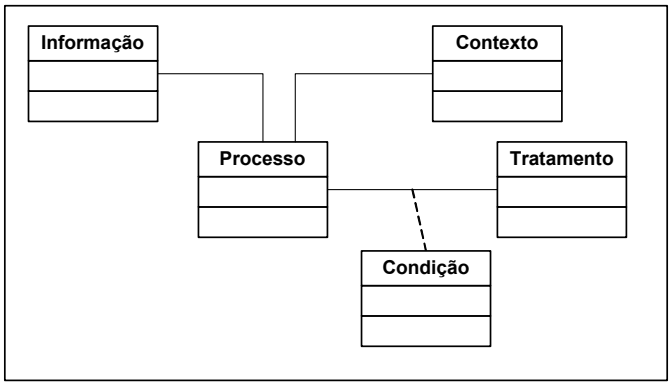


Figura 10 – Exemplo de padrão de coordenação

Este modelo de coordenação é similar à proposta apresentada em BPEL (IBM BPEL, 2002) para o controle e o encadeamento da execução de serviços.

## 2.3. Arquitetura de Software

Uma arquitetura de software envolve a descrição dos elementos básicos utilizados para a construção de sistemas, das interações entre estes elementos, dos padrões que guiam a sua utilização, e das restrições aplicadas a estes padrões (Shaw & Garlan, 1996). Resumidamente, uma arquitetura é utilizada para descrever o sistema em termos das unidades computacionais e das interações ou relacionamentos entre estas unidades (Buschman, 1996).

Um exemplo de unidades computacionais são camadas organizadas hierarquicamente em um sistema. Neste caso, as interações correspondem à utilização dos relacionamentos entre as abstrações previstas nestas camadas.

Além de especificar a topologia de um sistema, uma arquitetura pode ser utilizada para mostrar a correspondência entre os requisitos de um sistema e os elementos de construção do mesmo, provendo assim alguma racionalidade ao processo de decisões que podem ser tomadas ao longo do ciclo de vida de uma aplicação (Shaw & Garlan, 1996).

Neste trabalho, uma arquitetura de software representa um modelo reutilizável de soluções que pode ser aplicado ao desenvolvimento de um conjunto de sistemas (SEI, 2002). Neste sentido, uma arquitetura deve ser vista como uma estrutura conceitual utilizada para organizar o desenvolvimento de software.

Padrões arquiteturais são descrições utilizadas para documentar e preservar o conhecimento sobre as características comuns a uma série de arquiteturas de software. Um padrão arquitetural expressa um esquema recorrente de uma organização estrutural e funcional utilizada no desenvolvimento de sistemas de software (Buschman, 1996). A documentação final provê um conjunto predefinido de subsistemas, especificando as suas responsabilidades, além de regras e diretivas para a organização dos relacionamentos entre eles (Buschman, 1996).

Padrões arquiteturais são utilizados como modelos para a elaboração de novas arquiteturas concretas, auxiliando o desenvolvimento de soluções de software. O processo de escolha de um padrão adequado deve levar em consideração as características do problema abordado e o auxílio efetivo oferecido pelo padrão ao desenvolvimento, manutenção e evolução dos sistemas de software gerados a partir dele (Buschman, 1996).

### *2.3.1.* Padrão Arquitetural *Layer*

O padrão arquitetural *Layer* auxilia a estruturar uma aplicação, decompondo-a em grupos de sub-tarefas, sendo que cada grupo está em um nível de abstração (Buschman, 1996). Neste tipo de abordagem, devem existir questões mais abstratas que dependem, de alguma forma, de questões mais concretas.

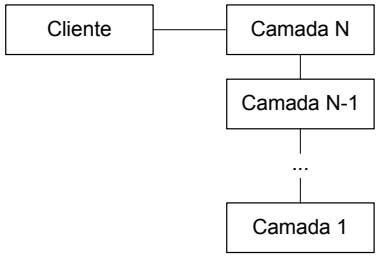


Figura 11 – Padrão arquitetural *Layer* (Buschman, 1996)

A partir deste entendimento, é necessário determinar a quantidade adequada de níveis de abstração, organizando as camadas hierarquicamente, de acordo com seu grau de detalhes e as dependências existentes entre elas (Figura 11). Começando pelo nível mais concreto, ou menos abstrato, define-se a base do sistema. Elaborando a arquitetura, todas camadas superiores devem ser determinadas em função de novas abstrações e de outras abstrações que porventura já foram definidas em camadas inferiores.

A maioria dos serviços que a camada N oferece são compostos por serviços oferecidos pela camada N-1. Em outras palavras, os serviços de uma camada implementam uma estratégia para a combinação dos serviços da camada inferior. Além disto, os serviços de uma camada podem depender de serviços oferecidos pela própria camada. O exemplo mais difundido de uso deste padrão arquitetural é a estruturação das camadas de protocolos de rede (Buschman, 1996).

Ao utilizar este padrão, algumas restrições devem ser observadas (Buschman, 1996). Alterações em código fonte devem estar confinadas a um único nível e não devem alterar os demais. Visando minimizar as conseqüências destas alterações, as interfaces utilizadas para o relacionamento entre as abstrações devem ser estáveis. Partes do sistema podem ser modificadas, isto é, podem ser alteradas por implementações alternativas, desde que não afetem o restante do sistema. Finalmente, responsabilidades similares devem ser agrupadas em uma mesma camada para auxiliar o seu entendimento e a sua manutenibilidade.