



Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Arquitetura de Software

[Software Architecture: Foundations, Theory, and Practice]
R. N. Taylor, N. Medvidovic, E. M. Dashofy. Capítulos 2, 3, 4, 5 e 11

Sandro S. Andrade
sandroandrade@ifba.edu.br



Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Arquitetura de Software - Introdução

Sandro S. Andrade
sandroandrade@ifba.edu.br

Introdução



- A área de Arquitetura de Software estuda como sistemas de *software* são projetados e construídos
- Arquitetura de Software:
- Conjunto formado pelas principais decisões de projeto tomadas durante seu desenvolvimento e qualquer evolução subsequente
- Desenvolvimento de *software* centrado em arquiteturas
- Qualidade de projeto → Qualidade de Software
- Famílias de produtos de *software*

Software e Construção Civil



- Analogia forte e de fácil compreensão
- As fases são similares (requisitos, projeto, etc)
- Outras similaridades:
- Projeto arquitetural com foco nas necessidades dos usuários
- Permite especialização de trabalho
- Planos e progressos podem ser avaliados em pontos intermediários
- Mas não é só isso ...

Software e Construção Civil



- 1) Toda construção tem uma arquitetura, separada, porém relacionada, à estrutura física
 - Esta arquitetura pode ser descrita, discutida e comparada com as de outras construções
 - A arquitetura antecipadamente projetada pode ser comparada com a arquitetura resultante do processo de construção
 - De forma similar, a arquitetura de um *software* existe de forma independente, porém relacionada, ao código-fonte que a implementa

Software e Construção Civil



2) Propriedades das estruturas são induzidas pelo projeto das suas arquiteturas:

- Castelo medieval: paredes altas e espessas e janelas estreitas, se existentes. Induz propriedades defensivas
- Propriedades de um *software*, como resiliência a tipos particulares de ataques, são determinadas pelo projeto de suas arquiteturas



Software e Construção Civil



- Objetivos de uma boa arquitetura:
 - Força: fundações para um assoalho sólido, escolha apropriada de materiais sem economia
 - Utilidade: distribuição sensata das partes, com seus propósitos devidamente atendidos e situação apropriada
 - Beleza: aparência agradável, boa percepção do “todo” e dimensões proporcionais entre as partes

Software e Construção Civil



3) Reconhecimento do papel distinto e característico do arquiteto – pessoa que cria a arquitetura

- Exige ampla formação:
 - Aspectos de engenharia
 - Senso apurado de estética
 - Conhecer o modo como as pessoas trabalham, comem, brincam e moram ajuda a projetar construções satisfatórias e que funcionam bem ao longo das estações e dos anos
 - Habilidades simples de programação não são suficientes para a criação de sistemas complexos que efetivamente funcionam

Software e Construção Civil



4) O processo não é tão importante quanto a arquitetura

- Isso não quer dizer que o processo não é importante, somente que ele não é garantia de sucesso
- O processo existe para servir um fim – o projeto e a qualidade da infra-estrutura – não para ser um fim em si próprio

Software e Construção Civil



5) A arquitetura (de *software*) amadureceu, ao longo dos anos, como uma disciplina

- Uma base de conhecimentos está disponível, capturando as experiências e lições de projeto prévios
- Foco no reuso de conhecimento, de projeto de sub-sistemas e de ferramentas
- Benefício de uso de materiais, partes e tamanhos padronizados

Software e Construção Civil



- Limitações da analogia:
 - Conhecemos muito sobre prédios e não tanto sobre *software*
 - Natureza essencial dos materiais totalmente diferente
 - O *software* é mais “maleável” do que os materiais físicos de construção
 - A indústria da construção civil é mais consolidada
 - O fase de implantação não existe na construção civil
 - Caráter extremamente dinâmico do *software*

Software e Construção Civil



- Resumindo:
 - A arquitetura do *software* deve ser o centro do projeto e desenvolvimento de sistemas, mais importante que o processo, análise e até mesmo programação
 - Ao dar proeminência à arquitetura obtém-se: controle intelectual, integridade conceitual, base adequada e efetiva para reuso, comunicação efetiva no projeto e gerenciamento de um conjunto de sistemas variantes, porém relacionados
 - O foco na arquitetura deve estar presente em todas as fases do projeto

Exemplos

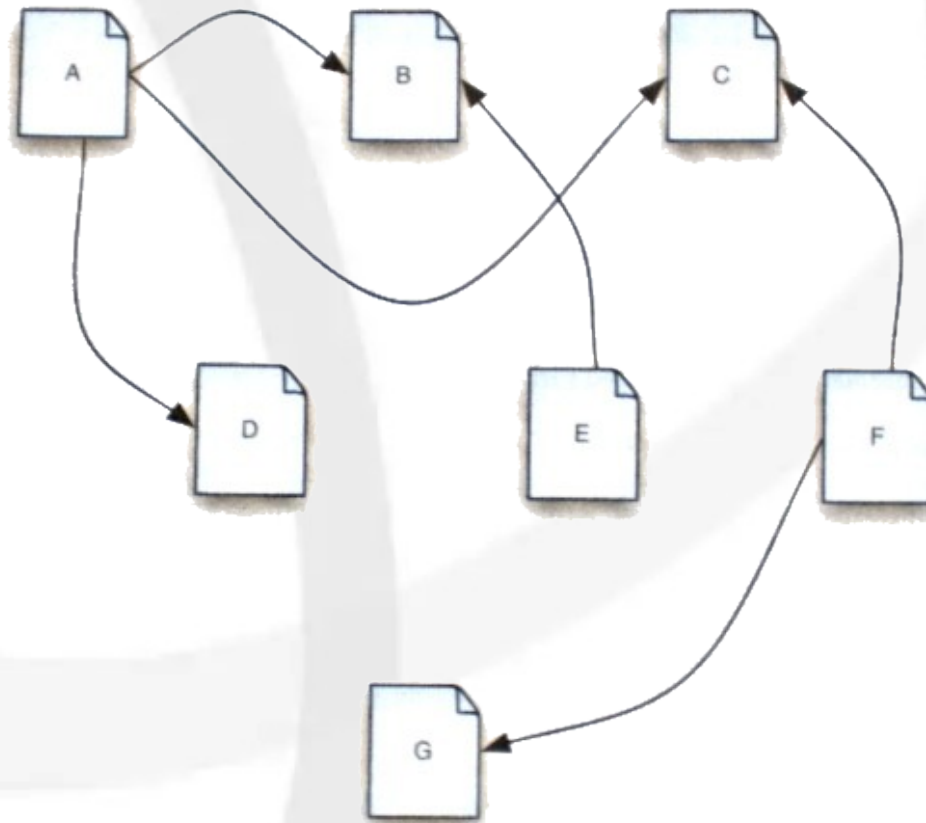


- Exemplo 1: Arquitetura da web
 - O que é a web ? Como ela é construída ? Como você projetaria um *software* para um site de comércio eletrônico ?
 - A arquitetura do sistema fornece o vocabulário e os meios para responder as questões acima, em particular o estilo arquitetural adotado para a web

Exemplos



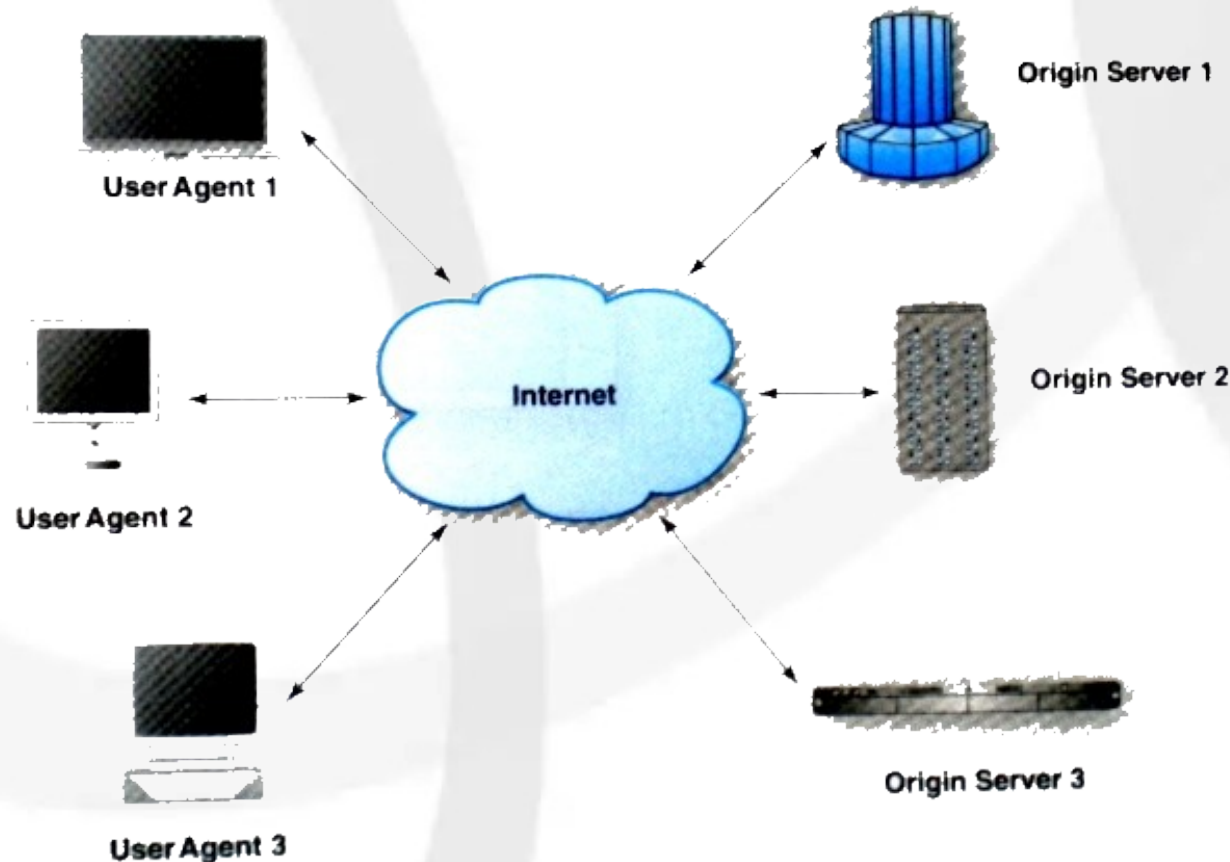
- Visão do usuário: conjunto dinâmico de relacionamentos entre coleções de informação



Exemplos

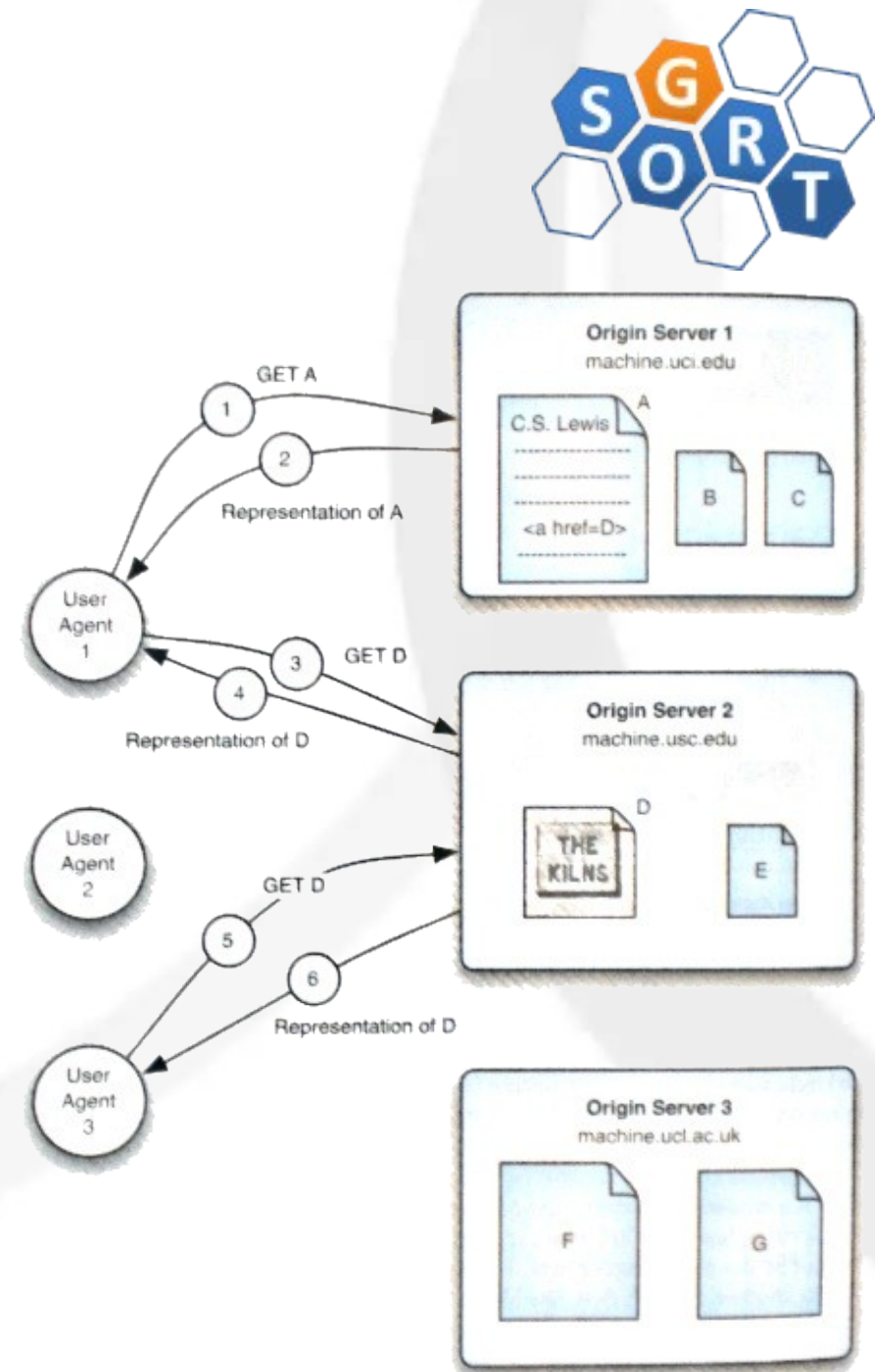


- Visão de rede: coleção de máquinas independentemente apropriadas e operadas, que se comunicam via rede



Exemplos

- Visão do desenvolvedor: coleção de programas independentemente desenvolvidos que se comunicam através dos padrões HTTP, URI, MIME e HTML



Exemplos



- Estas visões não explicam como a web funciona
- Uma estratégia melhor é apresentar um conjunto de definições e restrições que caracterizam a web:
 - Coleção de *resources*, identificados unicamente por uma URL
 - Cada *resource* denota uma informação, como um documento, imagem, serviço, coleção de outros resources, etc
 - URL's podem ser utilizadas para determinar a identidade da máquina que contém o *resource*
 - Toda comunicação é iniciada pelos clientes (*user agents*), realizando requisições aos servidores

Exemplos



- Uma estratégia melhor é apresentar um conjunto de definições e restrições que caracterizam a web:
 - *Resources* podem ser manipulados através de suas representações. O HTML é a linguagem de representação mais comum da web
 - Toda comunicação entre clientes e servidores é realizada através de um protocolo extremamente simples (HTTP), com poucas primitivas, tais como GET e POST
 - Toda comunicação entre clientes e servidores é *context-free*, ou seja, o servidor responde à requisição baseando-se somente na informação presente na própria requisição. Nenhum histórico de operações é mantido

Exemplos



- Exemplo 2: *Shell Script*

ls invoices | grep -e August | sort

- Um filtro é um programa que recebe um fluxo de caracteres como entrada e produz um fluxo de caracteres como saída. Filtros podem ser parametrizados
- Um *pipe* é uma forma de conectar dois filtros, onde a saída do primeiro filtro é conectada à entrada do segundo
- Conhecendo os filtros e *pipes* utilizados pode-se facilmente compreender o programa e criar outros
- O conjunto particular de regras aqui aplicado define um estilo arquitetural conhecido como *Pipe-and-Filter*
- Pode ser utilizado em qualquer sistema

Exemplos

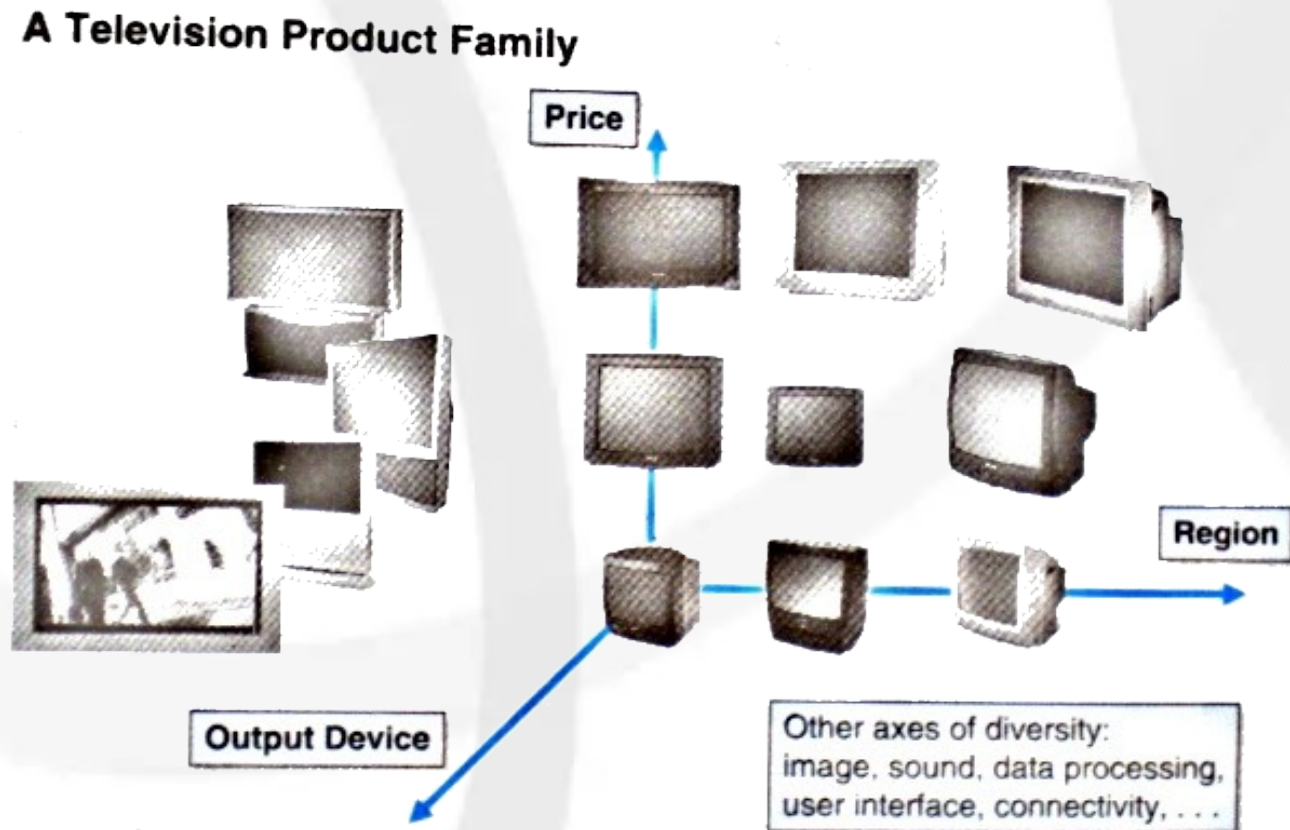


- Exemplo 3: Linhas de Produto
 - Famílias de produto são conjuntos de programas independentes que possuem um alto potencial de compartilhamento de estrutura e componentes constituintes
 - Ex: HD TV 35" com *DVD player* com sinal ATSC, HD TV 35" sem *DVD player* com sinal ATSC, HD TV 35" com *DVD player* com sinal DVB-T
 - Reutilizar estruturas, comportamentos e implementações simplifica o desenvolvimento, reduz prazos e custos e melhora a confiabilidade geral do sistema
 - Arquiteturas de *software* são abstrações essenciais para o gerenciamento de variações e de pontos em comum

Exemplos



- Linha de produtos da Philips
 - Metodologia arquitetural: Koala



Exemplos

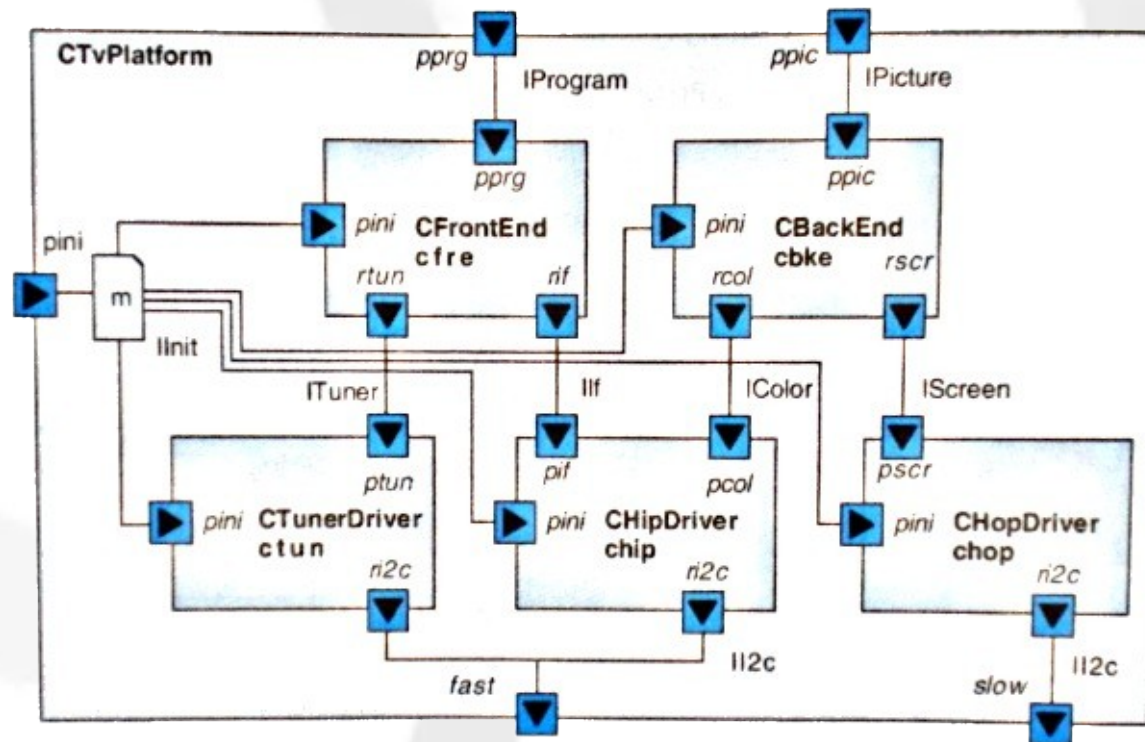


- Koala:
 - Modela e implementa o *software* como uma coleção de componentes que interagem entre si
 - Cada componente exporta um conjunto de serviços através de um conjunto de *provided interfaces*
 - Cada componente explicitamente define suas dependências com o ambiente (*hardware* ou *software*) através de um conjunto de *required interfaces*

Exemplos



- Arquitetura exemplo de uma plataforma de TV da Philips:
 - Compatibilidade de interfaces
 - *Composite*

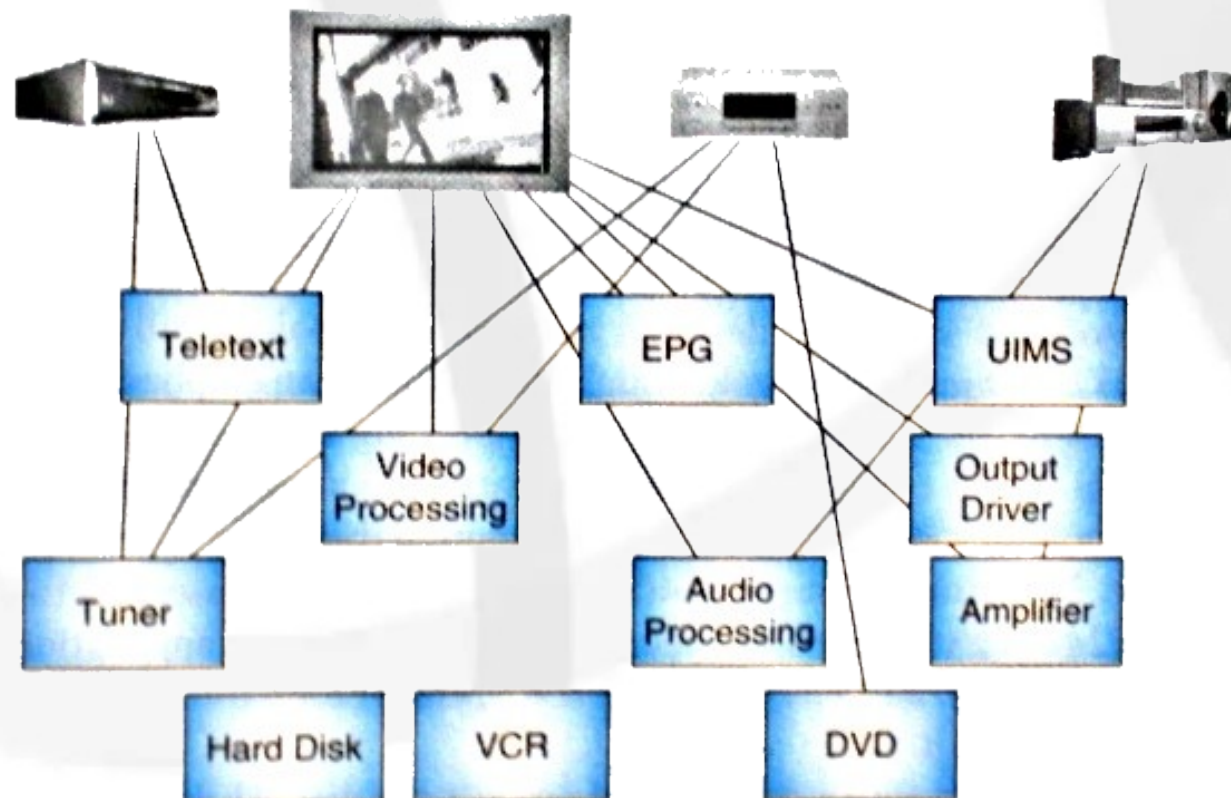


Exemplos



- Koala: população de produtos

Composition





Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Arquitetura de Software - Conceitos Básicos

Sandro S. Andrade
sandroandrade@ifba.edu.br

Arquitetura de Software



- Na sua essência a arquitetura de um *software* pode ser definida como:

Arquitetura de Software: conjunto formado pelas principais decisões de projeto tomadas a respeito do sistema

Arquitetura de Referência: conjunto formado pelas principais decisões de projeto que são simultaneamente aplicadas a múltiplos sistemas relacionados, geralmente dentro de um domínio de aplicação, com pontos de variação explicitamente definidos

Arquitetura de Software



- Mas o que é uma “decisão de projeto”? Exemplos:
 - Relacionada à estrutura do sistema: “os elementos arquiteturais devem ser organizados e compostos da seguinte forma: ...”
 - Relacionada ao comportamento funcional: “o processamento, armazenamento e visualização de dados serão realizados exatamente nesta sequência”
 - Relacionada a interações: “a comunicação entre todos os elementos do sistema será realizada somente através de notificação de eventos”
 - Relacionada a propriedades não-funcionais: “*dependability* será garantida através de módulos replicados de processamento”
 - Relacionada à implementação do sistema: será utilizado o *Java Swing* para os componentes de GUI

Arquitetura de Software



- Nem todas as decisões de projeto são “principais”:
 - Depende do grau de importância e particularidade da decisão
 - Detalhes dos algoritmos e estruturas de dados utilizados não são decisões principais
 - Pode depender das metas do sistema e dos interesses dos stakeholders
- Resumindo, a arquitetura é também determinada pelo contexto (eventualmente dirigido por questões não-técnicas)
- Diferentes *stakeholders* podem julgar como principais diferentes conjuntos de decisões

Arquitetura Prescritiva



- Arquitetura prescritiva:

Arquitetura Prescritiva: conjunto P formado pelas principais decisões arquiteturais tomadas pelos arquitetos em um tempo t qualquer. É a prescrição para a construção do sistema

- Representa a arquitetura “pretendida” ou “concebida” do sistema
- Pode não existir de forma tangível, apenas na cabeça do arquiteto
- ... ou pode ser capturada através de alguma notação ou outra forma de documentação

Arquitetura Prescritiva



- As decisões que compõem a arquitetura prescritiva serão implementadas por um conjunto A de artefatos:
 - Representação das decisões arquiteturais em UML
 - Implementações em uma linguagem de programação
 - Modelos dos estilos arquiteturais e padrões utilizados
 - Componentes *COTS* a serem utilizados
 - Infra-estruturas de *middleware* e *frameworks*
- Cada artefato em A encapsula certas decisões de projeto

Arquitetura Descritiva



- Arquitetura descritiva:

Arquitetura Descritiva: conjunto D formado pelas principais decisões arquiteturais encapsuladas por todos os artefatos do conjunto A . Descreve como o sistema foi implementado

- Representa a arquitetura “implementada” do sistema
- No início do projeto (tempo t_1) tem-se a criação do conjunto P_1 e os conjuntos A_1 e D_1 podem estar vazios (desenvolvimento *greenfield*)
- No desenvolvimento *brownfield*, onde um conjunto de artefatos implementando parcialmente a arquitetura já existe, A_0 e D_0 são não-vazios enquanto P_0 é vazio

Arquitetura Descritiva



- P0 também pode estar vazio e D0 com alta cardinalidade em um projeto envolvendo um sistema legado cuja intenção arquitetural foi perdida ao longo do tempo
- Tais discrepâncias entre os conjuntos P e D podem ser indícios de problemas na arquitetura do sistema

Degradação Arquitetural



- Degradação Arquitetural:
 - Durante a vida de um sistema várias arquiteturas prescritivas e descritivas serão criadas
 - Cada par correspondente de arquiteturas representa o sistema em um determinado tempo t
 - Quando os conjuntos P , A e D estiverem suficientemente completos e consistentes a aplicação é implantada
 - Em um cenário ideal P será sempre igual a D
 - Nem sempre é o caso:
 - *COTS* ou plataformas de *middleware* podem interferir nas decisões arquiteturais tomadas
 - É preciso que os *stakeholders* definam o limite aceitável de diferenças entre P e D

Degradação Arquitetural



- Degradação Arquitetural:
 - É possível que, dado os conjuntos P e D no tempo t , esses conjuntos permaneçam estáveis no tempo $t+1$, mesmo com um crescimento de A
 - É também possível que P mude enquanto D permanece o mesmo
 - De forma similar, D pode mudar enquanto P permanece o mesmo

Degradação Arquitetural



- Degradação Arquitetural:
 - Durante uma evolução o ideal é alterar primeiro P e depois D
 - Nem sempre isso acontece:
 - Por desleixo do desenvolvedor
 - Por prazos curtos que impedem o raciocínio e a documentação do impacto na arquitetura prescritiva
 - Por ausência de documentação da arquitetura prescritiva
 - Necessidade ou desejo de otimizar o sistema, “fato que pode ser feito somente no código”
 - Técnicas e ferramentas inadequadas
 - Qualquer que seja a razão elas são falhas e potencialmente perigosas

Degradação Arquitetural



Degradação Arquitetural: discrepância existente entre as arquiteturas prescritiva e descritiva do sistema

Desvio Arquitetural: é a introdução, na arquitetura descritiva do sistema, de decisões principais de projeto que: a) não estão incluídas na arquitetura prescritiva ou não são implicações dela, mas b) não violam nenhuma das decisões de projeto da arquitetura prescritiva

Erosão Arquitetural: é a introdução, na arquitetura descritiva do sistema, de decisões principais de projeto que violam decisões da arquitetura prescritiva

Degradação Arquitetural (Desvio)



- O desvio arquitetural é resultado de mudanças no conjunto A que resultam, por sua vez, em mudanças no conjunto D
- Nem todas as expansões de D resultam em desvio arquitetural:
 - Ex: P requer que criptografia seja utilizada na comunicação em rede pública e D pode afirmar que um algoritmo de chave pública será utilizado para suportar tal comunicação
- Exemplo de desvio arquitetural: ligação entre dois conectores na figura anterior:
 - Não existe em P, porém não foi afirmado que ligações entre conectores não poderiam ser realizadas

Degradação Arquitetural (Desvio)



- Desvios arquiteturais podem causar violações nas regras do estilo arquitetural
- Refletem a insensibilidade do engenheiro em relação à arquitetura do sistema, podendo conduzir a perdas na clareza da forma e da compreensão do sistema
- Se não apropriadamente corrigidos, desvios arquiteturais frequentemente evoluem para erosões arquiteturais

Degradação Arquitetural (Erosão)



- Erosões arquiteturais produzem sistemas difíceis de entender e adaptar e frequentemente com falhas em potencial
- Podem ocorrer quando um sistema sofre vários desvios e as decisões estão obscurecidas por várias mudanças pequenas intermediárias
- Embora seja menos provável de acontecer e mais fácil de corrigir, uma arquitetura pode sofrer erosão se ter tido desvios anteriores
- Pode ser causada por decisões que funcionam bem isoladas mas geram problemas em conjunto

Visão Arquitetural



- Visão arquitetural:
 - Tem como objetivo destacar determinados aspectos de uma arquitetura ao mesmo tempo em que omite outros

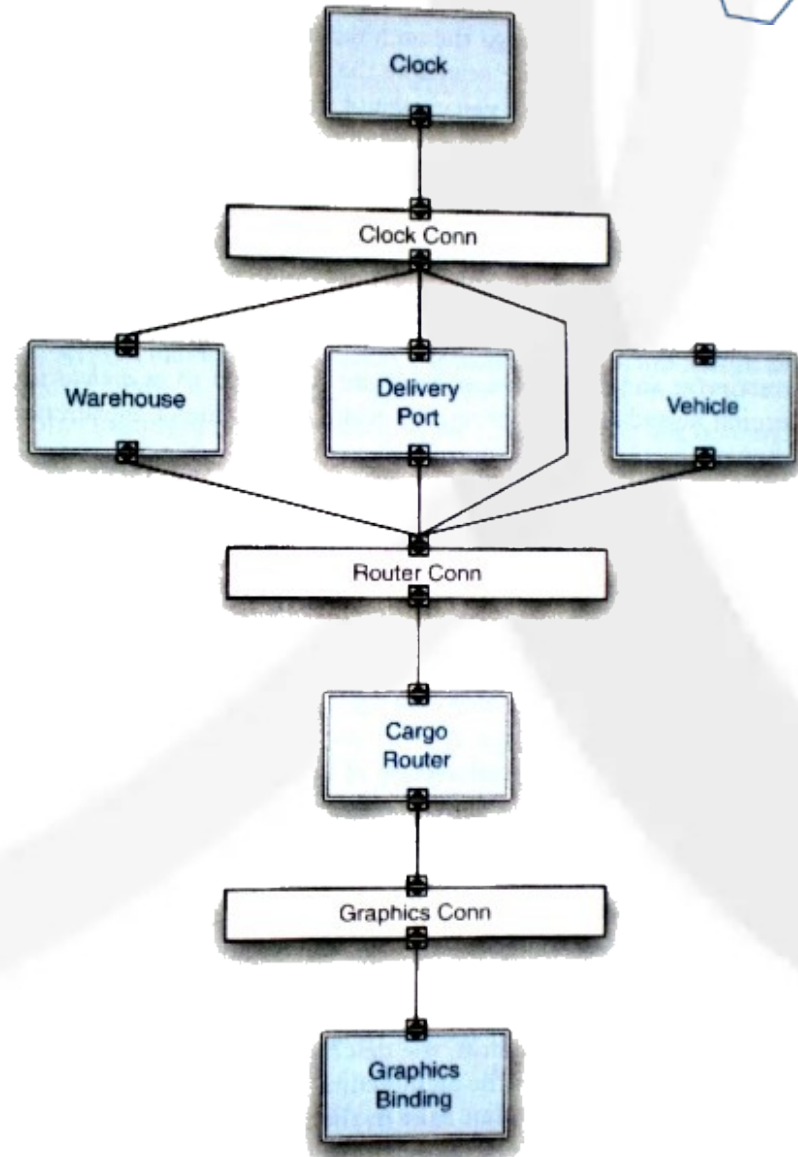
Visão Arquitetural: é um conjunto não-vazio de tipos de decisões arquiteturais de projeto

- Diversos *stakeholders* acrescentam decisões em diferentes detalhes e níveis de abstração
- Uma visão arquitetural direciona a atenção a um subconjunto dessas decisões

Visão Arquitetural



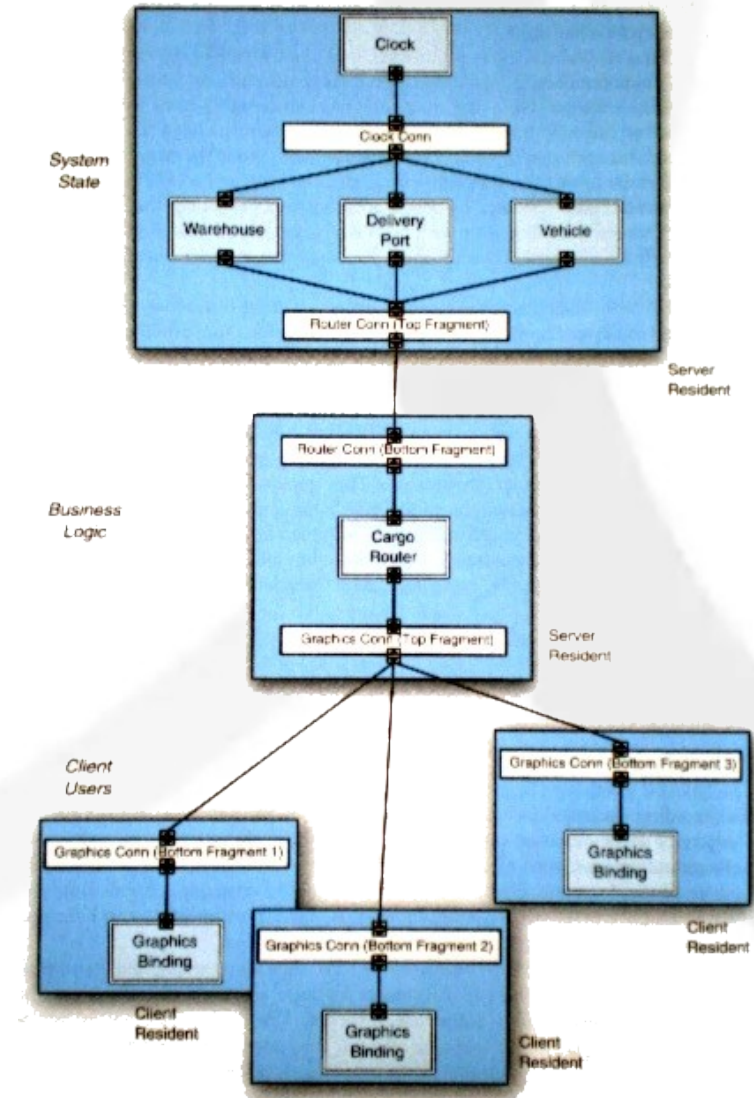
- Exemplo: visão estrutural
 - Nenhuma informação sobre comportamento, interações etc



Visão Arquitetural



- Exemplo: visão de implantação
 - Importante na avaliação da capacidade de satisfação dos requisitos
 - Ex: muitos componentes pesados em uma máquina com pouca memória e CPU modesta
 - Ex: transferência de altos volumes de dados em redes com baixa largura de banda



Arquitetura de Software



Arquitetura de Software = { *elements* (de processamento, dados ou conexão), *form*, *rationale* (intenções, pressupostos, escolhas sutis, restrições externas, estilos e padrões adotados, etc) }

[Perry & Wolf - 1992]

Arquitetura de Software: é a organização fundamental do sistema, implementada por seus componentes, os relacionamentos entre eles e deles com o ambiente, e os princípios que governam seu projeto e evolução

[ANSI/IEEE Standard 1471-2000]

A Arquitetura de Software de um sistema já implantado é determinada pelos seus aspectos mais difíceis de serem modificados

[Chris Verhoef - 2005]

Componente



- Elementos de uma arquitetura geralmente implementam:
 - Processamento: funcionalidade ou comportamento
 - Estado: informação ou dados
 - Interação: inter-conexão, comunicação, coordenação e mediação
- Os componentes de *software* lidam com os dois primeiros problemas:

Componente de Software: é uma entidade arquitetural que: 1) encapsula um subconjunto das funcionalidades e/ou dados do sistema; 2) restringe o acesso a este subconjunto através de interfaces explicitamente definidas; e 3) possui dependências - explicitamente definidas – em relação ao seu contexto de execução

Componente



- Um componente é um *locus* de computação e estado em um sistema [Shaw et al. - 1995]
- Pode ser simples como uma única operação ou complexo como um sistema inteiro
- É visto pelo usuário (humano ou outro *software*) somente através da sua interface pública
- São aplicações dos princípios de encapsulamento, abstração e modularidade

Componente



- O tratamento explícito do contexto de execução do qual o componente depende pode informar:
 - Interfaces requeridas pelo componente (*required interfaces*): serviços disponibilizados por outros componentes e dos quais o componente em questão depende para o seu correto funcionamento
 - Recursos necessários: arquivos de dados ou diretórios necessários ao componente
 - Softwares do sistema requeridos: ambientes de *run-time* plataformas de *middleware*, sistemas operacionais, protocolos de rede, *drivers* de dispositivos, etc
 - Configurações de *hardware* necessárias para executar o componente

Componente



- Geralmente são *application-specific*, mas não é sempre o caso (ex: servidores web, *front-ends*, *back-ends*, *toolkits* para GUI, componentes *COTS*, etc)
- Outra definição de componentes de *software*:

Componente de Software: unidade de composição formada somente de interfaces definidas de forma contratual e dependências explícitas de contexto

[Szyperski - 1997]

Conector



- Sistemas modernos são formados por um grande número de componentes complexos, distribuídos em múltiplos hosts (possivelmente móveis) e atualizados dinamicamente sem interrupção do serviço
- Nestes sistemas garantir uma interação apropriada pode ser mais importante e desafiador do que a implementação dos componentes

Conector de Software: elemento arquitetural responsável por efetivar e regular as interações entre componentes

Conector



- Em sistemas *desktop* convencionais os conectores são geralmente representados por simples chamadas de procedimento (*Procedure Call*) ou acesso a dados compartilhados (*Shared Ddata Access*)
- São constantemente não representados nas arquiteturas e se resumem a um meio de permitir a interação entre pares de componentes
- Entretanto, em sistemas complexos os conectores passam a ter identidades, papéis e artefatos de implementação únicos
- São elementos críticos, ricos e sub-apreciados

Configuração Arquitetural



Configuração Arquitetural: conjunto de associações específicas entre os componentes e os conectores de uma arquitetura de *software*

- Geralmente representada por um grafo onde nós são componentes e conectores e arestas ligações
- Indica uma possível comunicação entre componentes, mas não garante a real habilidade deles se comunicarem
- As ligações devem ser entre interfaces compatíveis. Caso contrário tem-se um *architectural mismatch*

Estilo Arquitetural



- Experiências prévias podem evidenciar certas decisões arquiteturais que regularmente levam a projetos melhores
- Exemplo: as decisões abaixo têm garantido a disponibilização eficiente de serviços em sistemas distribuídos multi-usuário:
 - Separe fisicamente os componentes utilizados para requisitar daqueles utilizados para prover o serviço
 - Mantenha os provedores de serviço desconhecedores da identidade do requisitante
 - Requisitantes não devem ter contato uns com os outros
 - Permita que múltiplos provedores possam dinamicamente integrar o sistema

Estilo Arquitetural



- As decisões anteriores se aplicam a qualquer sistema neste contexto de disponibilização de serviços distribuídos
- Não são definidos detalhes acerca de componentes utilizados, suas interfaces e seus mecanismos de interação
- O arquiteto deve detalhar estas decisões e adaptá-las para o contexto específico de uma aplicação em particular
- Embora em alto nível de abstração, tais decisões definem o *rationale* subjacente à arquitetura

Estilo Arquitetural



Estilo Arquitetural: coleção identificada de decisões arquiteturais de projeto que: 1) são aplicáveis a um determinado contexto de desenvolvimento; 2) restringe as decisões arquiteturais específicas de um sistema em particular dentro deste contexto; e 3) induz qualidades benéficas nos sistemas resultantes

- O exemplo anterior é uma descrição informal e parcial do estilo arquitetural *Client-Server*
- Outros exemplos: *REST, Pipe-and-Filter*

Padrão Arquitetural



- Estilos arquiteturais definem decisões gerais de projeto que impõem restrições e que podem precisar ser detalhadas em decisões mais específicas para o sistema em questão
- Em contraste, os padrões arquiteturais definem decisões de projeto consideradas eficientes para certas classes de sistemas e que podem ser configurados com os componentes e conectores do sistema em questão

Padrão Arquitetural: coleção identificada de decisões arquiteturais de projeto que são aplicáveis a um problema recorrente de desenvolvimento e parametrizadas de modo a serem aplicadas em qualquer contexto de desenvolvimento de *software* no qual o problema aparece

Padrão Arquitetural



- As definições de estilo e padrão arquitetural são parecidas, nem sempre poderemos fazer a distinção com clareza
- Porém, estilos e padrões diferem em alguns aspectos
 - Escopo: estilos se aplicam a um contexto de desenvolvimento (ex: sistemas altamente distribuídos, sistemas *GUI-intensive*), enquanto padrões se aplicam a um problema de projeto específico (ex: o estado do sistema deve ser apresentado de múltiplas formas, a camada de negócio deve ser separada da camada de dados)
 - Um problema é mais concreto que um contexto
 - Estilos são estratégicos enquanto padrões são táticos

Padrão Arquitetural

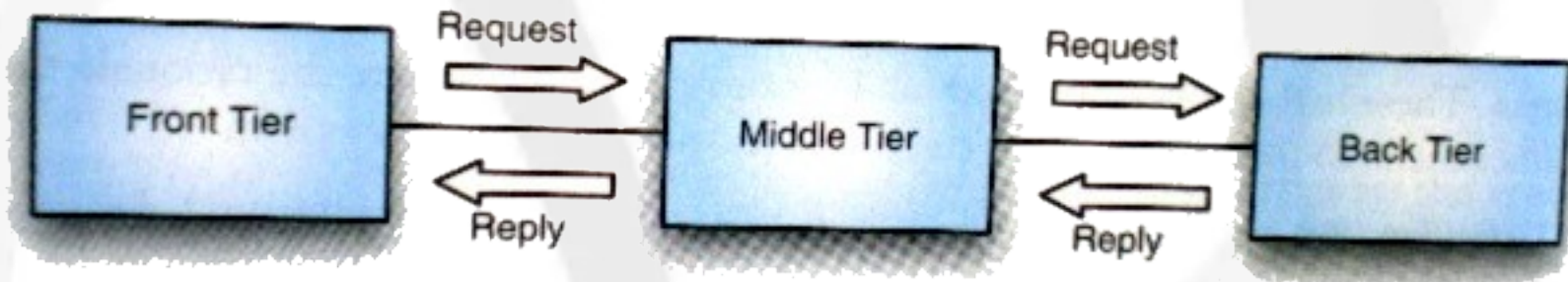


- Porém, estilos e padrões diferem em alguns aspectos
 - Abstração: um estilo ajuda a restringir as decisões arquiteturais do sistema. Requer intervenção humana para relacionar as diretrizes dadas pelo estilo com os problemas de projeto do sistema em questão
 - Por si só, são muito abstratos para já representar um projeto concreto do sistema
 - Padrões, por sua vez, são fragmentos arquiteturais parametrizados e podem ser considerados como peças concretas do projeto
 - Relacionamento: o mesmo padrão pode ser aplicado em sistemas que seguem diferentes estilos
 - Um sistema que segue um determinado estilo arquitetural pode envolver o uso de vários padrões arquiteturais

Padrão Arquitetural



- Exemplo de padrão arquitetural:
 - Sistema em três camadas



- *Front end (tier)*: contém as funcionalidades necessárias para acessar o serviço (GUI + cache + processamento mínimo)
- Camada de aplicação (*middle*): processa requisições do *front-end* e acessa e processa os dados do back-end
- *Back end (tier)*: contém as funcionalidades para armazenamento e acesso a dados
- Interações seguem o paradigma *request-reply*, porém nada além disso é prescrito (síncrono ? *request-triggered*? *single-request-single-reply*?)

Padrão Arquitetural



- Exemplo de padrão arquitetural:
 - Sistema em três camadas: parâmetros
 - Quais facilidades para interface de usuário, processamento, armazenamento e acesso a dados – específicos da aplicação – são necessários ?
 - Como elas devem ser organizadas dentro de cada camada ?
 - Quais mecanismos devem ser utilizados para permitir a interação entre as camadas ?

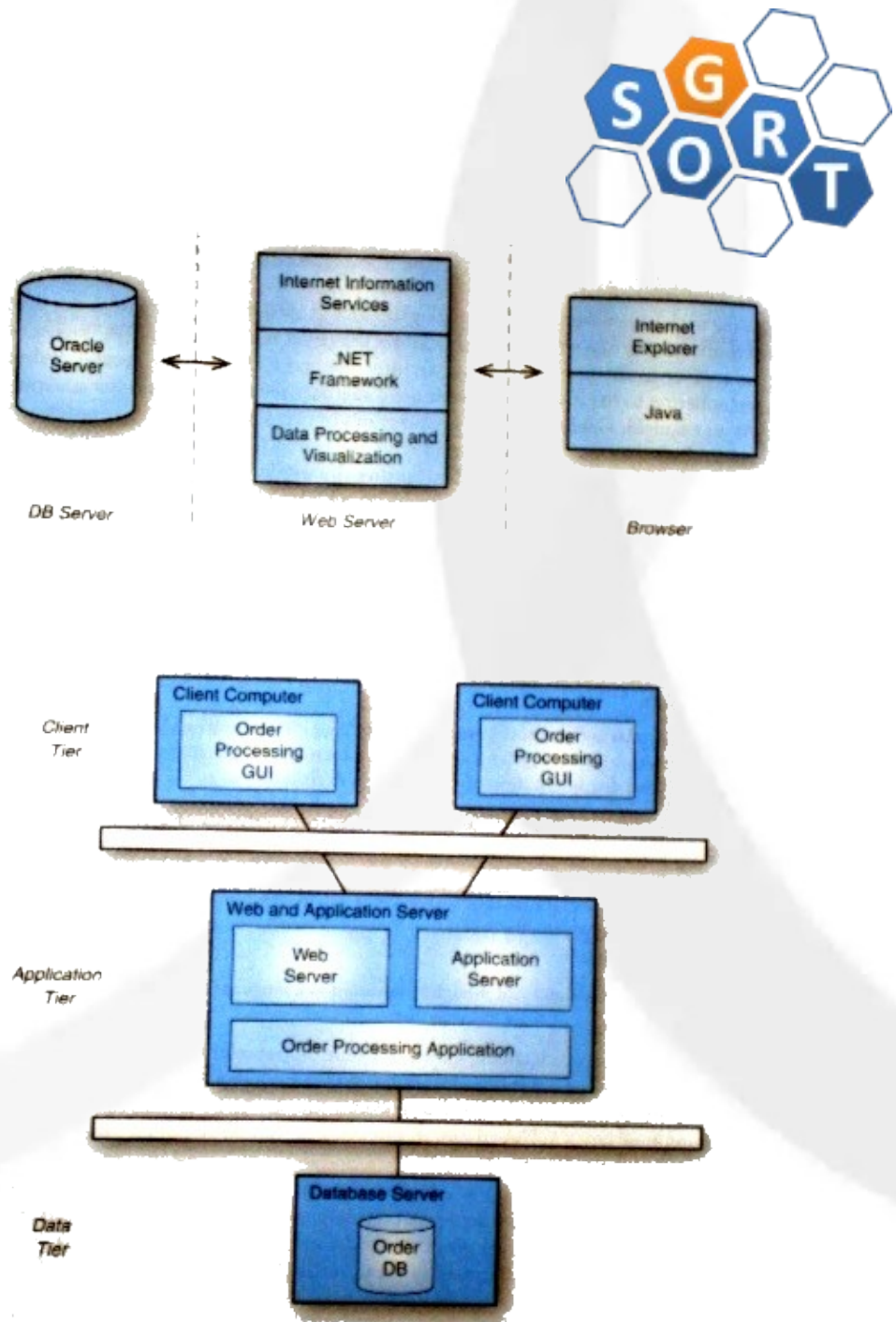
Padrões x Estilos



- Estilos demandam mais atenção do arquiteto e disponibilizam um suporte não tão direto quanto os padrões
- O padrão Sistema em Três Camadas pode ser visto como a sobreposição de duas arquiteturas que seguem o estilo *Client-Server*

Padrões x Estilos

- Dois sistemas em três camadas diferentes:
 - Quais traços arquiteturais são comuns ?
 - Quais são diferentes ?



Modelos Arquiteturais



- A arquitetura de um *software* é capturada em um modelo arquitetural, utilizando alguma notação de modelagem

Modelo Arquitetural: artefato que captura algumas ou todas as decisões de projeto que compõem a arquitetura do sistema

Modelagem Arquitetural: atividade que reifica e documenta estas decisões arquiteturais de projeto

- Um sistema pode ter diversos modelos associados
- Os modelos podem variar em relação à quantidade de detalhes capturados, perspectiva utilizada, tipo de notação, etc

Modelos Arquiteturais



Notação de Modelagem Arquitetural: é uma linguagem ou um meio de captura das decisões arquiteturais de projeto

- As notações podem ser textuais ou gráficas, informais (diagramas em slides), semi-formais (UML), formais (ADL's), de domínio específico ou propósito geral, proprietárias ou padronizadas, etc
- Modelos arquiteturais são artefatos críticos e servem como base para a realização das tarefas subsequentes

Recuperação Arquitetural



- Com degradações constantes chegará o momento onde mudanças adicionais no sistema se tornam inviáveis
- Os efeitos das mudanças também se tornam imprevisíveis visto que a arquitetura prescritiva está tão desatualizada que se torna inútil, podendo até atrapalhar o processo
- Realiza-se então a recuperação arquitetural:

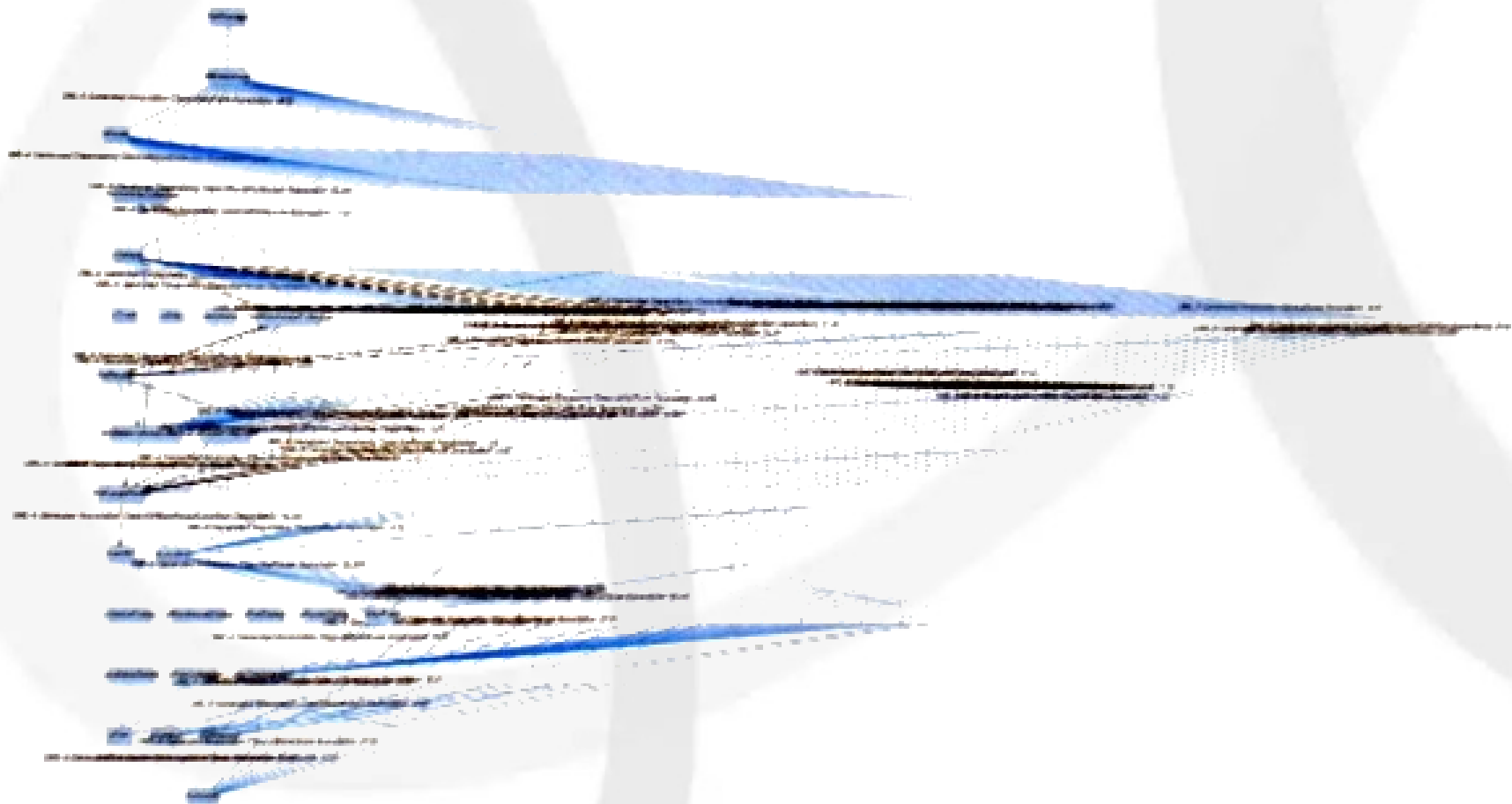
Recuperação Arquitetural: processo de determinação da arquitetura de um sistema a partir dos seus artefatos de implementação

- Artefatos = código-fonte, executáveis, *byte-codes*, etc

Recuperação Arquitetural



- Diagrama gerado diretamente do código-fonte:



Stakeholders



- Pessoas envolvidas e interessadas no projeto:
 - Arquiteto de *software*: define, modela, avalia e evolui a arquitetura do sistema. Mantém a integridade conceitual do sistema. É um stakeholder crítico
 - Desenvolvedores: consumidores primários dos produtos do arquiteto. Implementam as decisões de projeto presentes na arquitetura gerando a implementação do sistema
 - Gerente de *software*: supervisiona o projeto e apoia o arquiteto. Se necessário, exerce sua autoridade em nome do arquiteto
 - Clientes: desejam um sistema de alta qualidade, satisfazendo os requisitos no prazo e dentro dos custos estimados



Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Arquitetura de Software - Padrões e Estilos Arquiteturais

Sandro S. Andrade
sandroandrade@ifba.edu.br

Estilos e Padrões Arquiteturais

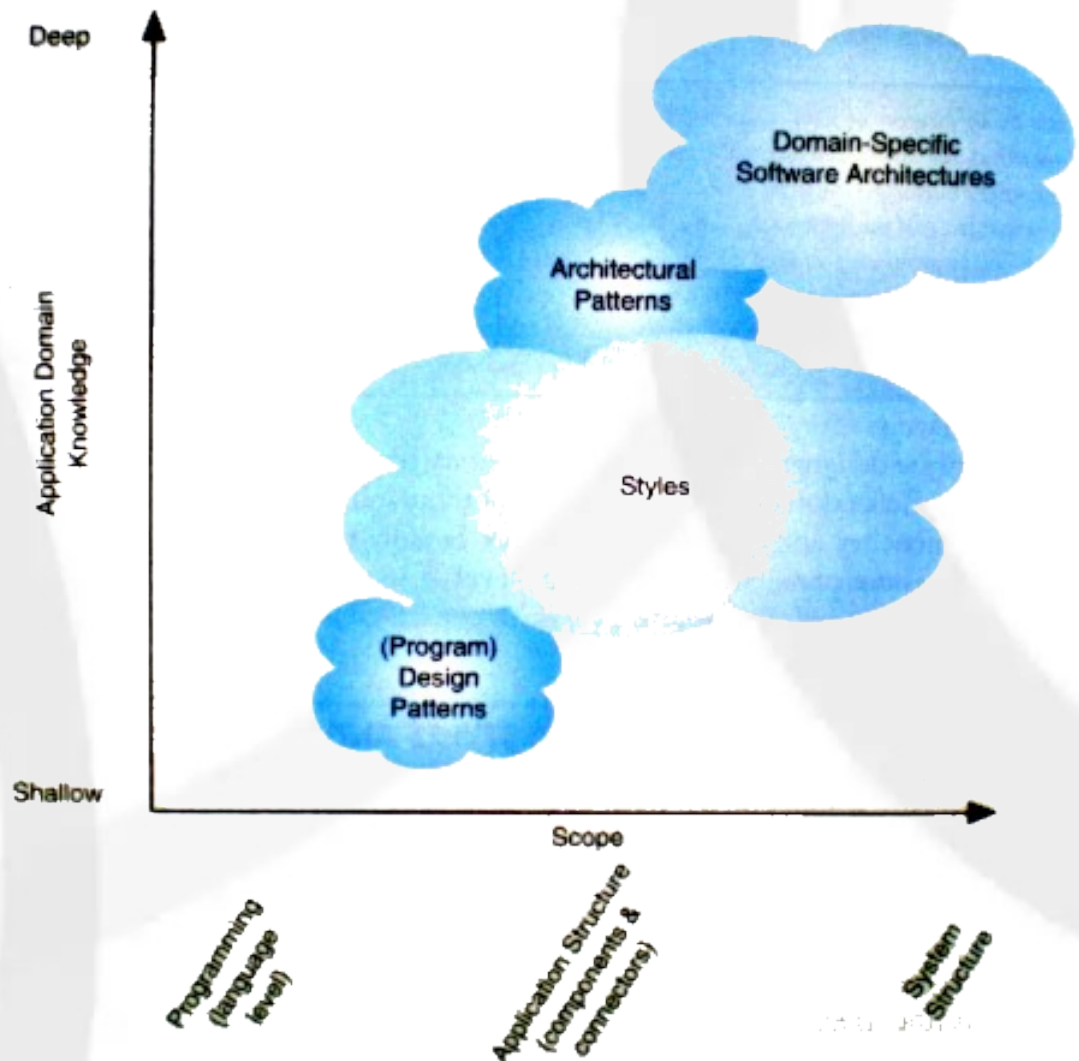


Estilos e padrões arquiteturais são projetados para capturar o conhecimento de projetos efetivos no alcance de metas específicas dentro de um contexto particular de aplicações

- Exemplo na construção civil:
 - Uma determinada combinação de metas e contexto pode demandar a construção de uma casa para uma família pequena numa região de clima mediterrâneo utilizando pedras e telhas como materiais básicos
 - O estilo apropriado seria *Single-Story Villa*
- Exemplo em software:
 - Metas e contexto demandam o desenvolvimento de um sistema de *instant messaging* operando entre sites remotos de uma empresa
 - O estilo apropriado seria *Client-Server*

Estilos e Padrões Arquiteturais

Estilos e padrões podem ser caracterizados tanto para problemas pequenos quanto mais complexos



Estilos e Padrões Arquiteturais



- As bordas do diagrama anterior não são precisamente definidas
- O eixo *scope* não é genuinamente linear ou totalmente ordenado
- O que um arquiteto denomina Padrão Arquitetural pode ser chamado de Estilo Arquitetural por outro

Padrões Arquiteturais



- Padrões arquiteturais são semelhantes às DSSAs, porém aplicados em um escopo bem mais específico

Padrão Arquitetural: coleção identificada de decisões arquiteturais de projeto que são aplicáveis a um problema recorrente de desenvolvimento e parametrizadas de modo a serem aplicadas em qualquer contexto de desenvolvimento de *software* no qual o problema aparece

Padrões Arquiteturais

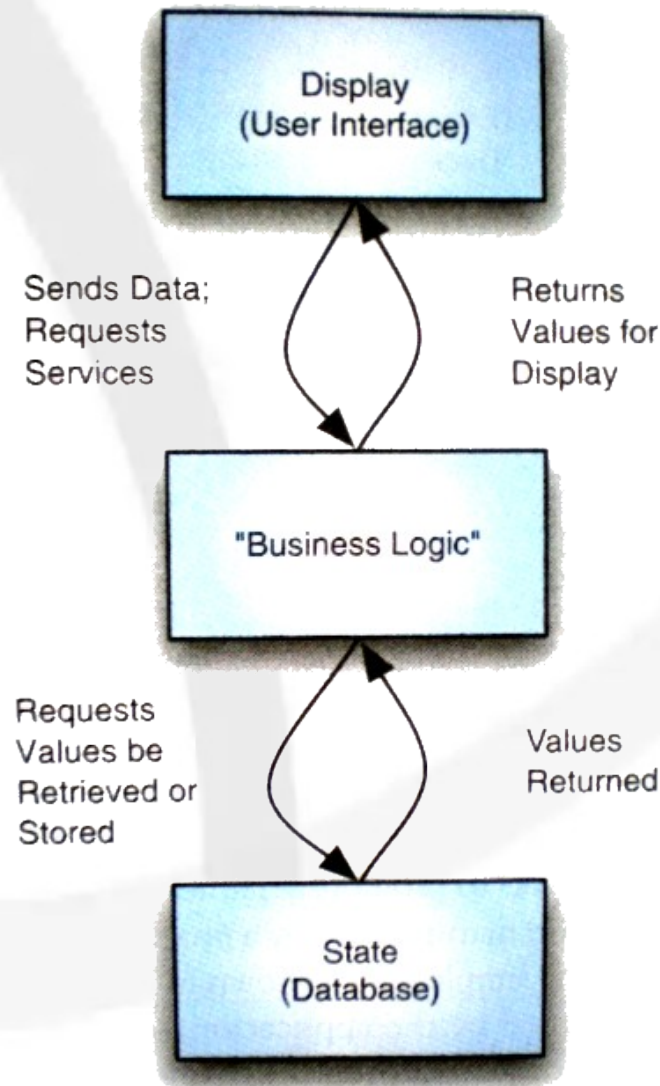
State-Logic-Display (Three-Tier)



- Comumente empregado em sistemas de informação:
 - *Data Store* + Lógica de Negócio + Interface de Usuário
- É facilmente mapeado em uma implementação distribuída com comunicação via *Remote Procedure Call*
- Jogos *multi-player* em rede
- Sistemas web

Padrões Arquiteturais

State-Logic-Display (Three-Tier)



Padrões Arquiteturais

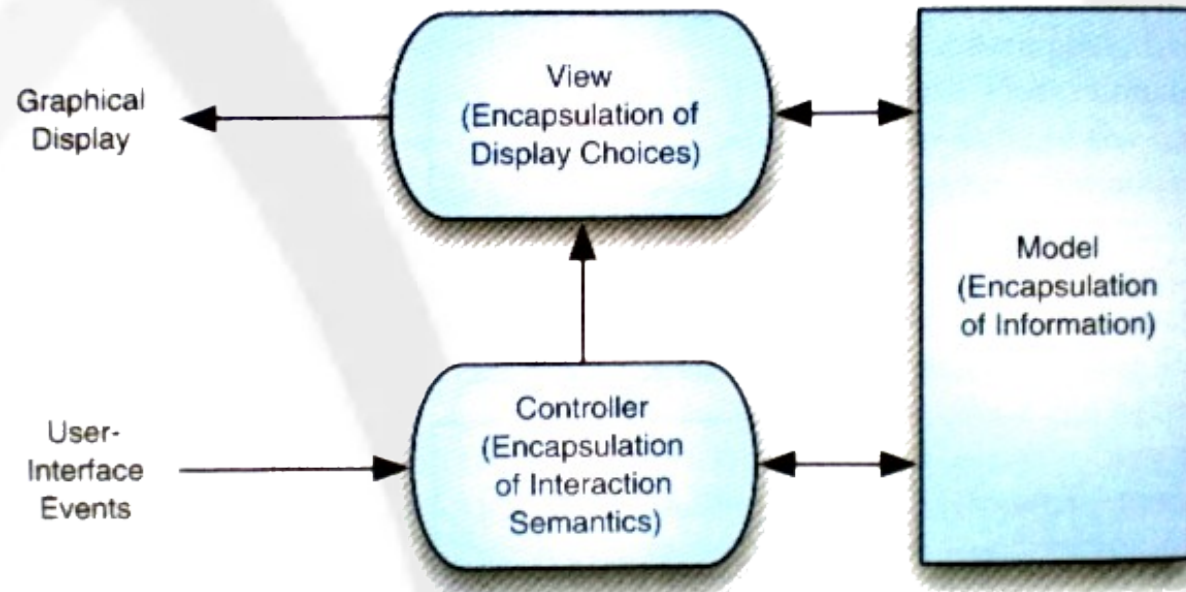
Model-View-Controller (MVC)



- Utilizado desde a década de 80 para o projeto de interfaces gráficas de usuário
- Pode também ser visto como um *design pattern*
- Promove a separação e, a consequente independência de desenvolvimento, da informação manipulada pelo programa e interações do usuário com esta informação

Padrões Arquiteturais

Model-View-Controller (MVC)



- *Model*: encapsula a informação usada pela aplicação
- *View*: encapsula artefatos necessários à descrição gráfica da informação
- *Controller*: encapsula a lógica necessária à manutenção da consistência entre o *model* e o *view*. É responsável pelo processamento dos eventos do usuário

Padrões Arquiteturais

Model-View-Controller (MVC)



- Colaborações entre os componentes (variações são consideradas na prática):
 - Quando a aplicação modifica um valor no model uma notificação é enviada à(s) *view(s)* de modo que a representação gráfica seja atualizada e re-exibida
 - Notificações também podem ser enviadas ao *controller*, que pode modificar a *view* se necessário
 - A *view* pode solicitar dados adicionais ao *model*
 - O sistema de janelas envia os eventos do usuário ao *controller* que pode consultar a *view*, obtendo informações que ajudam a determinar a ação a ser tomada
 - Como consequência o *controller* atualiza o *model*

Padrões Arquiteturais

Model-View-Controller (MVC)



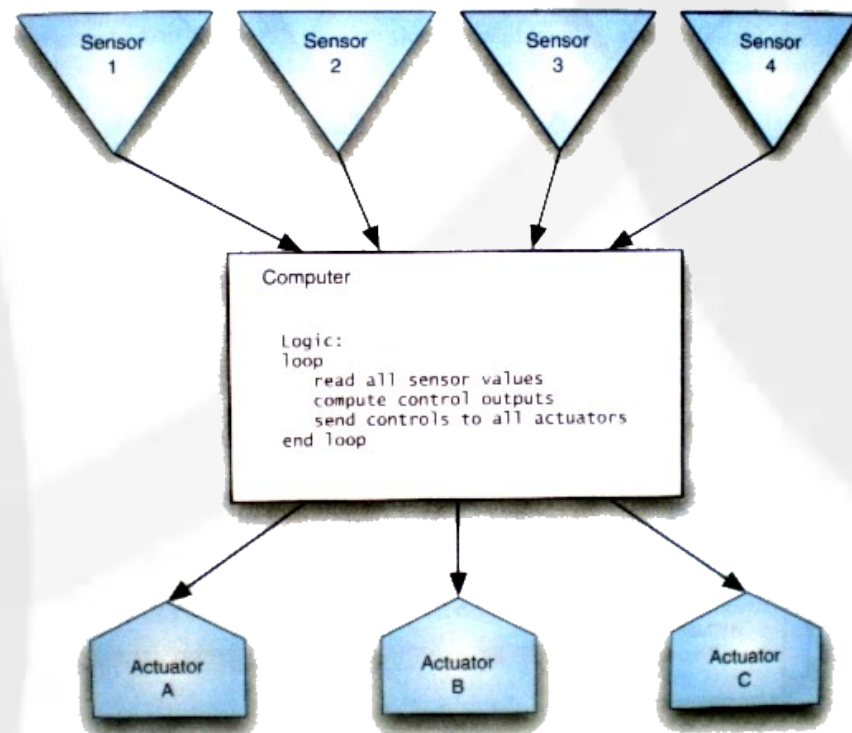
- Geralmente existe um acoplamento forte entre as ações da *view* e do *controller*, eventualmente justificando um *merge* destes componentes
- MVC na World Wide Web:
 - *Model*: recursos web
 - *View*: agente de renderização HTML do *browser*
 - *Controller*: parte do *browser* que responde aos eventos do usuário e que interaja com o servidor web ou modifica o que é exibido no *browser*. Pode também agregar código obtido do servidor web (ex: *JavaScript*)

Padrões Arquiteturais

Sense-Computer-Control



- Tipicamente utilizado na estruturação de aplicações embarcadas de controle:
 - Exemplos: forno de micro-ondas, DVD players, sistemas automotivos e robôs

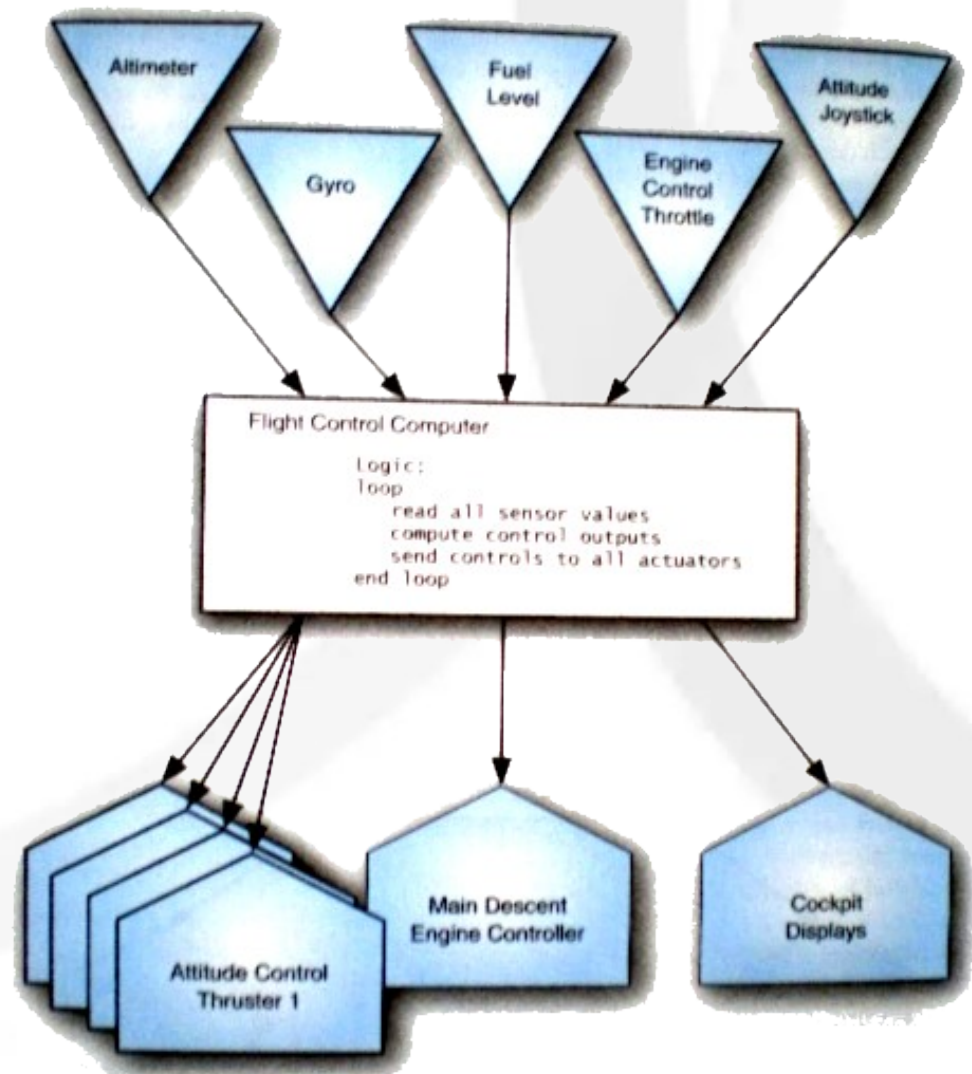


Padrões Arquiteturais

Sense-Computer-Control



- Jogo de pouso lunar:
 - *Feedback* implícito através do ambiente



Estilos Arquiteturais



- Principal forma de caracterizar experiências em projeto de *software*
- Elemento chave no desenvolvimento da concepção inicial ou detalhada da arquitetura do sistema
- São amplamente aplicados, refletindo menos conhecimento de domínio do que os padrões arquiteturais
- A fronteira entre estilos e padrões arquiteturais pode não ser clara, entretanto

Estilos Arquiteturais



Estilo Arquitetural: coleção identificada de decisões arquiteturais de projeto que: 1) são aplicáveis a um determinado contexto de desenvolvimento; 2) restringe as decisões arquiteturais específicas de um sistema em particular dentro deste contexto; e 3) induz qualidades benéficas nos sistemas resultantes

- Serão estudados:
 - As decisões e restrições que compõem o estilo arquitetural
 - As qualidades (benefícios) induzidas por estas decisões

Estilos Arquiteturais



- Estilos tradicionais influenciados por linguagens de programação
 - *Main Program and Subroutines*
 - *Object-Oriented*
- Estilos em Camadas:
 - *Virtual Machines*
 - *Client-Server*
- Estilos Baseados em Fluxo de Dados
 - *Batch-Sequential*
 - *Pipe-and-Filter*

Estilos Arquiteturais



- Estilos com Memória Compartilhada:
 - *Blackboard*
 - *Rule-Based / Expert System*
- Estilos Baseados em Interpretadores:
 - *Basic Interpreter*
 - *Mobile Code*
- Estilos Baseados em Invocação Implícita:
 - *Publish-Subscribe*
 - *Event-Based*
- *Peer-to-Peer*

Estilos Arquiteturais

Influenciados por Linguagens



- Linguagens tais como C, C++, Java e Pascal podem ser utilizadas para implementar arquiteturas de qualquer estilo
- Alguns estilos, entretanto, refletem os relacionamentos básicos de organização e controle de fluxo entre componentes disponibilizados por estas linguagens:
 - *Main Program and Subroutines*
 - *Object-Oriented*

Estilos Arquiteturais Influenciados por Linguagens



- *Main Program and Subroutines:*

Resumo: decomposição baseada na separação de passos funcionais de processamento

Componentes: programa principal e sub-rotinas

Conectores: *function/procedure calls*

Elementos de Dados: parâmetros e valores de retorno utilizados nas sub-rotinas

Topologia: organização estática e hierárquica de componentes; grafo direcionado

Restrições Adicionais: nenhuma

Qualidades Induzidas: modularidade – sub-rotinas podem ser substituídas por outras com implementação diferente, desde que a semântica da interface não mude

Usos Típicos: programas pequenos e de propósito pedagógico

Precauções: não é escalável para grandes aplicações; atenção inadequada às estruturas de dados; imprevisibilidade na determinação do esforço necessário para acomodar novas mudanças

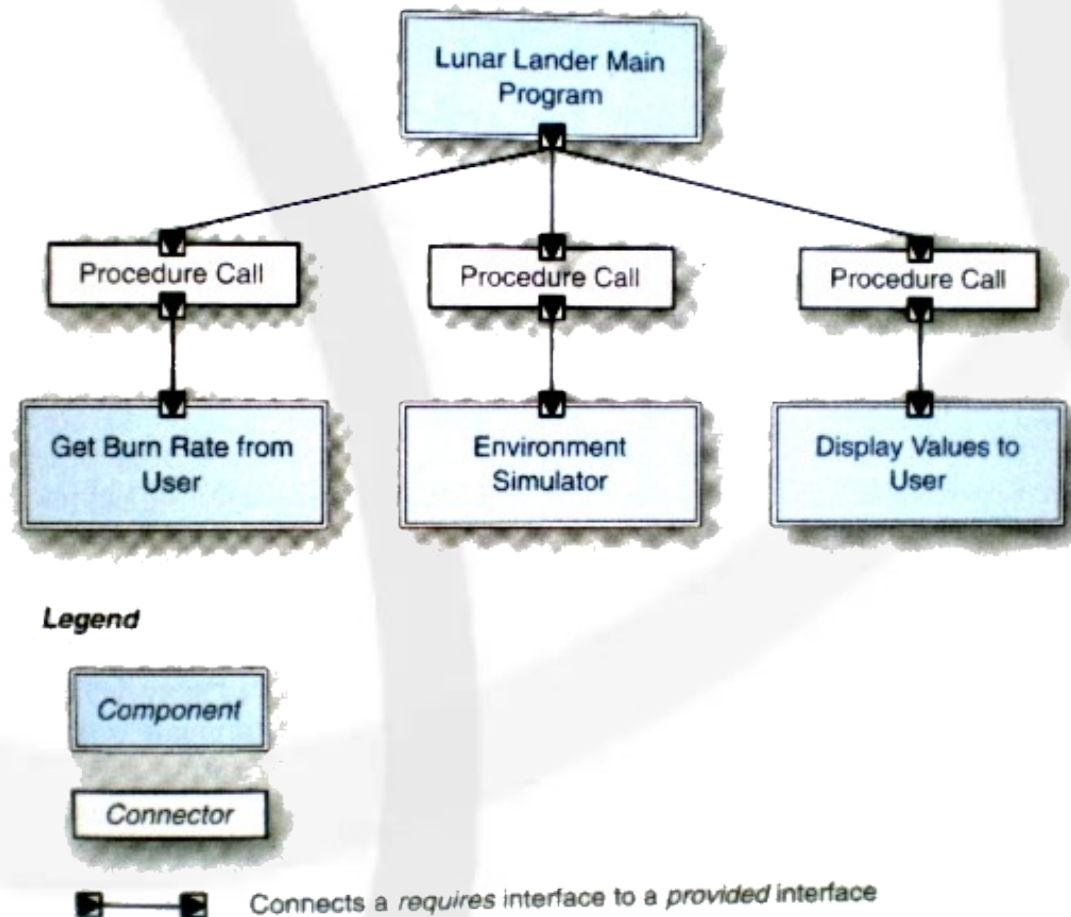
Relacionamento com Linguagens de Programação e Ambientes: linguagens de programação imperativas, tais como BASIC, Pascal ou C

Estilos Arquiteturais

Influenciados por Linguagens



- *Main Program and Subroutines* (pouso lunar):



Estilos Arquiteturais Influenciados por Linguagens



- *Object-Oriented (OO):*
 - A única estrutura disponibilizada é um conjunto de objetos cujo tempo de vida varia de acordo com os seus usos
 - Compreender um programa OO requer entender os numerosos relacionamentos estáticos e dinâmicos entre os objetos

Estilos Arquiteturais Influenciados por Linguagens



- *Object-Oriented (OO):*

Resumo: estado fortemente encapsulado com funções que operam neste estado, sob a forma de objetos. Objetos devem ser instanciados antes que seus métodos sejam invocados

Componentes: objetos (instâncias de uma classe)

Conectores: invocações de métodos (*procedure calls* que manipulam estado)

Elementos de Dados: argumentos de métodos

Topologia: pode variar arbitrariamente; componentes podem compartilhar dados e interfaces de funções através de hierarquias de herança

Restrições Adicionais: geralmente memória compartilhada (para ponteiros) e *single-threaded*

Qualidades Induzidas: integridade de operações nos dados – dados são manipulados somente por funções apropriadas. Abstração – detalhes de implementação estão ocultos

Usos Típicos: quando deseja-se um relacionamento forte entre entidades do mundo físico e do programa; propósitos pedagógicos; sistemas com estruturas de dados complexas e dinâmicas

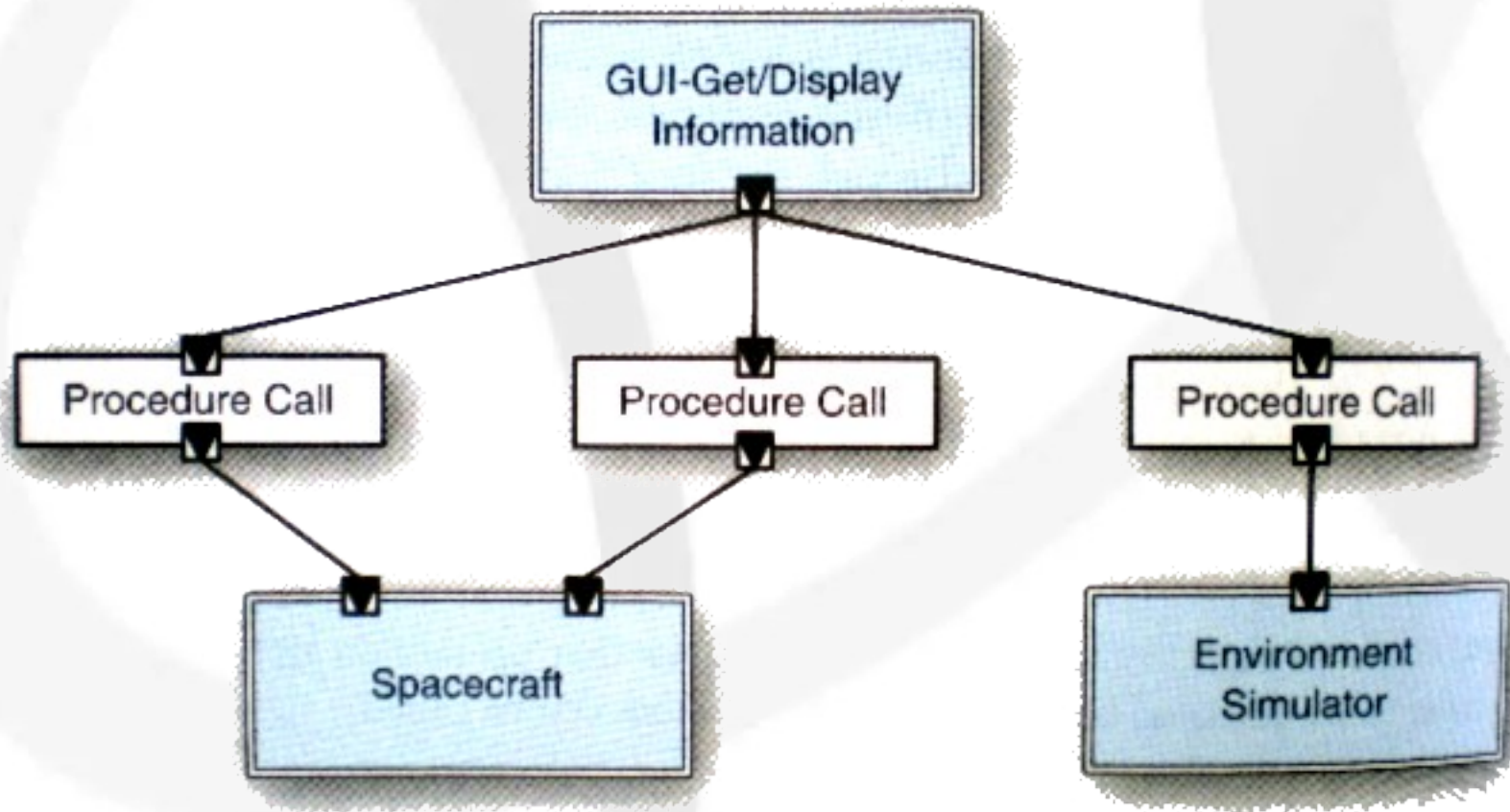
Precauções: uso em sistemas distribuídos requer alguma solução de *middleware*; relativamente ineficiente para aplicações de alto-desempenho com grandes estruturas de dados; ausência de princípios estruturantes adicionais pode resultar em aplicações altamente complexas

Relacionamento com Linguagens de Programação e Ambientes: Java, C++

Estilos Arquiteturais Influenciados por Linguagens



- *Object-Oriented (OO)* (pouso lunar):

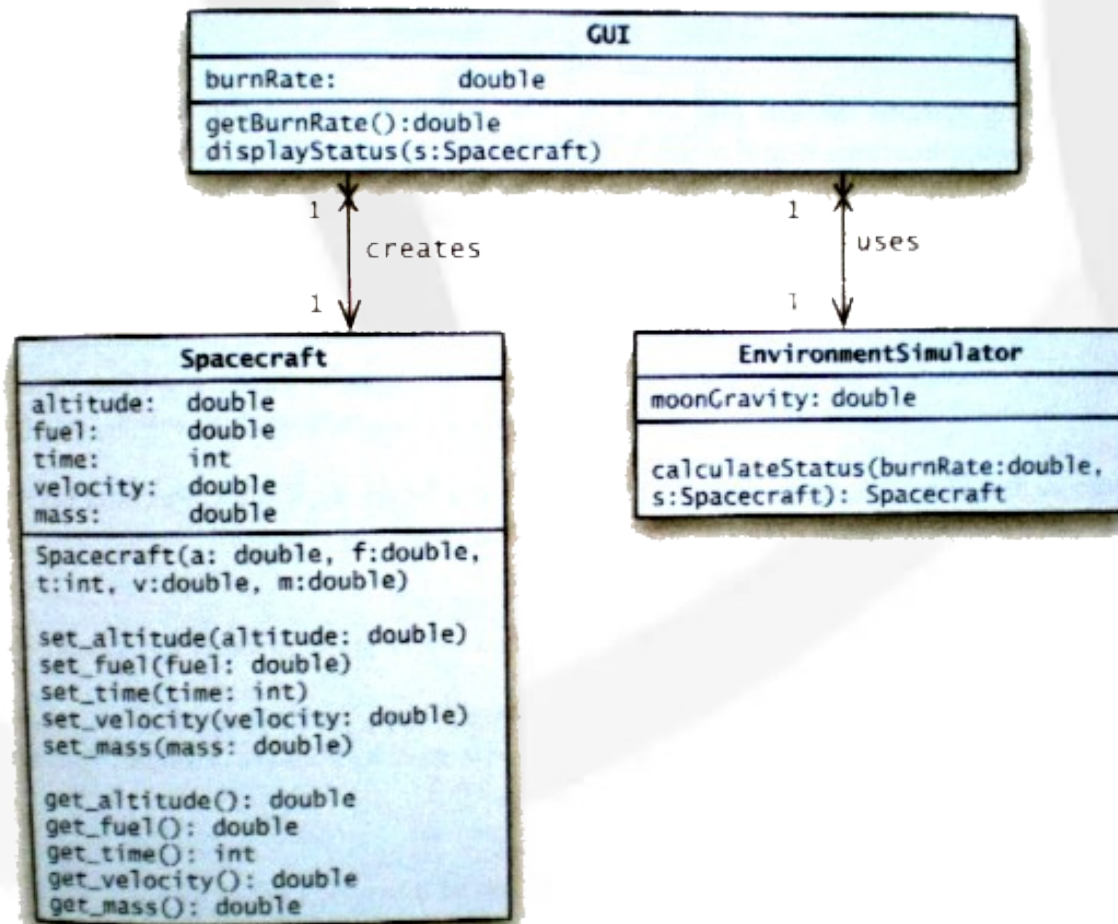


Estilos Arquiteturais

Influenciados por Linguagens



- *Object-Oriented (OO)* (pouso lunar):



Estilos Arquiteturais

Em Camadas



- A arquitetura é separada em camadas ordenadas, onde um programa de uma camada pode solicitar serviços de uma camada inferior
- Exemplos:
 - Arquiteturas de sistemas operacionais
 - *Client-Server*

Estilos Arquiteturais

Em Camadas

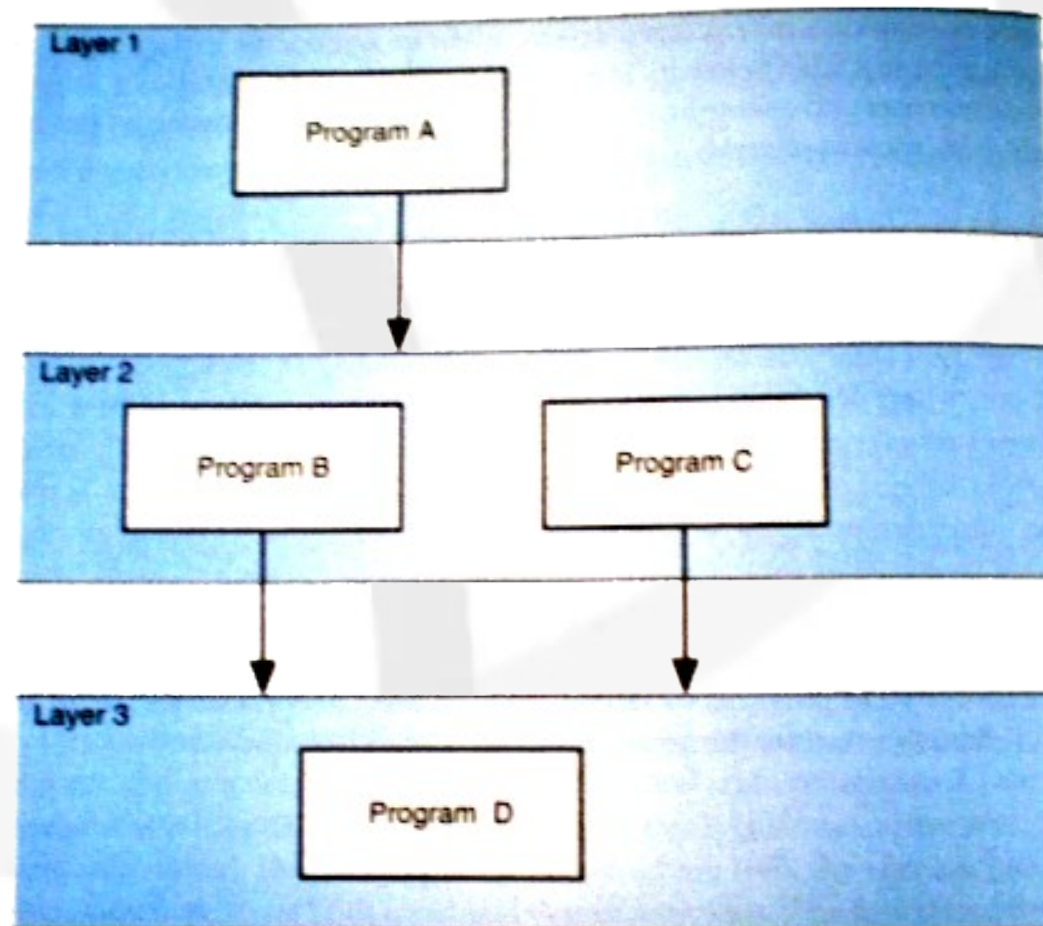


- *Virtual Machines:*
 - Uma camada oferece um conjunto de serviços (*provides interfaces*) que podem ser utilizados por programas que residem na(s) camada(s) acima
 - Os serviços podem ser implementados por diversos programas dentro da camada porém, para os clientes destes serviços, tal distinção não é aparente

Estilos Arquiteturais Em Camadas



- *Virtual Machines* (exemplo):



Estilos Arquiteturais Em Camadas



- *Virtual Machines* (classificação):
 - Máquina Virtual Estrita: programas de um determinado nível somente podem acessar serviços providos pela camada imediatamente inferior
 - Máquina Virtual Não-Estrita: programas de um determinado nível podem acessar serviços de qualquer camada abaixo do nível em questão
- Exemplo 1: camadas de um sistema operacional:
 - Aplicações do usuário (nível 1)
 - Serviço de manipulação de arquivos e diretórios (nível 2)
 - *Drivers* de disco e gerenciamento de volume (nível 3)
- Exemplo 2: protocolos de comunicação em rede

Estilos Arquiteturais

Em Camadas



- *Virtual Machines:*

Resumo: sequência ordenada de camadas; cada camada (ou máquina virtual) oferece um conjunto de serviços que podem ser acessados por programas (sub-componentes) de uma camada acima

Componentes: camadas oferecendo serviços para outras camadas, tipicamente compostas de vários programas (sub-componentes)

Conectores: tipicamente *procedure calls*

Elementos de Dados: parâmetros que transitam entre as camadas

Topologia: linear para máquinas virtuais estritas e grafo direcionado acíclico em interpretações mais fracas

Restrições Adicionais: nenhuma

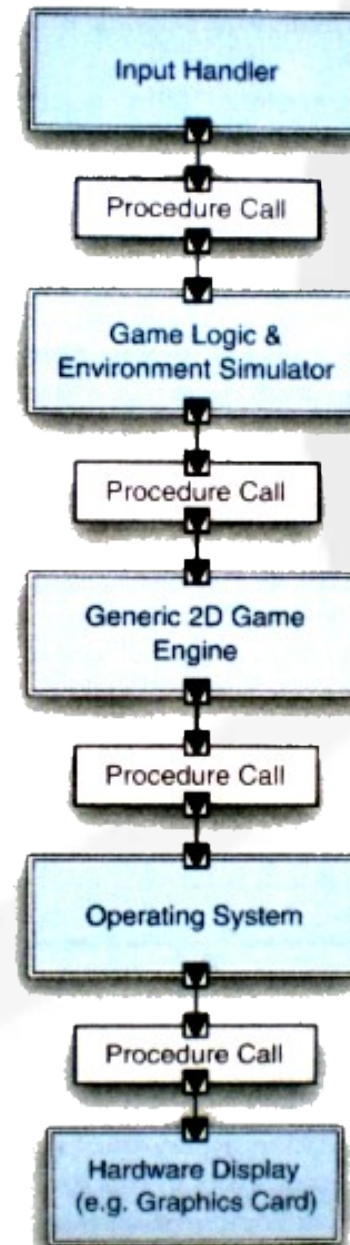
Qualidades Induzidas: estrutura de dependência clara; componentes em uma camada superior são imunes a modificações das camadas inferiores desde que as especificações do serviço não mudem; componentes em uma camada inferior são totalmente independentes de camadas superiores

Usos Típicos: projeto de sistemas operacionais, pilhas de protocolos de rede

Precauções: máquinas virtuais estritas com muitos níveis podem ser relativamente ineficientes

Estilos Arquiteturais Em Camadas

- *Virtual Machines* (pouso lunar):



Estilos Arquiteturais

Em Camadas



- *Client-Server*:
 - Máquina Virtual de duas camadas com conexões em rede
 - O servidor é a máquina virtual abaixo dos clientes
 - Múltiplos clientes podem acessar o servidor
 - Os clientes são independentes
 - Pode-se utilizar *thin* (*thick*) clients
 - Exemplo: máquina de saque eletrônico (ATM)

Estilos Arquiteturais

Em Camadas



- *Client-Server.*

Resumo: clientes enviam requisições de serviço ao servidor, que realiza as operações requeridas e responde conforme necessário com as informações solicitadas. Toda comunicação é iniciada pelos clientes

Componentes: clientes e servidor

Conectores: *remote procedure calls*; protocolos de rede

Elementos de Dados: parâmetros e valores de retorno conforme enviados pelos conectores

Topologia: em dois níveis, com múltiplos clientes realizando requisições ao servidor

Restrições Adicionais: não existe comunicação entre clientes

Qualidades Induzidas: centralização da computação e dos dados no servidor, que torna a informação disponível para os clientes. Um único servidor poderoso pode servir muitos clientes

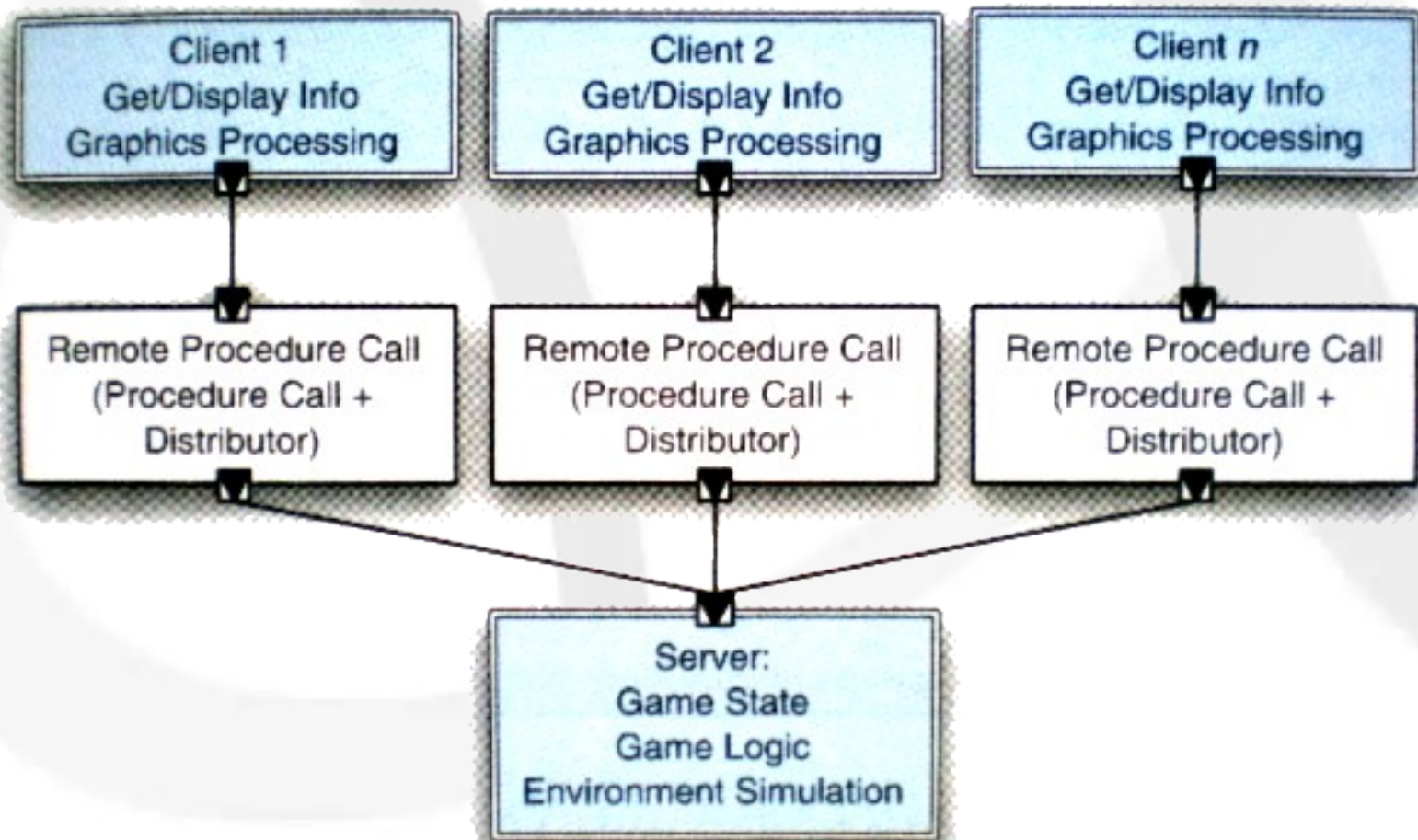
Usos Típicos: onde é necessário centralização de dados; onde processamento e armazenamento de dados se beneficiam de uma máquina de alta capacidade; e onde clientes realizam geralmente tarefas simples de interface de usuário, tais como em diversos sistemas de informação

Precauções: quando a largura de banda é limitada e existe um grande número de clientes

Estilos Arquiteturais Em Camadas



- *Client-Server* (pouso lunar):



Estilos Arquiteturais Baseados em Fluxos de Dados



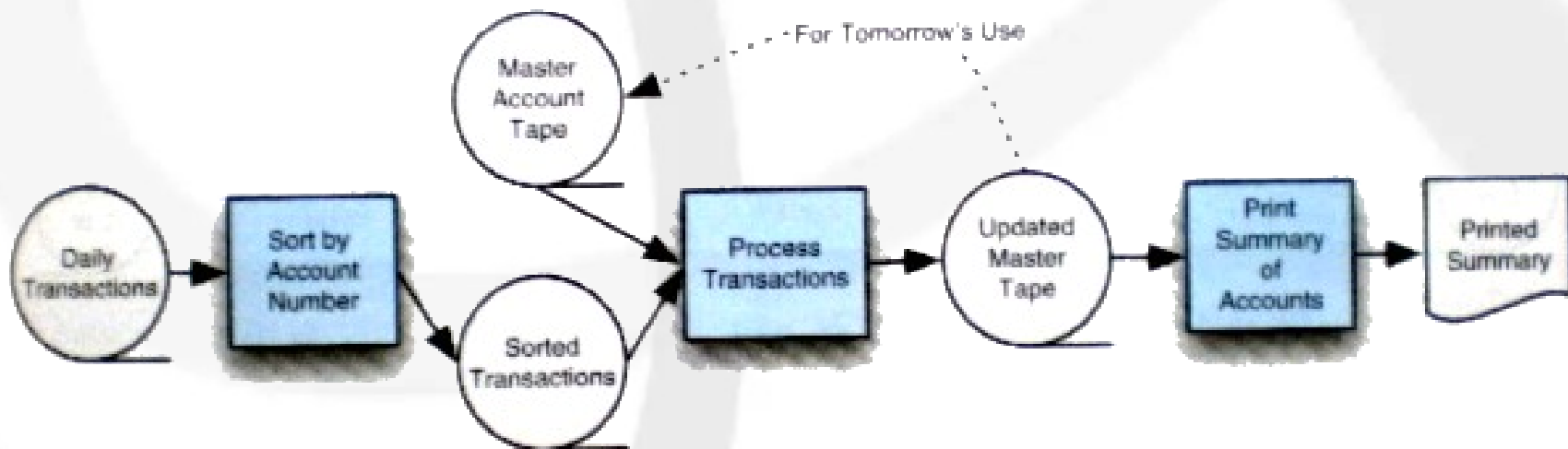
- Caracterizado pelo movimento de dados entre elementos independentes de processamento

Estilos Arquiteturais

Baseados em Fluxos de Dados



- *Batch-Sequential:*
 - Um dos estilos arquiteturais mais antigos: as limitações dos equipamentos exigiam que o problema fosse sub-dividido em componentes que se comunicavam através da transferência de fitas magnéticas
 - Exemplo: atualizar um registro bancário de todas as contas



Estilos Arquiteturais

Baseados em Fluxos de Dados



- *Batch-Sequential:*

Resumo: programas distintos são executados em ordem; os dados são passados, sob a forma de blocos (agregados), de um programa para o próximo

Componentes: programas independentes

Conectores: mãos humanas que carregam as fitas entre os programas (*sneaker-net*)

Elementos de Dados: elementos agregados explícitos que, após o término da execução de um componente, são repassados deste para o próximo componente

Topologia: linear

Restrições Adicionais: um único programa executa por vez até o seu término

Qualidades Induzidas: execuções separáveis; simplicidade

Usos Típicos: processamento de transações em sistemas financeiros

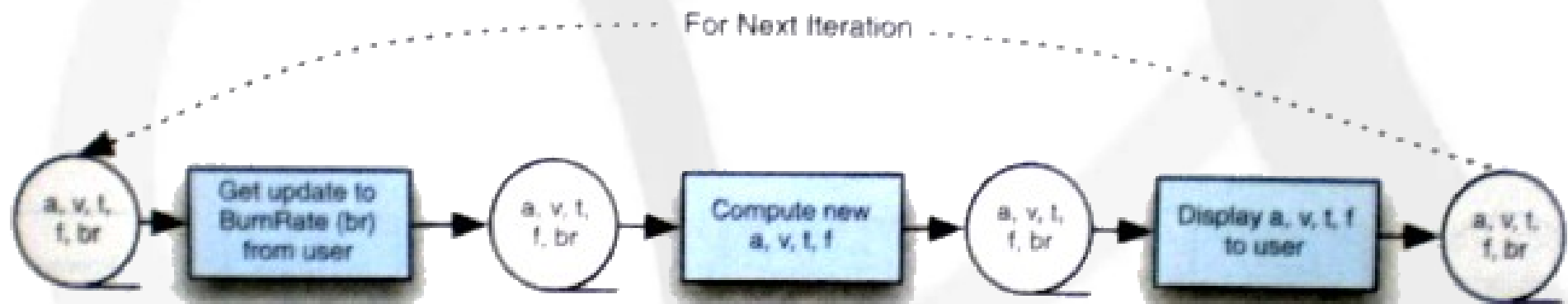
Precauções: quando interação entre componentes é requerida; quando concorrência entre componentes é possível ou necessário

Relacionamento com Linguagens de Programação e Ambientes: nenhum

Estilos Arquiteturais Baseados em Fluxos de Dados



- *Batch-Sequential* (pouso lunar):
 - Indica quão inapropriado é este estilo para a aplicação



Estilos Arquiteturais Baseados em Fluxos de Dados



- *Pipe-and-Filter.*
 - Os filtros podem operar de forma concorrente, não é necessário aguardar o término do produtor para que o componente que consome a saída do produtor inicie o seu funcionamento

Estilos Arquiteturais Baseados em Fluxos de Dados



- *Pipe-and-Filter.*

Resumo: programas distintos são executados, potencialmente de forma concorrente; os dados são passados, sob a forma de fluxo, de um programa para o próximo

Componentes: programas independentes (filtros)

Conectores: roteadores explícitos de fluxos de dados (*pipes*); possivelmente é um serviço disponibilizado pelo sistema operacional ou pela linguagem de programação

Elementos de Dados: não definidos explicitamente, porém devem ser *streams* lineares de dados

Topologia: *pipeline*, embora bifurcações sejam possíveis

Qualidades Induzidas: filtros são mutualmente independentes. Estruturas simples de chegada e saída de fluxos de dados facilitam novas combinações de filtros

Usos Típicos: programação de aplicações baseadas em primitivas de sistemas operacionais

Precauções: quando estruturas complexas de dados precisam ser transferidas entre componentes; quando interatividade entre os programas é necessário

Relacionamento com Linguagens de Programação e Ambientes: Unix shell

Estilos Arquiteturais

Com Memória Compartilhada



- Caracterizado pela presença de múltiplos componentes que acessam o mesmo repositório de dados (*data store*) e se comunicam através deste repositório
- Semelhante ao uso de dados globais porém, nestes estilos, o centro de atenção no projeto é explicitamente direcionado para este repositório
- O repositório é bem ordenado e cuidadosamente gerenciado

Estilos Arquiteturais Com Memória Compartilhada



- *Blackboard*:
 - Tem sua origem nas aplicações de Inteligência Artificial
 - Analogia:
 - Diversos peritos (*experts*) sentados ao redor de uma mesa (*data store*) tentando cooperar na solução de um problema grande e complexo
 - Quando um perito reconhece que pode resolver alguma parte do problema que está na mesa ele recolhe este sub-problema, vai embora e trabalha na sua solução
 - Quando concluída a solução, o perito retorna e disponibiliza a solução na mesa
 - A disponibilização desta solução pode habilitar outro perito a resolver uma outra parte do problema
 - O processo continua até que todo o problema esteja resolvido

Estilos Arquiteturais Com Memória Compartilhada



- *Blackboard:*

Resumo: programas independentes acessam e se comunicam exclusivamente através de um repositório global de dados, conhecido como *blackboard*

Componentes: programas independentes (*knowledge sources*) e *blackboard*

Conectores: acesso ao *blackboard* pode ser através de referência direta a memória, *procedure call* ou uma consulta em um banco de dados

Elementos de Dados: dados armazenados no *blackboard*

Topologia: em estrela, com o *blackboard* no centro

Variações: 1) programas consultam o *blackboard* para verificar se algum valor de seu interesse mudou; 2) um *blackboard manager* notifica atualizações do *blackboard* aos componentes interessados

Qualidades Induzidas: estratégias completas de solução para problemas complexos não precisam ser pré-planejadas, pois são determinadas por visões, em constante mudança, dos dados/problema

Usos Típicos: solução heurística de problemas em aplicações de Inteligência Artificial

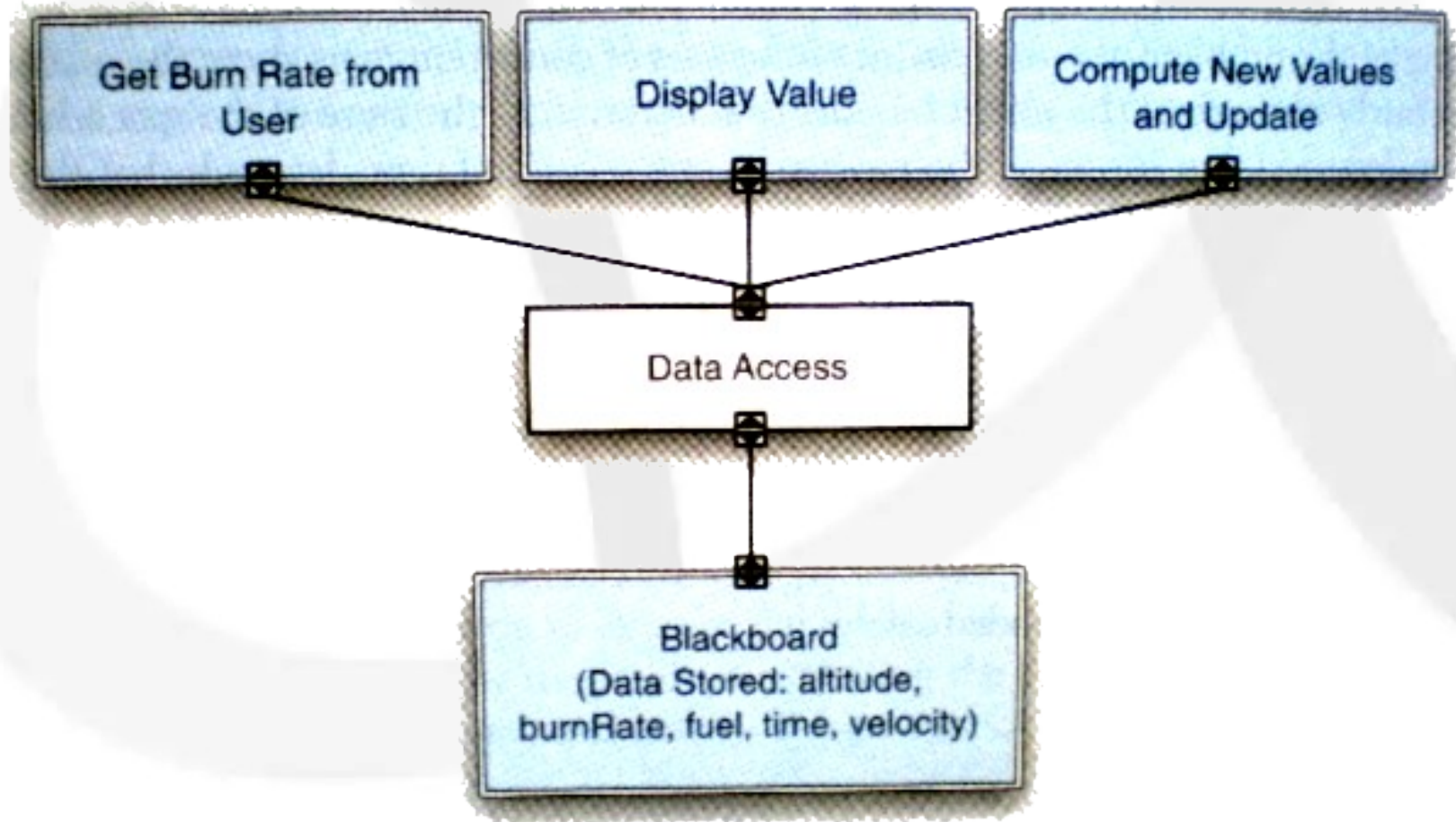
Precauções: quando existe uma estratégia bem-estruturada de solução; quando as interações entre os programas independentes requerem coordenações complexas; quando as representações dos dados do *blackboard* mudam frequentemente (requerendo modificações em todos os participantes)

Relacionamento com Linguagens de Programação e Ambientes: versões que permitem concorrência entre os programas requerem primitivas para gerenciar o *blackboard* compartilhado

Estilos Arquiteturais Com Memória Compartilhada



- *Blackboard* (pouso lunar):



Estilos Arquiteturais Com Memória Compartilhada



- *Rule-Based / Expert System:*
 - Tipo altamente especializado de arquitetura com memória compartilhada
 - A memória compartilhada funciona como uma base de conhecimento que contém fatos e regras de produção (cláusulas *if...then* sobre o conjunto de variáveis)
 - A interface gráfica de usuário disponibiliza duas operações básicas:
 - Entrada de novos fatos e regras de produção
 - Entrada de consultas (*goals*)
 - Uma máquina de inferência opera na base de conhecimento em resposta às entradas do usuário

Estilos Arquiteturais Com Memória Compartilhada



- *Rule-Based / Expert System:*

Resumo: a máquina de inferência analisa a entrada do usuário e determina se é um fato/regra ou consulta. Se for um fato/regra, esta entrada é adicionada à base de conhecimento. Caso contrário, é realizada uma consulta à base, buscando regras aplicáveis, com o objetivo de resolver a consulta

Componentes: interface de usuário, máquina de inferência e base de conhecimento

Conectores: componentes são fortemente inter-conectados com *procedure calls* ou *data access*

Elementos de Dados: fatos e consultas

Topologia: três camadas fortemente acopladas

Qualidades Induzidas: o comportamento da aplicação pode ser facilmente modificado através da adição ou remoção dinâmica de regras na base de conhecimento; sistemas pequenos podem ser rapidamente prototipados; útil para explorar iterativamente problemas cuja abordagem para uma solução genérica não é clara

Usos Típicos: quando o problema pode ser visto como uma resolução sucessiva de predicados

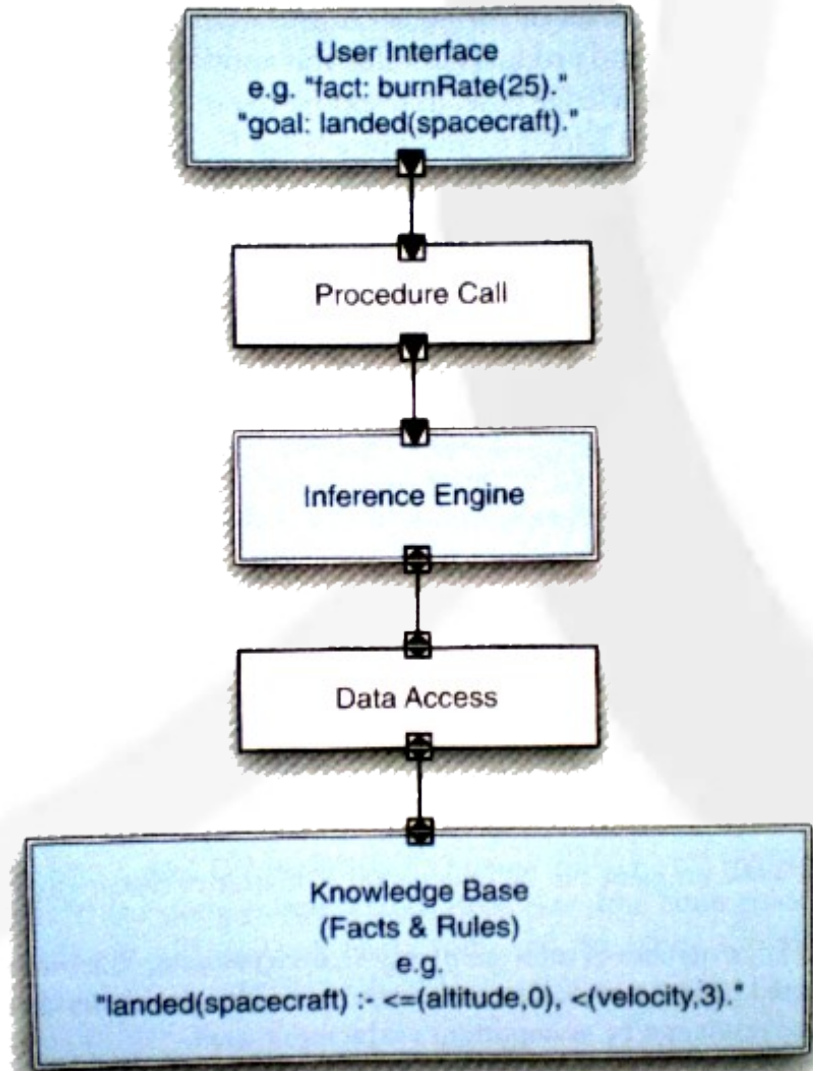
Precauções: quando muitas regras estão envolvidas pode ser difícil entender as interações entre múltiplas regras afetadas pelos mesmos fatos; entender a base lógica do resultado gerado pode ser tão importante quanto o próprio resultado

Relacionamento com Linguagens de Programação e Ambientes: Prolog é comumente utilizado para construir sistemas baseados em regras

Estilos Arquiteturais Com Memória Compartilhada



- *Rule-Based / Expert System*
(pouso lunar):



Estilos Arquiteturais

Baseados em Interpretadores



- Caracterizado pela interpretação dinâmica, *on-the-fly*, de comandos
- Comandos são sentenças explícitas, possivelmente criados momentos antes da sua execução, geralmente representados por um texto que pode ser compreendido e editado por humanos
- Comandos são construídos a partir de um comandos primitivos pré-definidos

Estilos Arquiteturais

Baseados em Interpretadores



- Execução:
 - 1) Inicia-se no estado inicial de execução
 - 2) Obtém-se o primeiro (próximo) comando a ser executado
 - 3) Executa-se o comando sobre o estado atual
 - 4) Avança-se para um novo estado
 - 5) Procede-se à execução do próximo comando (*goto 2*)
- A identificação do próximo comando pode ser afetada pelo resultado da execução do comando anterior

Estilos Arquiteturais Baseados em Interpretadores



- *Basic Interpreter.*
 - Similar ao Baseado em Regras / Sistema Especialista porém utiliza um interpretador de comandos no lugar da máquina de inferência
 - Este interpretador executa comandos mais genéricos (do que inferências em regras) e a interpretação de um único comando pode envolver várias operações primitivas
 - A base de conhecimento é similarmente mais genérica visto que estruturas de dados arbitrárias podem estar envolvidas
 - Exemplos: fórmulas de uma planilha de cálculo são interpretadas pela máquina de execução (interpretador) do sistema de planilhas; máquinas CNC; programação de trajetórias de robôs

Estilos Arquiteturais Baseados em Interpretadores



- *Basic Interpreter.*

Resumo: o interpretador analisa e executa comandos de entrada, atualizando o estado por ele mantido

Componentes: interpretador de comandos, estado do programa/interpretador e interface de usuário

Conectores: tipicamente o interpretador de comandos, interface de usuário e estado são fortemente acoplados via *procedure calls* ou *shared state*

Elementos de Dados: comandos

Topologia: três camadas altamente acopladas; o estado pode estar separado do interpretador

Qualidades Induzidas: possibilidade de comportamentos altamente dinâmicos, onde o conjunto de comandos é dinamicamente modificado; a arquitetura do sistema permanece inalterada enquanto novas funcionalidades são criadas com base nas primitivas existentes

Usos Típicos: excelente para permitir a programação pelo usuário final; para suportar mudança dinâmica do conjunto de funcionalidades

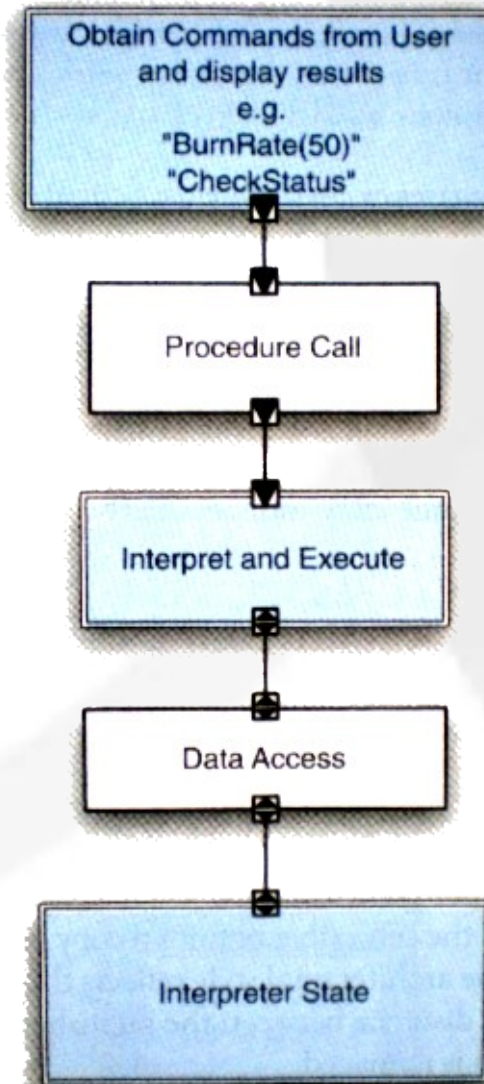
Precauções: quando processamento rápido é necessário (código interpretado é executado de forma mais lenta que código compilado); gerenciamento de memória pode se tornar um problema, especialmente quando múltiplos interpretadores são simultaneamente executados

Relacionamento com Linguagens de Programação e Ambientes: *Lisp* e *Scheme* são linguagens interpretadas e são eventualmente utilizadas para construir outros interpretadores; macros

Estilos Arquiteturais Baseados em Interpretadores



- *Basic Interpreter* (pouso lunar):



Estilos Arquiteturais Baseados em Interpretadores



- *Mobile Code:*
 - Permite que um código seja transmitido a um host remoto e por este host interpretado
 - Um elemento de dado (representação de um programa) é dinamicamente transformado em um componente de processamento de dados
- Motivações para a transmissão: falta de poder computacional, falta de recursos ou conjunto extenso de dados localizados remotamente
- Classificações:
 - *Code on Demand*
 - *Remote Evaluation*
 - *Mobile Agent*

Estilos Arquiteturais Baseados em Interpretadores



- *Mobile Code:*
 - *Code on Demand:*
 - Possui recursos e estado porém o código é obtido de um host remoto e executado localmente
 - *Remote Evaluation:*
 - Possui o código mas não os recursos para execução do código (ex: o interpretador)
 - O código é transmitido a um host remoto para processamento (ex: *grid*) e os resultados enviados de volta
 - *Mobile Agent:*
 - Possui o código e o estado mas parte dos recursos estão em outro *host*
 - O código + estado + alguns recursos (*agent*) são transferidos para o *host* remoto
 - Os resultados não necessariamente precisam ser enviados de volta ao *host* original

Estilos Arquiteturais Baseados em Interpretadores



- *Mobile Code:*

Resumo: o código se desloca com o objetivo de ser interpretado em outro *host*; dependendo da variação do estilo o estado pode também se deslocar

Componentes: doca de execução (que trata o recebimento e implantação do código e do estado) e o interpretador/compilador de código

Conectores: elementos e protocolos de rede que empacotam código e dados para fins de transmissão

Elementos de Dados: representações de código sob a forma de dados; estado do programa e dados

Topologia: em rede

Variações: *code on demand*, *remote evaluation* e *mobile agent*

Qualidades Induzidas: adaptabilidade dinâmica; se beneficia do poder computacional agregado nos *hosts* disponíveis; *dependability* melhorada em função da provisão de migração para novos *hosts*

Usos Típicos: quando deseja-se processar um conjunto extenso de dados localizados remotamente; quando deseja-se configurar dinamicamente um nó através da inclusão de código externo

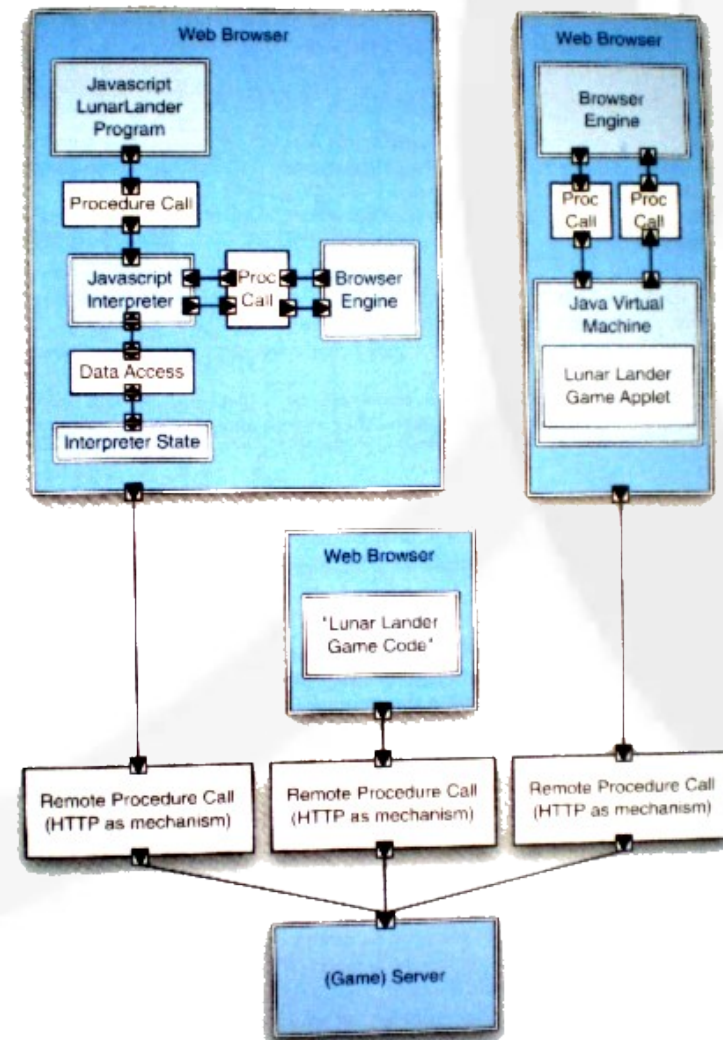
Precauções: aspectos de segurança (códigos maliciosos); quando precisa-se alto controle sobre as diferentes versões do *software* implantadas; quando o custo de transmissão é maior que o de processamento; quando as conexões de rede não são confiáveis

Relacionamento com Linguagens de Programação e Ambientes: linguagens de *scripting* (ex: *JavaScript*); computação em *grid*

Estilos Arquiteturais Baseados em Interpretadores



- *Mobile Code* (pouso lunar):



Estilos Arquiteturais

Baseados em Invocação Implícita



- Caracterizados por chamadas que são invocadas indiretamente e implicitamente em resposta a uma notificação ou a um evento
- Esta interação indireta entre componentes fracamente acoplados facilita a adaptação e melhora a escalabilidade do sistema

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Publish-Subscribe*:
 - A denominação surge da analogia com os editores (*publishers*) e assinantes (*subscribers*) de revistas e jornais:
 - O editor periodicamente cria a informação e o assinante obtém uma cópia desta informação ou pelo menos é informado da sua disponibilidade
 - É adequado para aplicações onde existe uma distinção clara entre produtores e consumidores de informação:
 - Ex: agência *on-line* de empregos
 - Variações geralmente existem em função da distância entre o *publisher* e os *subscribers* e da forma de gerenciamento destes relacionamentos

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Publish-Subscribe* simples:
 - O *publisher* mantém uma lista de *subscribers*
 - Para cada *subscriber* uma *Procedure Call* é disparada sempre que uma nova informação estiver disponível
 - *Subscribers* realizam suas assinaturas com o *publisher*, informando a interface de *procedure (callback)* a ser utilizada quando a informação for publicada
 - *Subscribers* podem cancelar suas assinaturas e ter seus respectivos *callbacks* removidos da lista de assinantes do *publisher*

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Publish-Subscribe*:
 - Em aplicações baseadas em rede e de larga escala algumas modificações são necessárias:
 - *Publishers* precisam anunciar (no *start-up* do sistema, periodicamente ou sob demanda) a existência de recursos de informação que podem ser assinados
 - Assinaturas não são mais representadas por *procedures* de *callback* e passam a envolver protocolos de rede
 - Aspectos de desempenho impedem o uso de conexões ponto-a-ponto entre *publishers* e *subscribers*, demandando *proxies* ou *caches* intermediários

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Publish-Subscribe*:

Resumo: *subscribers* solicitam/cancelam o recebimento de mensagens específicas; *publishers* mantêm uma lista de assinantes e a eles enviam mensagens de forma síncrona ou assíncrona

Componentes: *publishers*, *subscribers*, *proxies* para gerenciamento da distribuição

Conectores: *procedure calls* (dentro de programas) ou protocolos de rede; assinaturas baseadas em conteúdo requerem conectores mais sofisticados

Elementos de Dados: assinaturas, notificações e informações publicadas

Topologia: *subscribers* se conectam diretamente aos *publishers* ou através de intermediários

Variações: usos específicos do estilo podem requerer passos particulares na assinatura e cancelamento; suporte a correspondências complexas entre interesses de assinatura e informação disponível pode ser realizada por intermediários

Qualidades Induzidas: disseminação (*one-way*) eficiente de informação; baixo acoplamento

Usos Típicos: disseminação de notícias; programação de interfaces gráficas de usuário; jogos *multiplayer* baseados em rede

Precauções: quando o número de assinantes de uma mesma informação é alto geralmente é necessário um protocolo especializado de *broadcast*

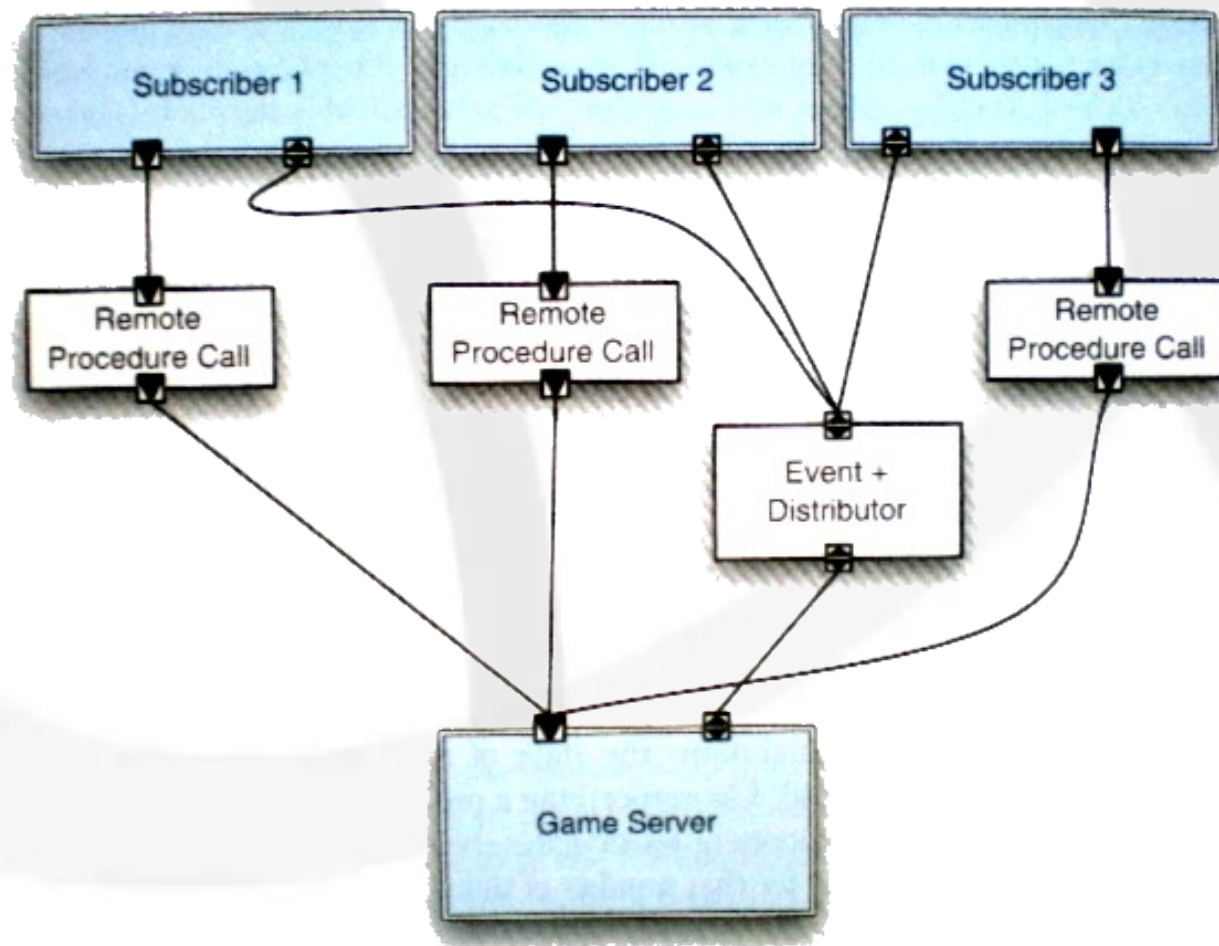
Relacionamento com Linguagens de Programação e Ambientes: geralmente disponibilizado por alguma tecnologia de *middleware*

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Publish-Subscribe* (pouso lunar):



Estilos Arquiteturais

Baseados em Invocação Implícita



- *Event-Based:*
 - Caracterizado por componentes independentes que se comunicam somente através de eventos transmitidos por um barramento (conector)
 - Na sua forma mais pura componentes emitem eventos para o barramento que, por sua vez, os re-transmite para todos os outros componentes
 - Componentes podem reagir em resposta ao recebimento de um evento ou ignorá-lo
 - Embora aparentemente caótico e imprevisível é similar à forma com a qual humanos se comportam em sociedade

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Event-Based*:
 - Por razões de eficiência a forma pura deste estilo raramente é utilizada
 - É mais eficiente distribuir os eventos somente para aqueles componentes que demonstraram interesse por eles
 - Com esta modificação o *Event-Based* se torna similar ao *Publish/Subscribe*, entretanto não há distinção entre produtores e consumidores
 - A replicação e otimização de distribuição dos eventos (ex: registro de interesse em um evento particular) é responsabilidade somente dos conectores

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Event-Based:*
 - Pode funcionar em modo:
 - *Pull (pooling)*: receptores de eventos consultam o conector (de forma síncrona ou assíncrona) para verificar se algum novo evento está disponível
 - *Push*: o conector replica e re-transmite os eventos aos possíveis interessados
 - Altamente indicado para sistemas com componentes concorrentes altamente desacoplados onde, em um determinado momento, um componente pode estar ou criando ou consumindo informação
 - Ex: mercado financeiro / bolsa de valores

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Event-Based:*

Resumo: componentes independentes emitem e recebem, de forma assíncrona, eventos transmitidos por barramentos de eventos

Componentes: produtores e consumidores independentes e concorrentes de eventos

Conectores: barramento de eventos; em certas variações, mais de um conector pode ser utilizado

Elementos de Dados: eventos – dados enviados como entidades de primeira-ordem através de barramentos de eventos

Topologia: componentes se comunicam somente com os barramentos de eventos

Variações: a comunicação dos componentes com os barramentos pode acontecer em modo *push* ou *pull*

Qualidades Induzidas: altamente escalável; fácil de evoluir; efetivo para aplicações altamente distribuídas e heterogêneas

Usos Típicos: interfaces gráficas de usuário; aplicações *wide-area* envolvendo partes independentes (mercado financeiro, logística, redes de sensores)

Precauções: não há garantia se ou quando um evento particular será processado

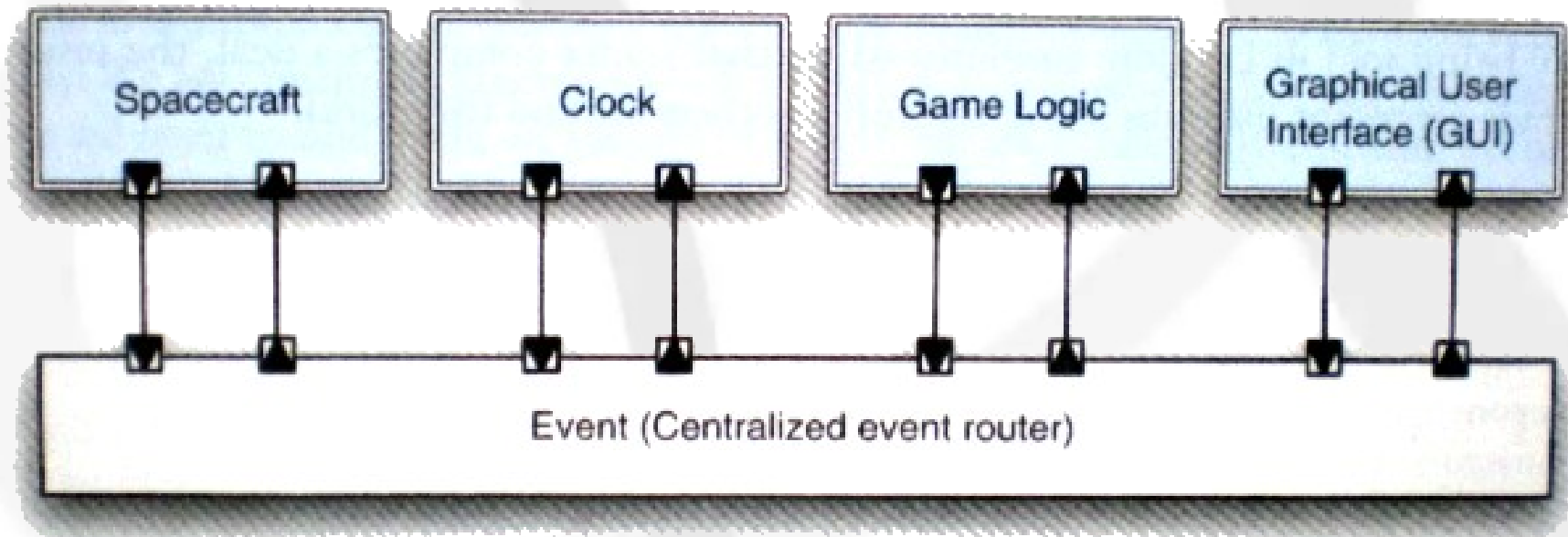
Relacionamento com Linguagens de Programação e Ambientes: tecnologias de *middleware* orientado a mensagens (*JMS*, *CORBA Event Service*, *MSMQ*)

Estilos Arquiteturais

Baseados em Invocação Implícita



- *Event-Based* (pouso lunar):



Estilos Arquiteturais

Peer-to-Peer



- *Peer-to-Peer* (P2P):
 - Consiste em uma rede de peers autônômos fracamente acoplados
 - Cada peer atua tanto como cliente quanto servidor
 - Peers se comunicam utilizando um protocolo de rede, provavelmente especializado para comunicação P2P (ex: Napster, Gnutella)
 - Descentraliza tanto informação quanto controle, fazendo com que a descoberta de recursos seja um aspecto importante

Estilos Arquiteturais

Peer-to-Peer



- *Peer-to-Peer* (P2P):
 - Descoberta de recursos em sistemas P2P puros:
 - A solicitação da informação é lançada na rede como um todo
 - A requisição se propaga até que a informação seja descoberta ou algum limite de propagação (ex: número de *hops*) seja alcançado
 - Se a informação é encontrada o *peer* obtém o endereço direto do outro *peer* e o contacta diretamente
 - É limitado pelo algoritmo distribuído utilizado para consultar o sistema e pela largura de banda disponível

Estilos Arquiteturais

Peer-to-Peer



- *Peer-to-Peer* (P2P):
 - Descoberta de recursos em sistemas P2P híbridos:
 - O processo é otimizado através da presença de peers especiais, especializados na localização de outros *peers* e/ou disponibilização de diretórios que localizam as informações
 - Ex: Napster – utilizava um servidor centralizado para indexação das músicas e localização de outros *peers*
 - Embora o estilo tenha se tornado popular nas aplicações de compartilhamento de arquivos é frequentemente utilizado em *B2B commerce*, chat, colaboração remota e redes de sensores

Estilos Arquiteturais

Peer-to-Peer



- *Peer-to-Peer* (P2P):

Resumo: estado e comportamento estão distribuídos entre *peers* que podem atuar tanto como clientes quanto como servidores

Componentes: *peers* – componentes independentes com seu estado e *thread* de controle próprios

Conectores: protocolos de rede, frequentemente especializados

Elementos de Dados: mensagens de rede

Topologia: em rede (com possibilidade de conexões redundantes entre *peers*); pode variar arbitrariamente e dinamicamente

Qualidades Induzidas: computação descentralizada com fluxo de controle e recursos distribuídos entre os *peers*; altamente robusto na presença de falhas em qualquer nó; escalável em relação ao acesso a recursos e poder computacional

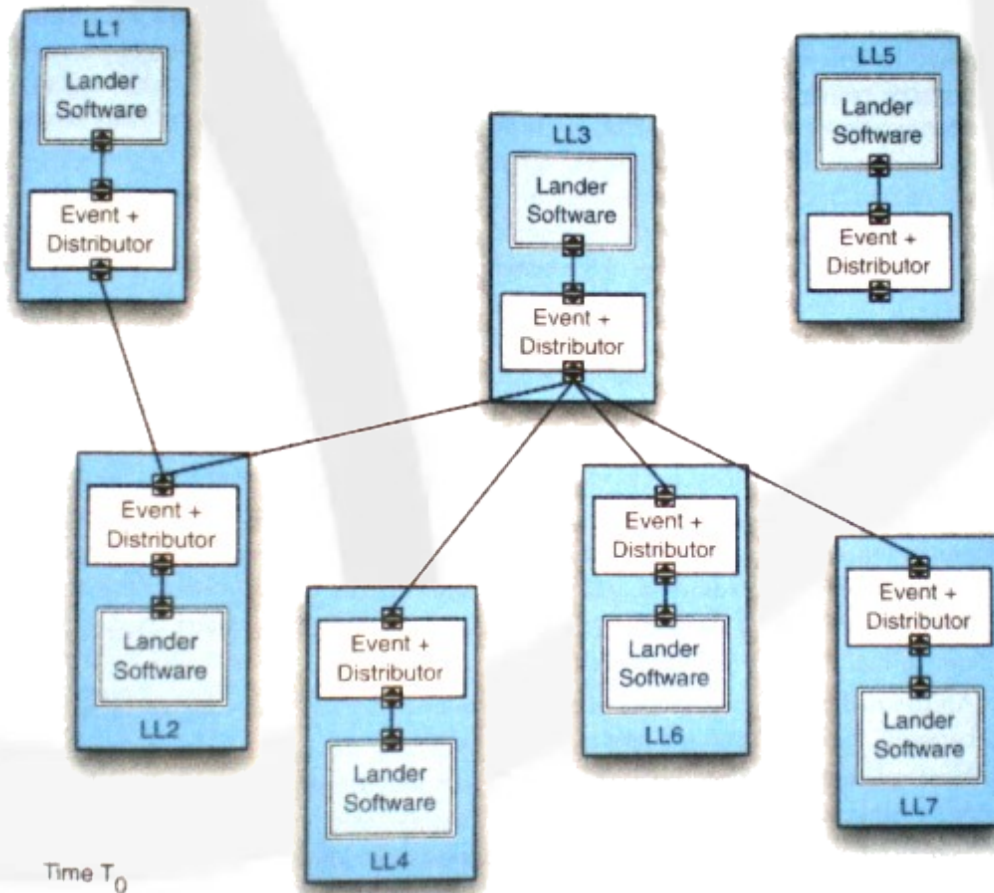
Usos Típicos: onde as operações e fontes de informação estão distribuídas e a rede é *ad-hoc*

Precauções: quando o tempo necessário para recuperação da informação é importante e é inviável lidar com a latência imposta pelo protocolo; segurança (deve-se detectar *peers* maliciosos e prover meios para gerenciar a confiança – *trust* – em ambientes abertos)

Estilos Arquiteturais Peer-to-Peer



- *Peer-to-Peer* (P2P) (pouso lunar):



Estilos Arquiteturais

Peer-to-Peer



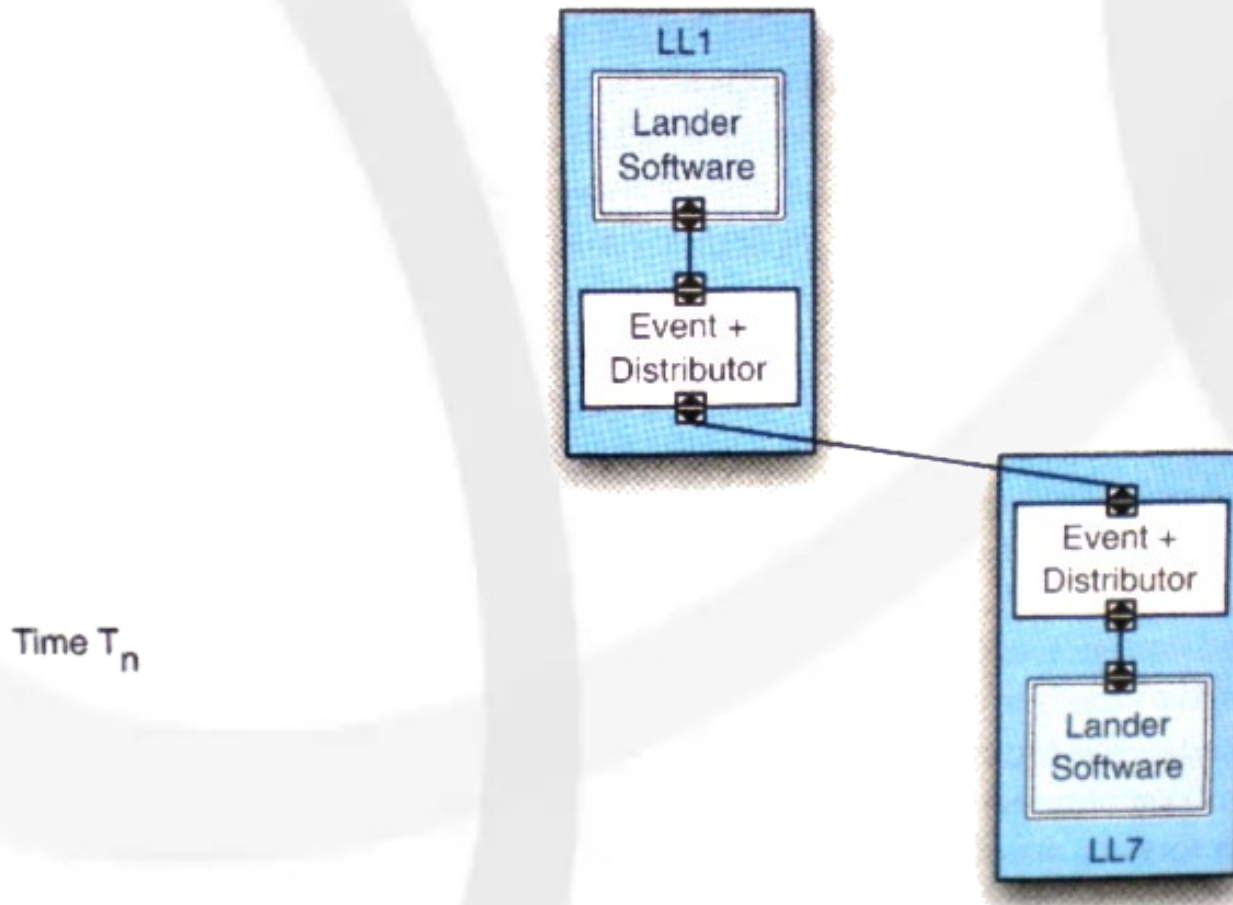
- *Peer-to-Peer* (P2P) (pouso lunar):
 - Obtenção da informação pelo *Lunar Lander 1* (LL1):
 - 1) No tempo T_0 , LL1 contacta todas as naves presentes no raio de comunicação
 - 2) Somente LL2 responde
 - 3) LL1 pergunta a LL2 se ela possui a informação desejada
 - 4) Visto que LL2 não possui esta informação ela repassa a pergunta para o seu nó de comunicação adjacente – LL3 (assume-se que LL2 e LL3 já se conhecem)
 - 5) Visto que LL3 não possui esta informação ela repassa a pergunta para LL4, LL6 e LL7
 - 6) LL7 informa, a LL3, que possui a informação e a envia para ela
 - 7) LL3 passa a informação de volta a LL2 e, subsequentemente, a LL1
 - Em um tempo T_n LL7 adentra o raio de comunicação de LL1 e elas agora podem se contactar diretamente

Estilos Arquiteturais

Peer-to-Peer



- *Peer-to-Peer* (P2P) (pouso lunar):





Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Arquitetura de Software - Conectores

Sandro S. Andrade
sandroandrade@ifba.edu.br

Conectores



- Conectores de *software* realizam transferência de controle e dados entre componentes
- Conectores também podem disponibilizar serviços (ex: persistência, invocação, messaging e transações) independente da funcionalidade dos componentes envolvidos
- Tais serviços são geralmente considerados “*facilities components*” (ex: no CORBA, DCOM e RMI)
 - Tratar estes serviços, entretanto, como conectores deixa a arquitetura mais clara e mantém os componentes com foco nos aspectos referentes à aplicação e ao domínio

Conectores



- O que os conectores podem fazer ?
 - Distribuir uma requisição de serviço para componentes especificamente identificados
 - Realizar *broadcast* de uma notificação de evento para qualquer componente interessado (não identificado ou até mesmo desconhecido)
 - Requerer que o componente solicitante suspenda o seu processamento até que um *ACK* seja recebido (síncrono/blocking) ou permitir que o componente solicitante continue com o seu processamento (assíncrono/*non-blocking*)
 - Rotear requisições na ordem recebida
 - Ordenar, filtrar e combinar requisições de acordo com alguma regra pré-definida

Conectores



- Os conectores mais simples estão tipicamente já implementados nas linguagens de programação
- Conectores compostos, por outro lado, formados pela composição de vários conectores (e possivelmente componentes), são geralmente disponibilizados como bibliotecas ou *frameworks*
- Conectores simples disponibilizam somente um tipo de serviço de interação
- Os conectores compostos ajudam a superar as limitações das linguagens de programação modernas

Tipos de Serviços de Interação



- Um conector disponibiliza um ou mais dos seguintes tipos de serviços de interação:
 - 1) Comunicação
 - 2) Coordenação
 - 3) Conversão
 - 4) Facilitação
- Todo conector disponibiliza serviços de pelo menos uma dessas categorias
- Um conjunto rico de capacidades de interação pode demandar o uso de múltiplos serviços:
 - Ex: *Procedure Call* (comunicação + coordenação)

Tipos de Serviços de Interação



1) Serviços de Comunicação:

- Suporta a transmissão de dados entre componentes
- *Building block* primário para interação entre componentes
- Exemplos de dados: mensagens, dados a serem processados, resultados de computações

Tipos de Serviços de Interação



2) Serviços de Coordenação:

- Suportam a transferência de controle entre componentes
- A *thread* de execução é passada de um componente para outro
- Exemplos de conectores de coordenação: chamadas de função e invocações de método
- Conectores de mais alta ordem, tais como aqueles utilizados para balanceamento de carga, disponibilizam interações mais ricas e complexas construídas em torno dos serviços de coordenação

Tipos de Serviços de Interação



3) Serviços de Conversão:

- Transformam a interação requerida por um componente naquela disponibilizada por outro
- Permitir que componentes heterogêneos interajam não é uma tarefa fácil, interações divergentes estão sempre presentes
- Tais divergências são causadas por incompatibilidades no tipo, número, frequência e ordem das interações
- Serviços de conversão permitem que componentes que não foram projetados para trabalhar em conjunto possam estabelecer e conduzir interações
 - Exs: conversão de formato de dados e *wrappers* para componentes legados

Tipos de Serviços de Interação



4) Serviços de Facilitação:

- Realizam a mediação e simplificação da interação entre componentes
- Mesmo quando os componentes foram projetados para trabalhar em conjunto pode ser necessário disponibilizar mecanismos para melhor facilitar e otimizar as suas interações
- Mecanismos tais como balanceamento de carga, serviços de escalonamento e controle de concorrência podem ser necessários para atender certos requisitos não-funcionais e para reduzir inter-dependência entre componentes

Tipos de Conectores



- Classificação dos conectores quanto à forma de realização dos serviços de interação:
 - 1) Procedure Call
 - 2) Event
 - 3) Data Access
 - 4) Linkage
 - 5) Stream
 - 6) Arbitrator
 - 7) Adaptor
 - 8) Distributor

Tipos de Conectores

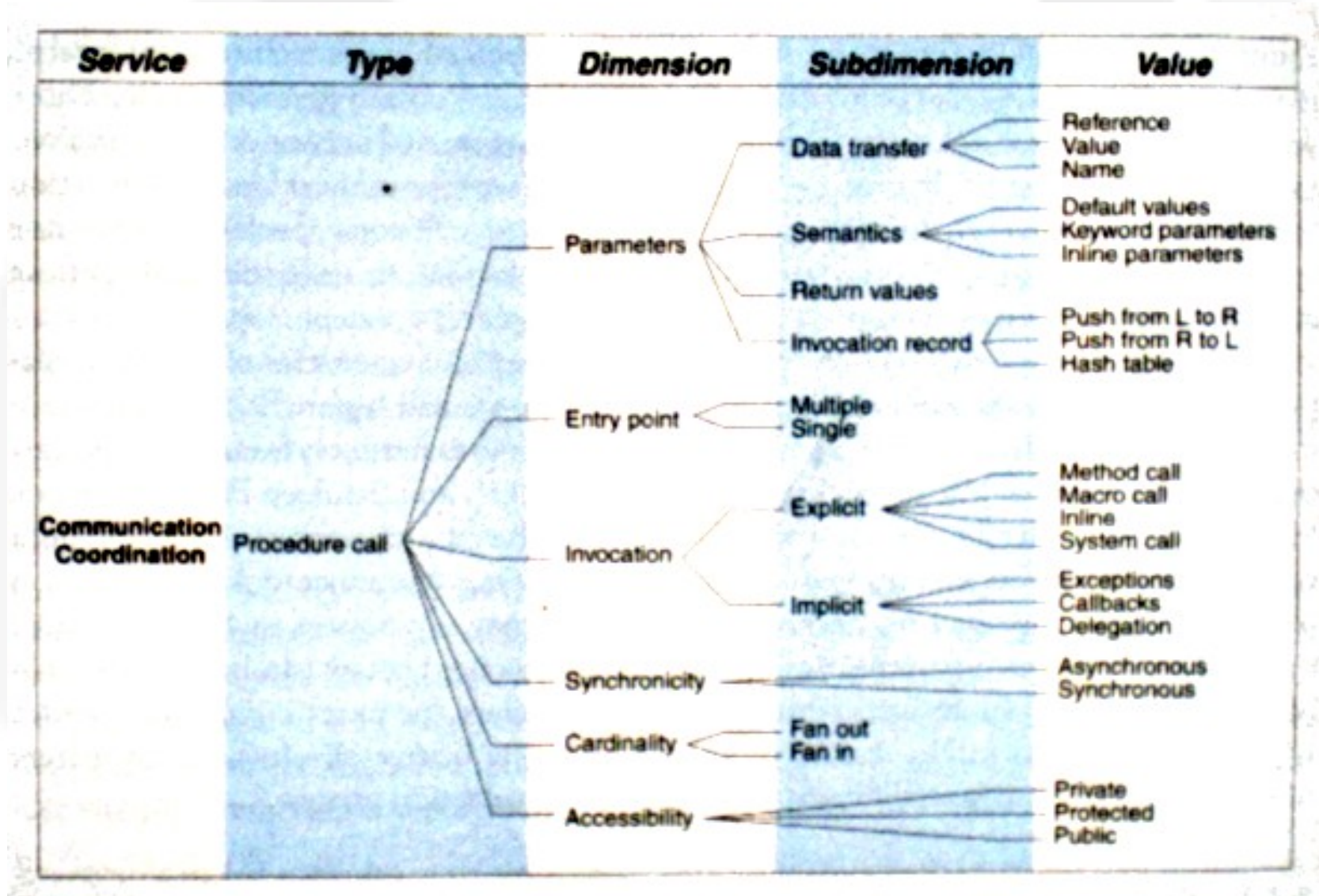
Procedure Call



- Modelam o fluxo de controle entre componentes através de diversas técnicas de invocação (serviço de coordenação)
- Adicionalmente, podem transferir dados entre os componentes envolvidos através de parâmetros e valores de retorno (serviço de comunicação)
- São os mais conhecidos e amplamente utilizados: métodos na orientação a objetos, chamadas de sistema e callbacks
- Frequentemente utilizados como base para conectores composite, como *Remote Procedure Call* (RPC). Neste caso, também realizando serviços de facilitação

Tipos de Conectores

Procedure Call



Tipos de Conectores

Event



Evento: efeito instantâneo da conclusão (normal ou errônea) da invocação de uma operação em um objeto, ocorrendo na localização do próprio objeto

David Rosenblum e Alexander Wolf (1997)

- São similares ao *Procedure Call* pois afetam o fluxo de controle entre componentes (serviços de coordenação)
 - O fluxo de controle é iniciado pela ocorrência do evento
 - O conector, uma vez ciente da ocorrência do evento, gera mensagens (notificações de eventos) para todas as partes interessadas e produz controle para que os componentes processem estas mensagens

Tipos de Conectores

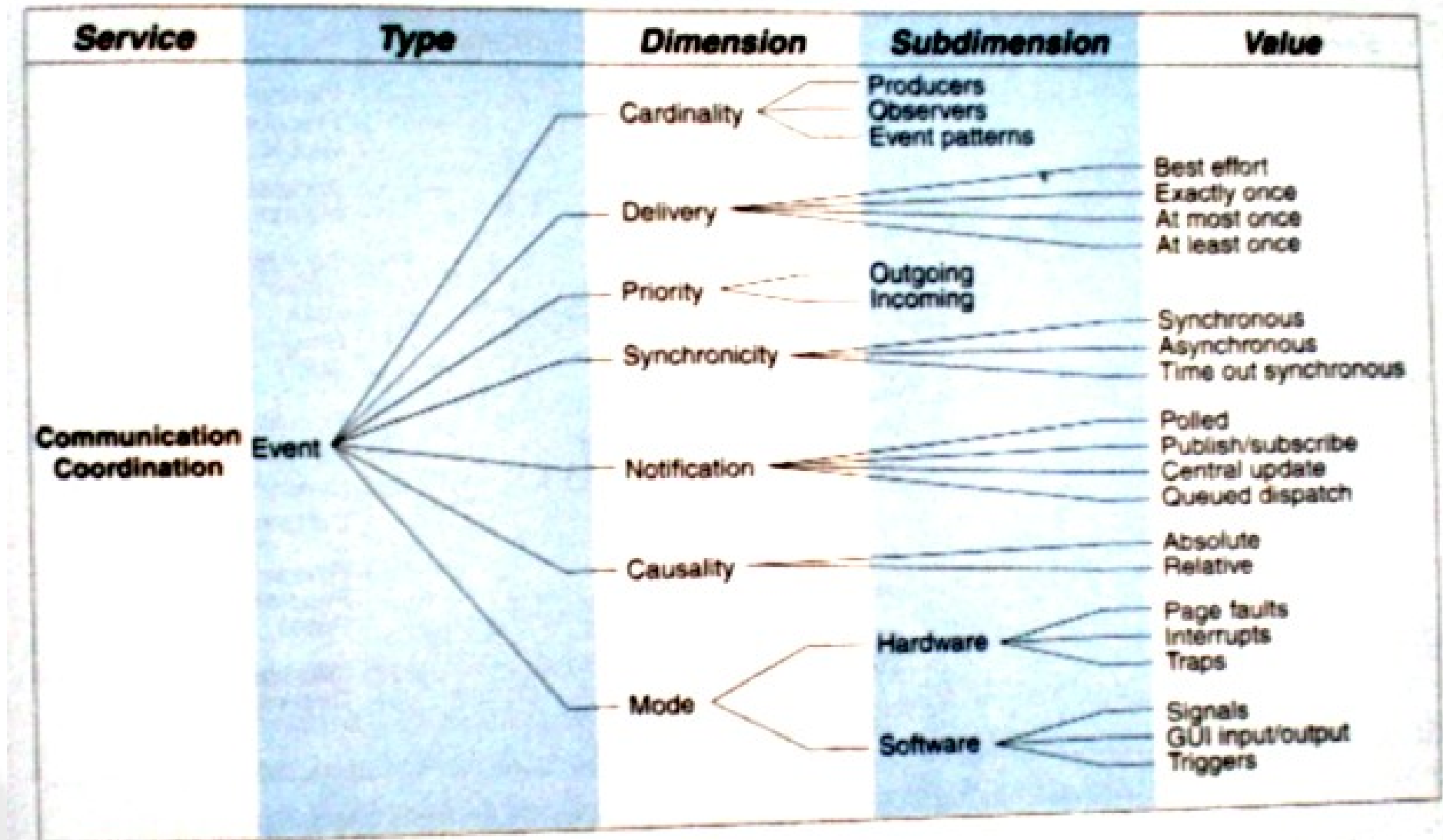
Event



- Entretanto, diferem do *Procedure Call* pois:
 - Mensagens podem ser geradas na ocorrência de um único evento ou de um padrão específico de eventos
 - O conteúdo do evento pode ser estruturado para conter informações tais como hora e local da ocorrência do evento e outros dados específicos de aplicação (serviços de comunicação)
 - Formam “conectores virtuais” entre os componentes interessados no mesmo evento
 - Tais “conectores virtual” aparecem e desaparecem dinamicamente durante a execução do sistema

Tipos de Conectores

Event



Tipos de Conectores

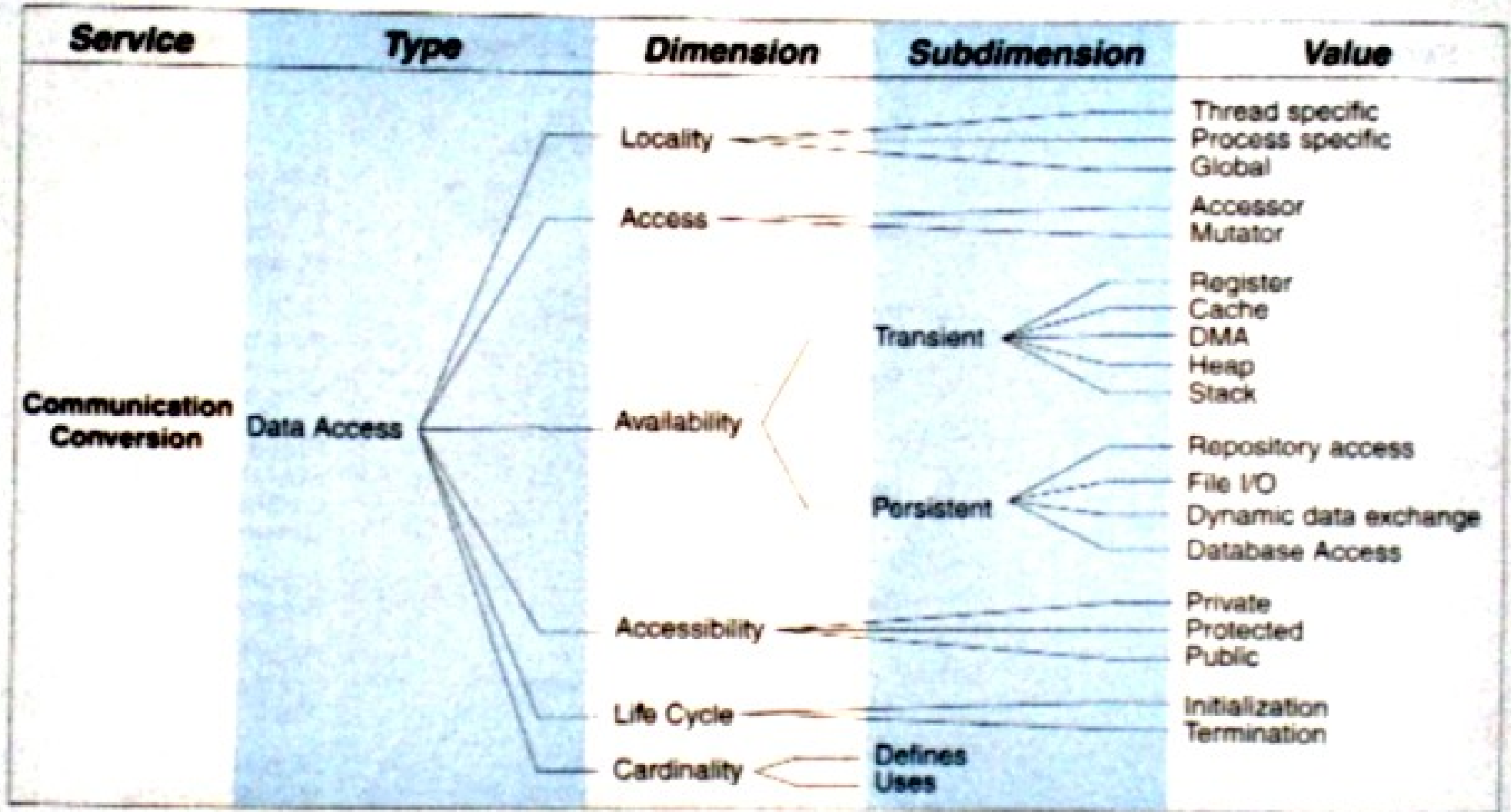
Data Access



- Permitem que componentes acessem dados mantidos em um componente de armazenamento de dados - *Data Store* (serviços de comunicação)
- Tal acesso frequentemente requer inicialização e limpeza do *Data Store* antes e depois da operação, respectivamente
- Caso exista diferença de formato entre o dado requerido e aquele armazenado o conector pode realizar tradução da informação (serviço de conversão)
- O *Data Store* pode ser persistente ou temporário, impactando no mecanismo utilizado pelo conector
- Ex: mecanismos de *query* (SQL) e acesso a informação de repositórios tais como os de componentes de *software*

Tipos de Conectores

Data Access



Tipos de Conectores

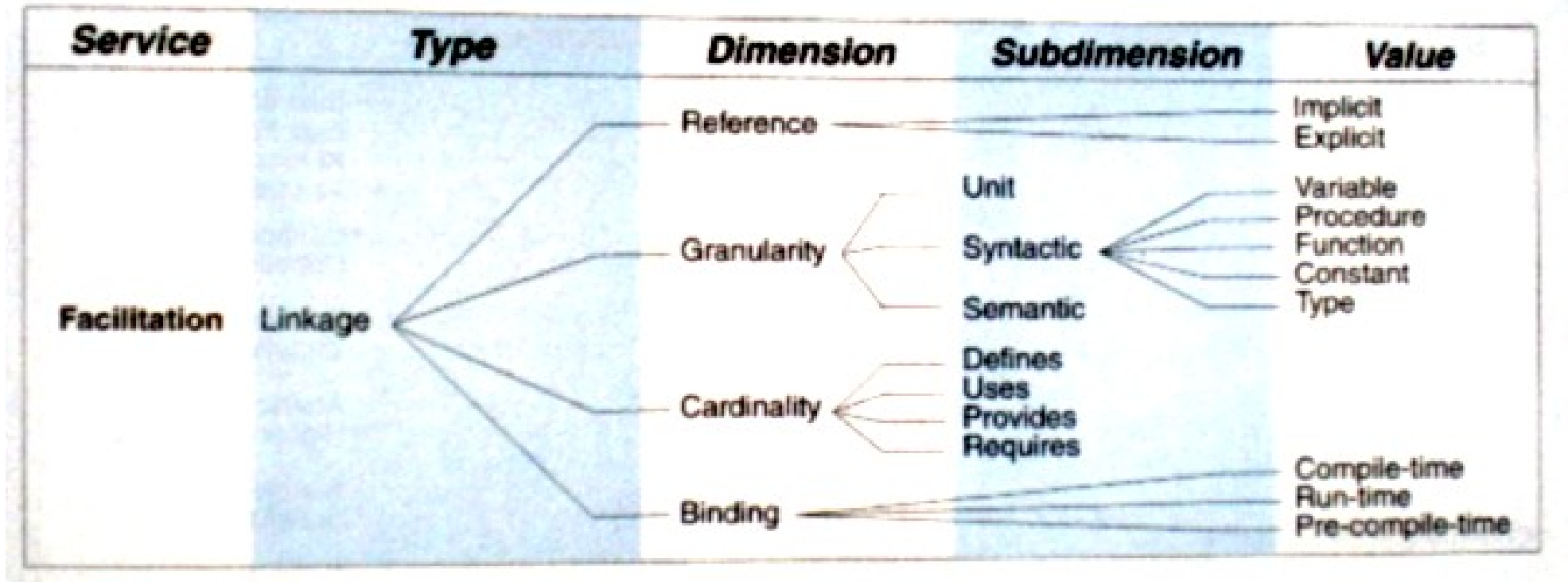
Linkage



- Utilizados para unir os componentes e mantê-los desta forma durante sua interação
- Viabilizam o estabelecimento de dutos – canais de comunicação e coordenação – que são então utilizados por conectores de mais alta ordem para garantir a semântica de interação
- Realizam serviços de facilitação
- Uma vez o duto estabelecido o conector linkage pode ser removido do sistema ou nele permanecer para facilitar a evolução do *software*
- Ex: ligações entre componentes e barramentos no C2 e relacionamentos de dependência entre módulos de *software* em *Module Interconnection Languages* (MIL)

Tipos de Conectores

Linkage



Tipos de Conectores Linkage



- Sub-dimensões de granularidade:
 - Interconexões de unidade especificam somente que um componente (modulo, objeto, arquivo, ...) depende de outro. Ex: *build tools* tais como o Make
 - Interconexões sintáticas refinam este relacionamento e estabelecem ligações entre variáveis, procedures, funções, constantes e tipos definidos dentro do componente conectado. Ex: análise estática e *smart compilation*
 - Interconexões semânticas especificam como os componentes ligados devem interagir. Garantem que os requisitos e restrições da interação são explicitamente afirmados e satisfeitos. Ex: protocolos de interação

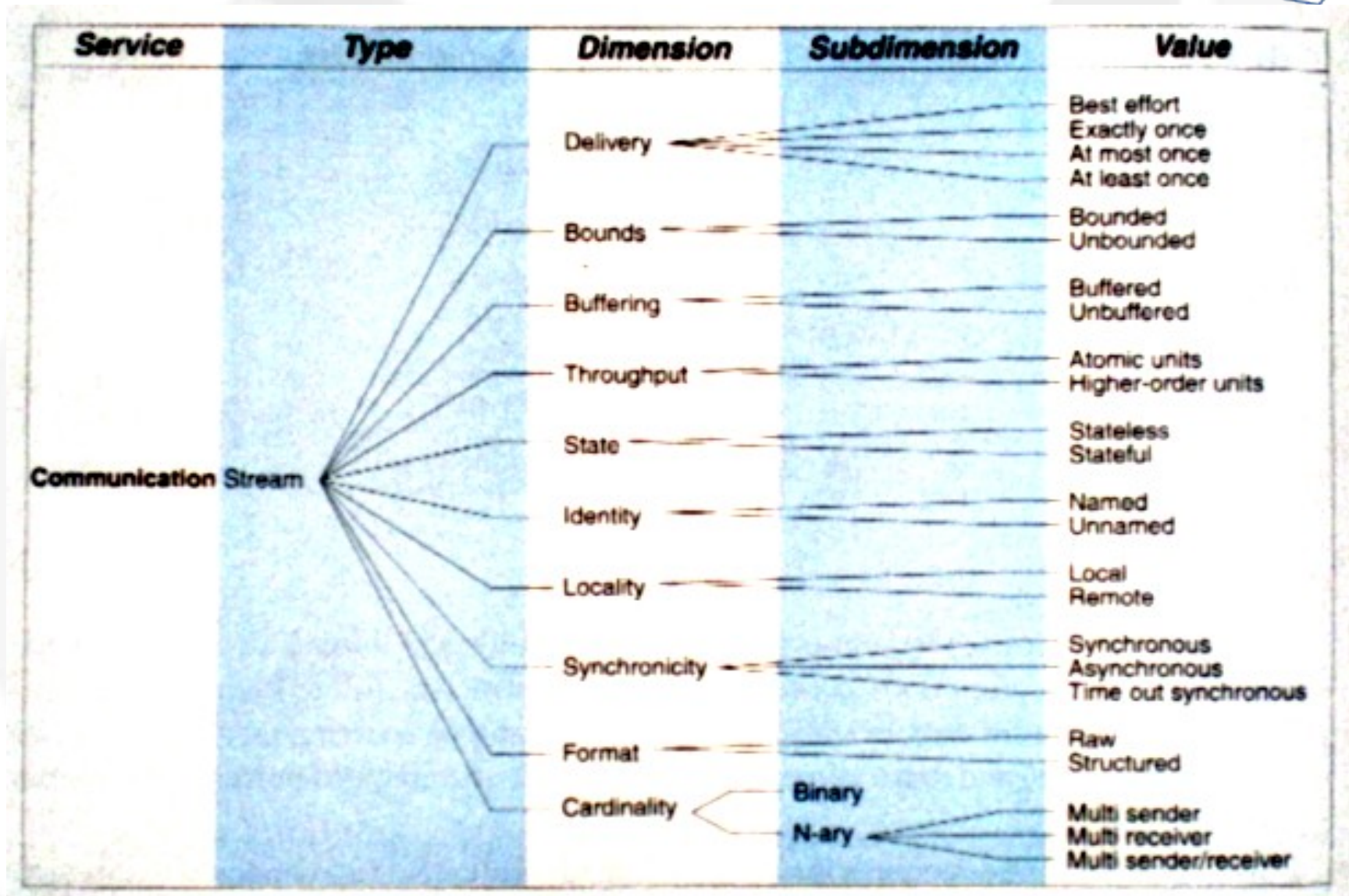
Tipos de Conectores

Stream



- Utilizados para realizar transferências de grandes quantidades de dados entre processos autônomos (serviços de comunicação)
- São também utilizados em protocolos de transmissão de dados em sistemas *client-server* para realizar a entrega de resultados de computações
- Podem ser combinados com outros tipos de conectores:
 - *Data Access* para definir conectores composite de acesso a bancos de dados
 - *Event* para multiplexar a entrega de uma grande quantidade de eventos
- Ex: *pipes* do Unix, *sockets* de comunicação TCP/UDP e protocolos *client-server* proprietários

Tipos de Conectores Stream



Tipos de Conectores

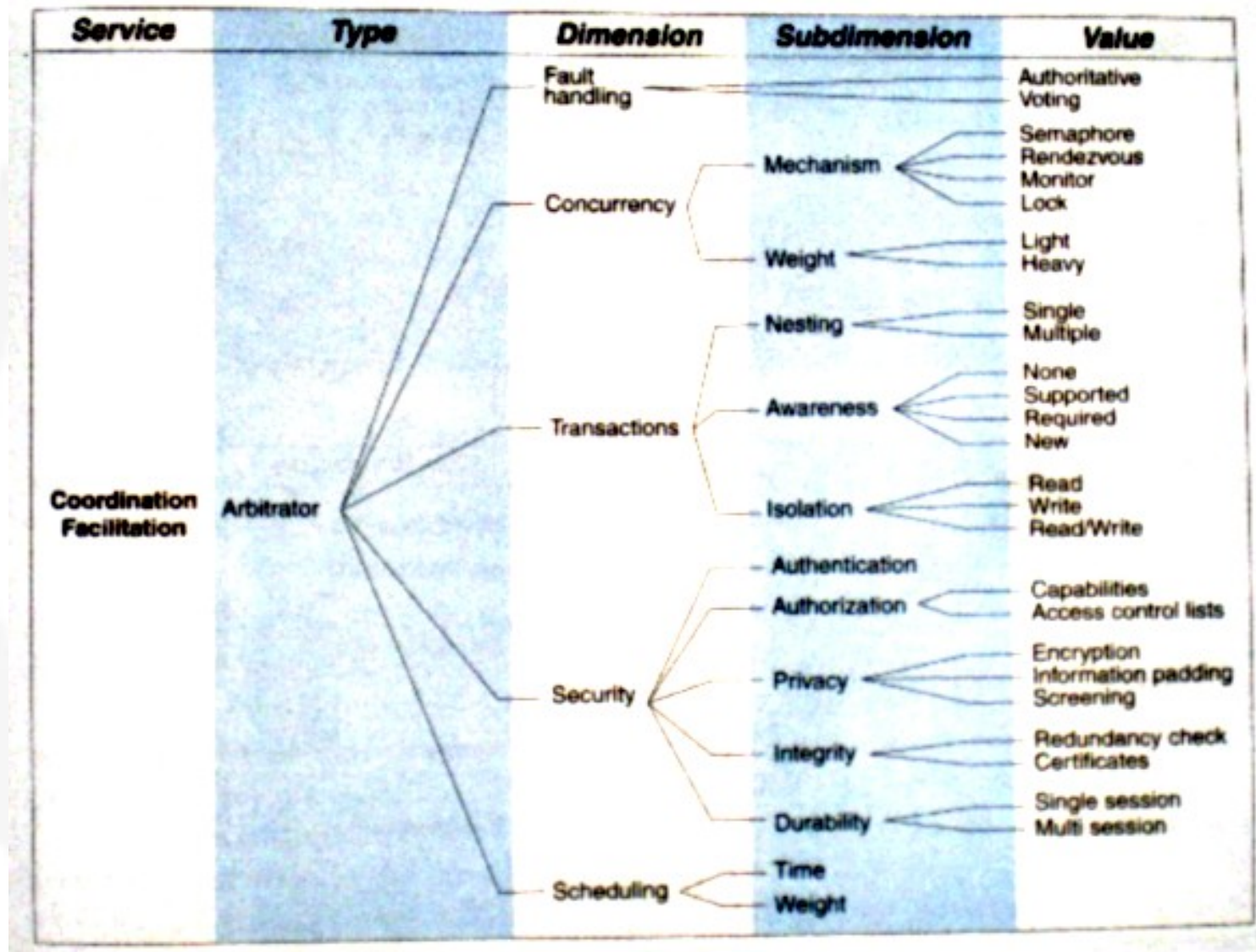
Arbitrator



- Facilitam a operação do sistema, resolvem eventuais conflitos (serviços de facilitação) e redirecionam o fluxo de controle (serviços de coordenação) naquelas situações onde um componente conhece a presença de outros componentes porém nada pode assumir sobre suas necessidades e seus estados
 - Ex: garantia de consistência e atomicidade de operações, através de sincronização e controle de concorrência, em sistemas *multithreaded* com memória compartilhada
 - Ex: negociação de níveis de serviço e mediação de interações que requerem confiabilidade e atomicidade
 - Ex: serviços de escalonamento e balanceamento de carga
 - Ex: *reliability*, *security* e *safety* para implementação de *dependability* e *trustworthiness*

Tipos de Conectores

Arbitrator



Tipos de Conectores

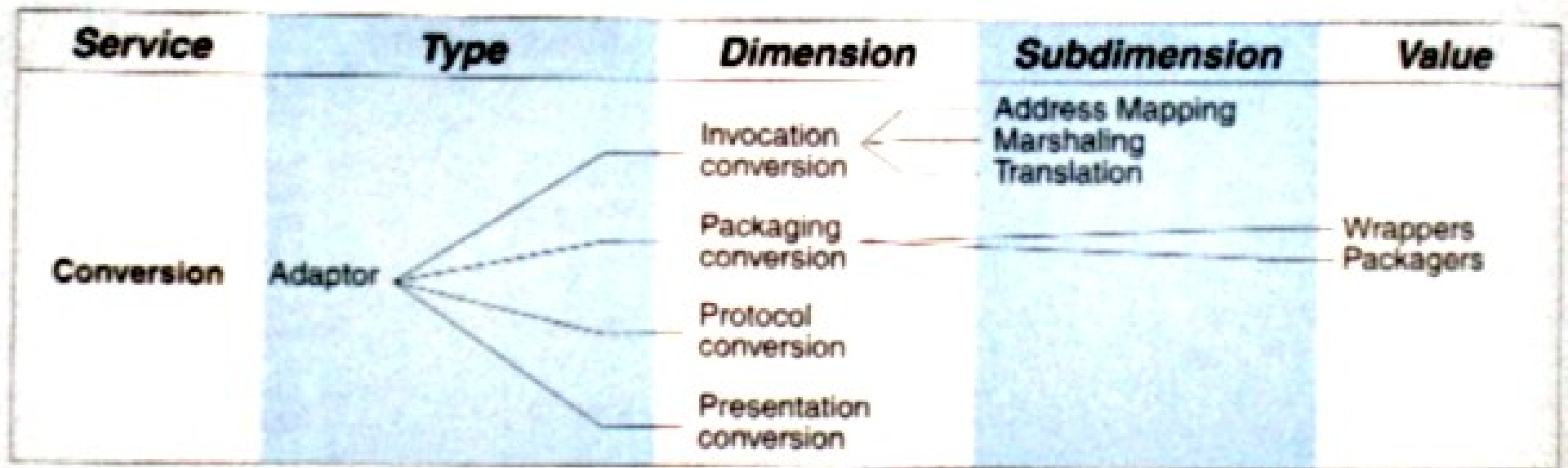
Adaptor



- Suportam a interação entre componentes que não foram originalmente projetados para interoperar
- Compatibilizam políticas de comunicação e protocolos de interação (serviços de conversão)
- Necessários em ambientes heterogêneos
- A conversão pode ter a melhoria do desempenho como foco:
 - Um *Remote Procedure Call* pode ser automaticamente convertido para um *Procedure Call* local caso os dois componentes estejam na mesma máquina
- Podem aplicar transformações (ex: *look-ups*) para compatibilizar os serviços requeridos com as facilidades disponíveis

Tipos de Conectores

Adaptor



Tipos de Conectores

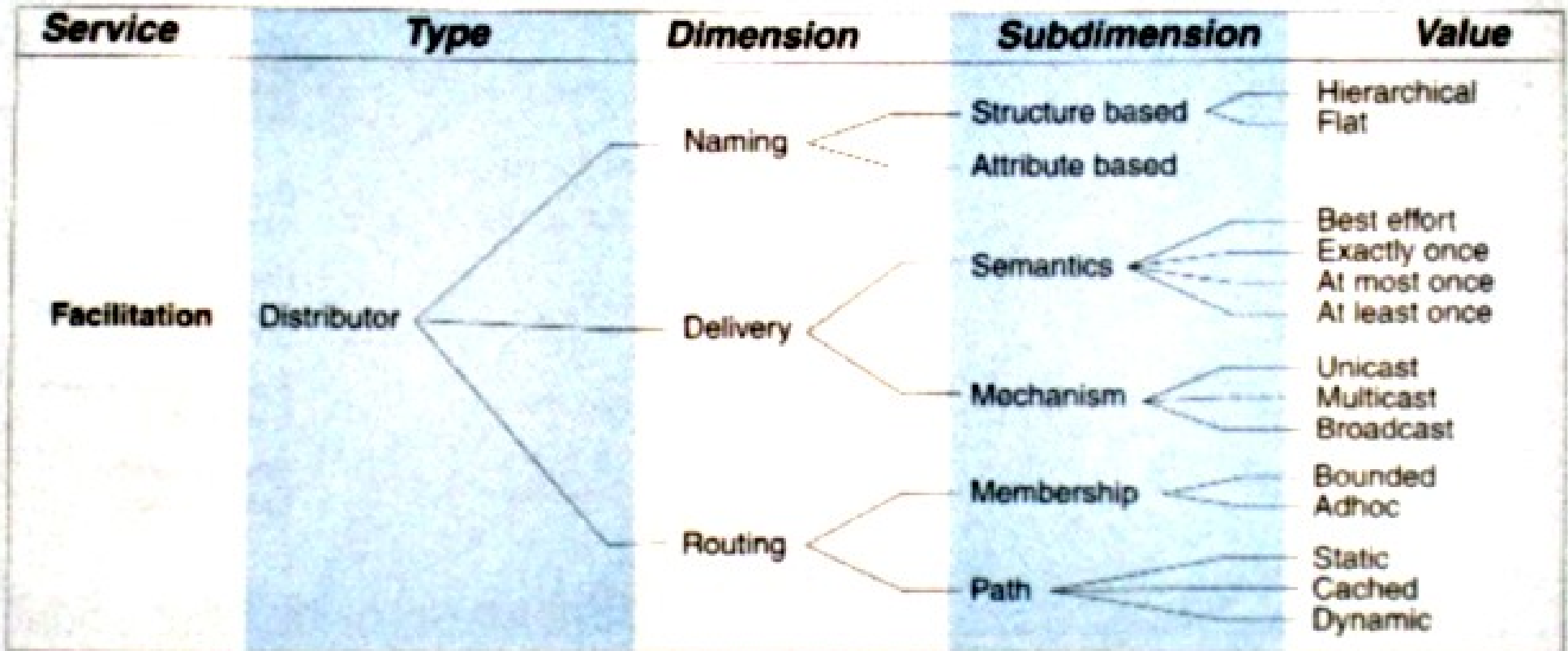
Distributor



- Realizam a identificação dos caminhos (*paths*) de interação e o subsequente roteamento, de informações de comunicação e coordenação, através de objetos ao longo deste caminho (serviços de facilitação)
- Nunca existem de forma isolada, eles dão assistência a outros conectores como *Stream* ou *Procedure Call*
- Direcionam o fluxo de dados durante troca de informação em sistemas distribuídos
- Ex: serviços de identificação da localização de componentes e de caminhos até eles, a partir de nomes simbólicos (DNS)
- Tem efeito importante na escalabilidade e resiliência do sistema

Tipos de Conectores

Distributor





Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Arquitetura de Software - Arquiteturas e Estilos Aplicados

Sandro S. Andrade
sandroandrade@ifba.edu.br

Arquiteturas Distribuídas e em Rede

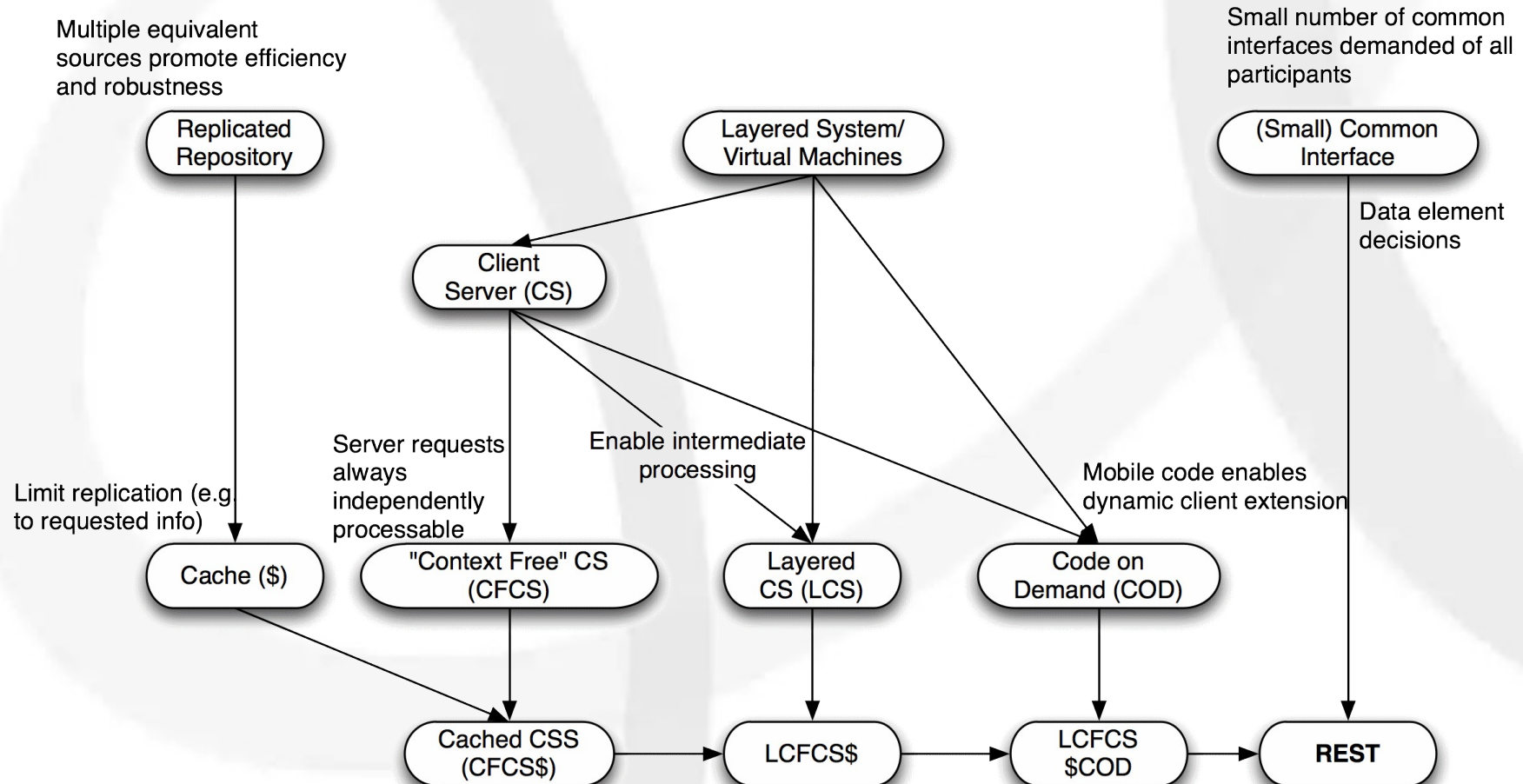


- Estudos de caso:
 - Arquiteturas para aplicações baseadas em rede:
 - *The REpresentational State Transfer* (REST)
 - Google
 - Arquiteturas Descentralizadas
 - Peer-to-Peer
 - Napster – *Hybrid Client-Server / Peer-to-Peer*
 - Skype – *Overlayered P2P*
 - BitTorrent – *Resource Trading P2P*

Arquiteturas Distribuídas e em Rede



- *The REpresentational State Transfer (REST):*



Arquiteturas Distribuídas e em Rede



- Google:
 - Evolução: *search engine* → conjunto amplo de aplicações
 - Produtos fortemente baseados na web porém não são *REST-based*
 - A arquitetura dos sistemas do Google foca na escalabilidade, assim como a web, porém a natureza das aplicações e as estratégias da empresa demandam uma arquitetura completamente diferente
 - Os diferentes produtos do Google compartilham elementos comuns

Arquiteturas Distribuídas e em Rede



- Google:
 - Características das aplicações:
 - Devem manipular uma quantidade imensa de informação: armazenamento, estudo e manipulação de terabytes
 - Armazenamento e manipulação suportado por milhares de *hardware commodity* (PC baratos rodando Linux)



Arquiteturas Distribuídas e em Rede



- Google:
 - Características das aplicações:
 - Ao suportar efetivamente a replicação de processamento e armazenamento de dados, uma plataforma de computação altamente escalável e tolerante a falhas pode ser construída
 - Premissa: falhas irão ocorrer e deverão ser acomodadas
 - As aplicações do Google não precisam de todas as funcionalidades disponibilizadas por um serviço de gerenciamento de banco de dados

Arquiteturas Distribuídas e em Rede



- Google:
 - Características das aplicações:
 - Solução: *Google File System* (GFS) – sistema de armazenamento simples (poucas funcionalidades) porém executando sobre uma plataforma altamente tolerante a falhas
 - Otimizações do GFS (em contraponto a um banco de dados):
 - Arquivos tipicamente muito grandes (vários gigabytes)
 - Falhas de componentes de armazenamento são esperadas e tratadas
 - Arquivos geralmente sofrem apenas append (ao invés de modificações randômicas)
 - Regras mais relaxadas para manutenção de consistência em acessos concorrentes

Arquiteturas Distribuídas e em Rede



- Google:
 - Características das aplicações:
 - Um número de aplicações executam sobre o GFS. Dentre elas, destaca-se o *MapReduce* que disponibiliza um modelo de programação com operações para seleção e redução de dados, presentes nos imensos conjuntos de dados do Google
 - O *MapReduce* é responsável pela paralelização da operação, onde centenas de processadores são utilizados de forma transparente ao desenvolvedor
 - Falhas nos processadores envolvidos na execução paralela são graciosamente acomodadas

Arquiteturas Distribuídas e em Rede



- Google – lições arquiteturais:
 - Uso abundante de camadas de abstração:
 - *GFS* – abstrai detalhes da distribuição dos dados e falhas
 - *MapReduce* – abstrai os detalhes da paralelização das operações
 - Desde o início, o projeto foi concebido de modo a lidar com falhas de processamento, armazenamento e comunicação → alta robustez
 - Escala é tudo, tudo é construído com escalabilidade como foco
 - Projeto especializado para o domínio → alto desempenho e baixo custo
 - Desenvolvimento de abordagem genérica (*MapReduce*) para extração/redução de dados → alto reuso

Arquiteturas Distribuídas e em Rede

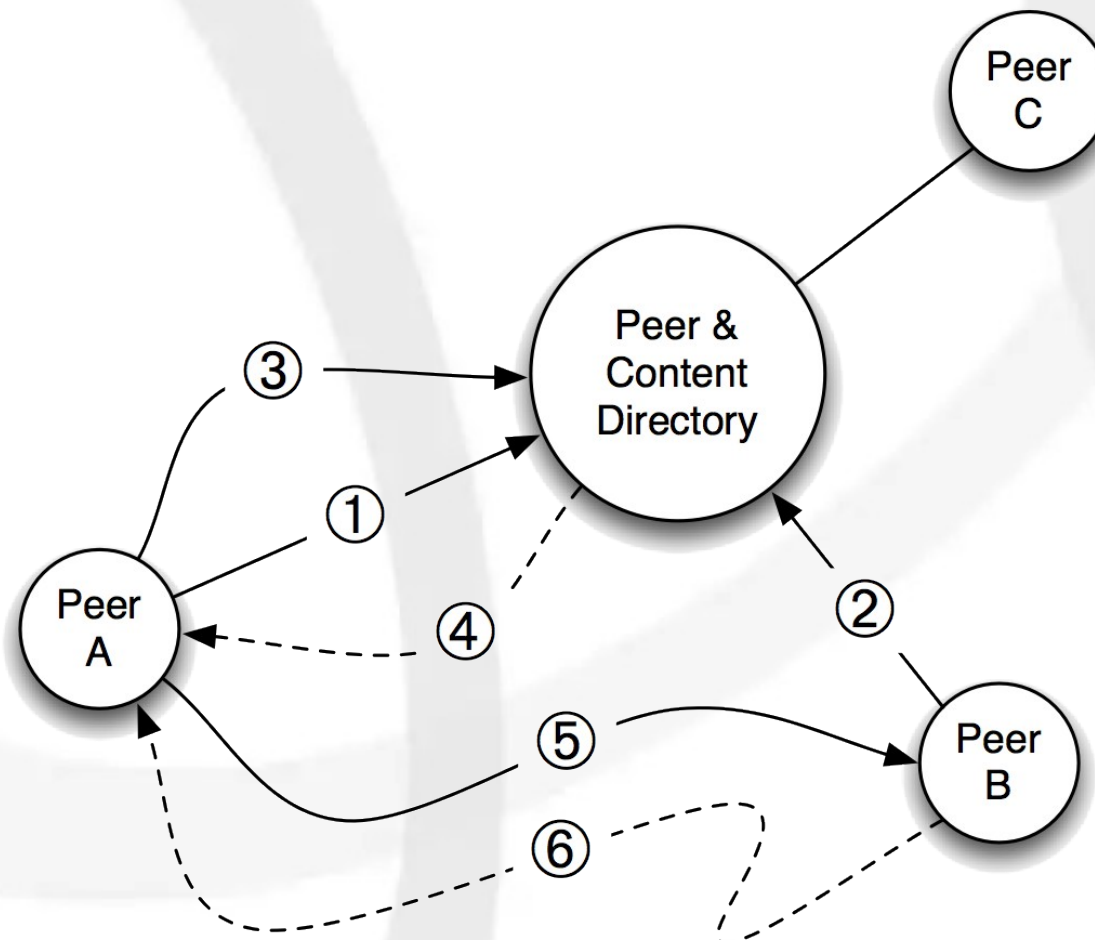


- Google – lições arquiteturais:
 - As decisões surgiram de um profundo conhecimento sobre:
 - O que as aplicações do Google são
 - O que elas demandam
 - Os aspectos chave de *commonality* presentes

Arquiteturas Descentralizadas



- Napster – *Hybrid Client-Server / Peer-to-Peer*



Arquiteturas Descentralizadas

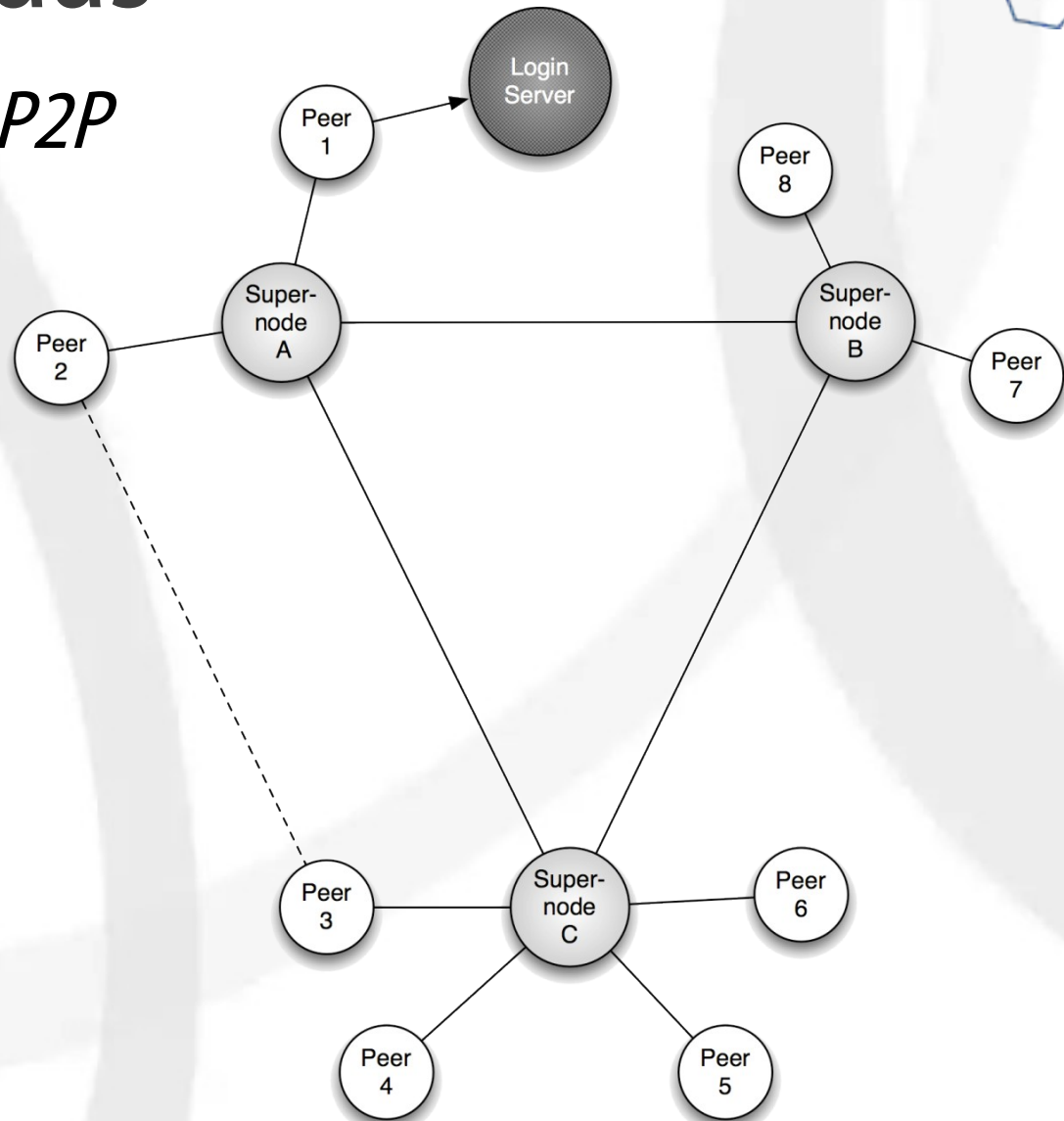


- Napster – *Hybrid Client-Server / Peer-to-Peer*
- Considerações:
 - Qualquer peer atua ora como cliente (solicitando informações sobre músicas) ora como servidor (enviando a música ao solicitante)
 - Uso de protocolo proprietário para as interações entre peers e entre um peer e o diretório de conteúdo (limitando o tipo de arquivos a .mp3)
 - Uso do HTTP para receber conteúdo de um peer
 - Uma música altamente desejada sobrecarregaria o diretório de conteúdo
 - O diretório de conteúdo é um ponto único de falha

Arquiteturas Descentralizadas



- Skype – *Overlayered P2P*



Arquiteturas Descentralizadas



- Skype – *Overlayered P2P*
- Considerações:
 - Não há implementações open-source, o protocolo é proprietário e secreto. Binários são obtidos somente de *skype.com*
 - Inicialmente o usuário se registra/conecta no servidor de login do Skype e recebe um IP de um supernode. A partir daí a comunicação é P2P
 - Quando deseja-se verificar quem está on-line ou realizar uma ligação o peer emite uma consulta a um supernode
 - O supernode retorna o IP desejado ou repassa a requisição para outro supernode

Arquiteturas Descentralizadas



- Skype – *Overlayered P2P*
- Considerações:
 - O servidor de *login* está sob autoridade da *skype.com*
 - Os *supernodes*, entretanto, são *peers* convencionais que foram “promovidos” a supernodes devido a um bom histórico de conectividade de rede e poder de processamento

Arquiteturas Descentralizadas



- Skype – *Overlayered P2P*
- Lições arquiteturais:
 - Arquitetura híbrida (*client-server / P2P*) → otimização do problema da descoberta de recursos
 - Replicação e distribuição dos diretórios, sob a forma de *supernodes* → melhor escalabilidade e robustez
 - “Promoção” de *peers* ordinários a supernodes → outro aspecto do desempenho: não é qualquer peer que se torna um supernode. Pode-se adicionar mais supernodes a depender da demanda
 - Protocolo proprietário com criptografia → privacidade
 - Restrição a clientes obtidos somente no *skype.com* e implementados de modo a impedir inspeções ou modificações → ausência de clientes maliciosos

Arquiteturas Descentralizadas



- BitTorrent – *Resource Trading P2P*
 - Arquitetura especializada para atender metas particulares
 - Meta principal: suportar a replicação rápida de arquivos grandes em *peers* individuais, sob demanda
 - Estratégia: tentar maximizar o uso de todos os recursos disponíveis de modo a minimizar a sobrecarga de um participante específico (o que não acontece no Napster e Gnutella), melhorando a escalabilidade
 - Um *peer* recebe o arquivo em partes, obtidas de diferentes *peers* e re-integradas ao final
 - Um *peer* faz o *download* e, ao mesmo tempo, pode já fornecer as partes que ele possui:
 - Contexto = muitos *peers* simultaneamente interessados em obter uma cópia do arquivo

Arquiteturas Descentralizadas

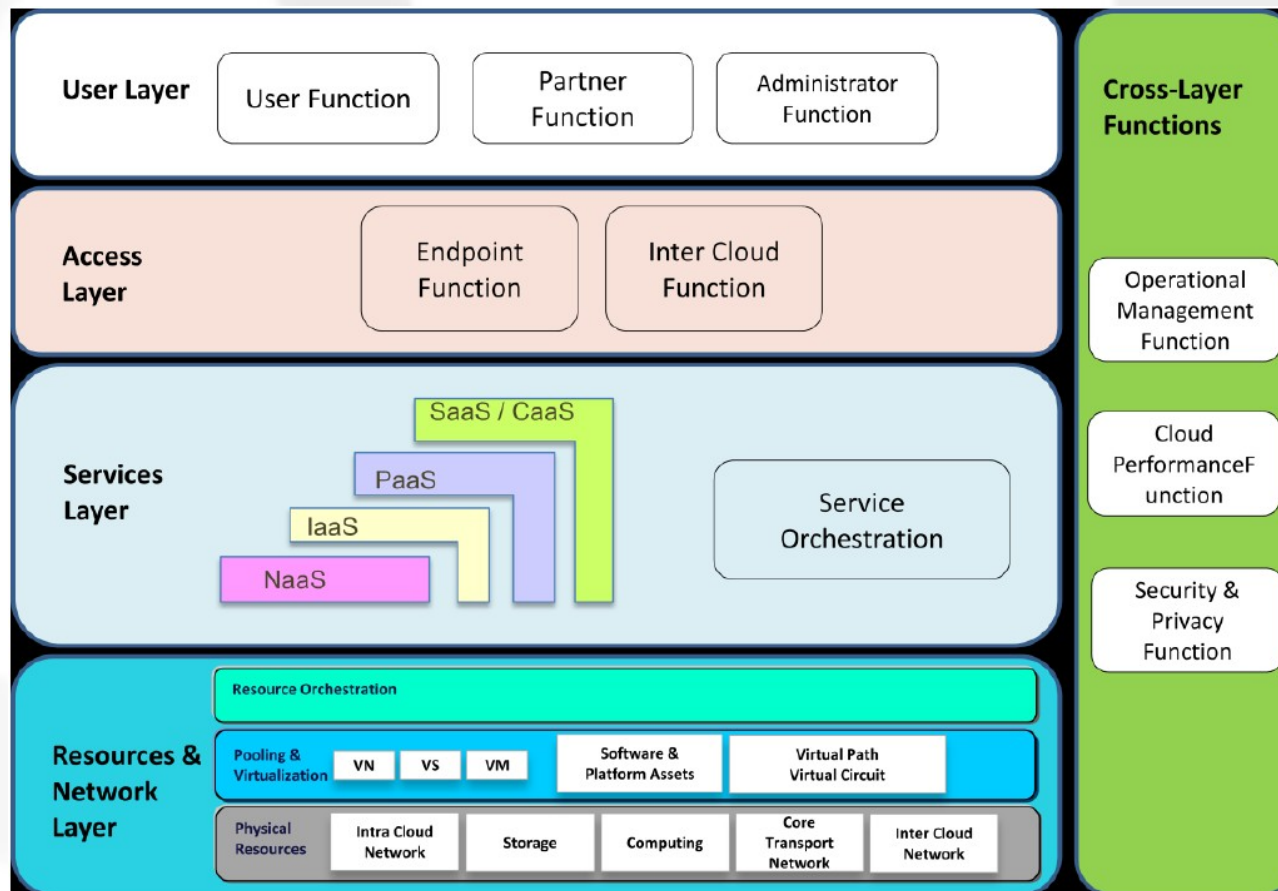


- BitTorrent – *Resource Trading P2P*
- Lições arquiteturais:
 - A responsabilidade da descoberta de conteúdo está fora do escopo do BitTorrent
 - Uma máquina centralizada (*tracker*) coordena a entrega de um arquivo a um conjunto de *peers* interessados. Entretanto, esta máquina não realiza transferências
 - *Peers* interagem com o *tracker* para identificar os outros peers com os quais eles se comunicam para realizar o *download*
 - Meta-dados descrevem como o arquivo é dividido, os atributos de cada parte e a localização do *tracker*
 - Cada *peer* determina i) a próxima parte a ser obtida e ii) de qual *peer* obter a parte
 - Todo *peer* conhece quais *peers* contêm quais partes do arquivo
 - Se um *peer* só realiza *download*, sem disponibilizar as partes para *upload*, sua prioridade de obtenção de partes é reduzida

Arquiteturas Descentralizadas



- Cloud computing – arquiteturas de referência - ITU



Arquiteturas Descentralizadas



- Cloud computing – arquiteturas de referência - IBM

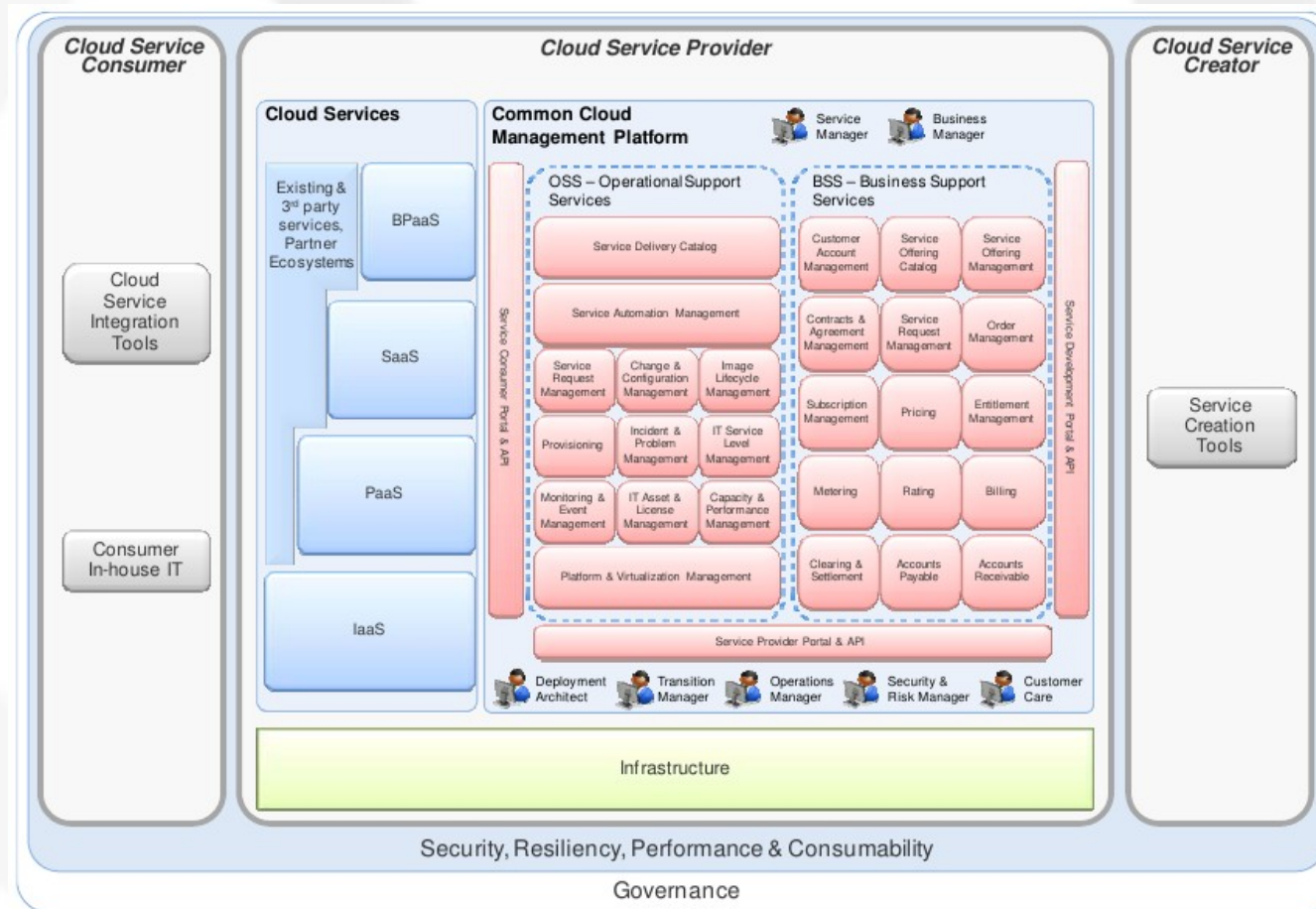


Figure 5: IBM CC RA – CCMP Details



Pós-Graduação em Computação Distribuída e Ubíqua

INF612 - Aspectos Avançados em Engenharia de Software
Arquitetura de Software

Sandro S. Andrade
sandroandrade@ifba.edu.br