UNIDADE 4 – DOCUMENTANDO A ARQUITETURA DE SOFTWARE
MÓDULO 1 – A UML NA DOCUMENTAÇÃO DA ARQUITETURA DE SOFTWARE

01

1 - CONTEXTO DA DOCUMENTAÇÃO DA ARQUITETURA DE SOFTWARE

Após estudarmos todos os aspectos referentes à arquitetura de software, vamos nos concentrar na sua documentação. Nosso foco será o que deve ser documentado, onde a UML se encaixa nesta documentação e como ela é usada para descrever a arquitetura de software.

A documentação da Arquitetura muitas vezes é uma questão que requer especial atenção em um projeto de TI. É comum que haja pouca ou nenhuma documentação referente à arquitetura em muitos projetos. Muitas vezes, quando existe alguma documentação, ela está desatualizada, é inadequada ou, basicamente, não é muito útil.

No outro extremo, há projetos que têm uma grande massa de informação de arquitetura capturada e documentada em diferentes documentos e ferramentas de design.

Diante deste cenário, fica claro que documentar arquitetura não é uma tarefa simples. Mas há boas razões para que façamos a documentação das nossas arquiteturas, por exemplo:

Outros envolvidos podem compreender e avaliar o design. Isso inclui qualquer um dos intervenientes da aplicação, outros membros da equipe de design e da equipe de desenvolvimento.
Podemos entender o projeto quando o retomarmos depois de um período de tempo.
A equipe de desenvolvimento pode aprender com a arquitetura digerindo o pensamento por trás do design.
Podemos fazer análises sobre o design para avaliar o seu provável desempenho, ou para gerar padrões de métricas, como acoplamento e coesão.

02

Alguns fatores **dificultam a documentação** da arquitetura de *software*, tais como:

- Não há um padrão de documentação de arquitetura universalmente aceito.
- Uma arquitetura pode ser complexa e documentá-la de forma compreensível é demorado e não trivial.
- Uma arquitetura tem muitos pontos de vista possíveis. Documentar todos os potencialmente úteis é demorado e caro.

Vale ressaltar que uma arquitetura muitas vezes evolui à medida que o sistema é desenvolvido de forma incremental e mais abordagens sobre o domínio do problema são acrescentadas. Guardar a arquitetura atual do sistema é muitas vezes uma atividade negligenciada, especialmente com a pressão para a entrega de um projeto.

03

2- O que documentar

Provavelmente o elemento mais importante do documento é a **complexidade da arquitetura** sendo projetada. Uma aplicação cliente-servidor em duas camadas com uma lógica de negócio complexa pode ser muito simples arquitetonicamente. Pode-se exigir um diagrama que descreva os principais componentes, e uma visão estrutural dos componentes principais, além de uma descrição do esquema de banco de dados, gerada automaticamente por ferramentas de banco de dados. Documentação deste nível é rápida para ser produzida e rotineira para ser descrita.

Outro fator a considerar é a **longevidade da aplicação**. Será que o sistema irá atender a um negócio de longo prazo, ou está sendo construído para lidar com uma necessidade de integração, ou é apenas um *tapa buracos* até que o sistema definitivo seja instalado?

Projetos com poucas perspectivas de uma vida longa provavelmente não necessitam de uma grande quantidade de documentação. Ainda assim, isso não pode ser uma desculpa para eliminar algum código ou boa prática de *design*. Às vezes, esses sistemas paliativos sobrevivem por muito mais tempo do que o inicialmente previsto.

O próximo fator a considerar são as **necessidades das várias partes interessadas no projeto**. A documentação da arquitetura desempenha um importante papel na comunicação entre os vários membros da equipe do projeto, incluindo arquitetos, *designers*, desenvolvedores, testadores, gerentes de projeto, clientes, organizações parceiras e assim por diante.

04

Em uma equipe pequena, na qual a comunicação interpessoal é boa, a documentação é mínima, e talvez até mesmo mantida em um mural usando técnicas de desenvolvimento ágil. Em equipes maiores e especialmente quando os grupos não são colocados nos mesmos escritórios ou prédios, a

documentação de arquitetura se torna de vital importância para descrever os elementos de *design*, tais como:

- Interfaces de componentes;
- Restrições de subsistemas;
- Cenários de teste;
- Decisões de compra de componentes de terceiros;
- Estrutura da equipe e dependências do cronograma;
- Serviços externos a serem oferecidos pelo aplicativo.

Então, não há nenhuma resposta simples aqui.

Documentação leva tempo para se desenvolver e custa dinheiro. Portanto, é importante pensar cuidadosamente sobre que tipo de documentação será mais útil dentro do contexto do projeto, e procurar produzir e manter estes documentos de referência como chave para o projeto.

05

3 - A UML na documentação da Arquitetura de *Software*

Existe também a questão de **como documentar** uma arquitetura. Naturalmente, existem muitas maneiras de descrever as várias visões de arquitetura que podem ser úteis em um projeto.

Nos últimos anos, a Unified Modeling Language (UML) tornou-se a linguagem de descrição de *software* predominante em toda gama de domínios de desenvolvimento de *software*, apesar de todas as suas forças e fraquezas serem muito debatidas. A qualidade e o baixo custo de suporte da ferramenta a tornam facilmente acessível e utilizável para arquitetos de *software*, *design*ers, desenvolvedores, estudantes, enfim, todos os possíveis envolvidos.

A UML é a notação mais popular para documentar projetos de *software*. Os principais **pontos fortes** desta linguagem são:

- Grande variedade de modelos, permitindo apresentar diferentes pontos de vista,
- Amplo suporte de ferramentas,
- Adoção generalizada.

A UML fornece a seus usuários uma grande variedade de construções e conceitos de modelagem. Um verdadeiro debate se seguiu ao longo dos anos, sobretudo em como se daria a adequação da UML para a modelagem de sistemas de *software* no nível de arquitetura. As primeiras versões da UML tinham um foco maior no *design* da construção, com elementos como componentes e conectores. A versão 2.0 expandiu significativamente a UML para fornecer um suporte muito melhor para o nível superior das construções arquitetônicas. Os pontos de vista existentes foram aprimorados com novos elementos.

06

A UML 2.0 também contribuiu fortemente para a utilização desta linguagem na documentação da arquitetura de *software*. Com seus novos recursos a UML:

- Ajuda a eliminar a ambiguidade dos modelos, permitindo maior inteligibilidade;
- Proporciona melhor suporte ao desenvolvimento conduzido por modelo, em que modelos UML são usados para geração de código.

As notações de modelagem UML 2.0 cobrem os aspectos estruturais e comportamentais de sistemas de *software*. Resumidamente, os diagramas de estrutura definem a arquitetura estática de um modelo, e, especificamente, são:

- Diagramas de classe;
- Diagramas de componentes;
- Diagramas do pacote;
- Diagramas de implantação;
- Diagrama de objetos;
- Diagramas de estrutura composta;

Diagramas de classe

Mostram as classes no sistema e seus relacionamentos.

Diagramas de componentes

Descrevem a relação entre os componentes com interfaces bem definidas. Os componentes compreendem tipicamente várias classes.

Diagramas do pacote

Dividem o modelo em grupos de elementos e descrevem as dependências entre eles.

Diagramas de implantação Mostram componentes e outros artefatos de *software* e como os processos são distribuídos no hardware.
Diagrama de Objetos Descrevem como os objetos relacionam-se e são usados em tempo de execução. Estes são frequentemente chamados de diagramas de instância.
Diagramas de estrutura composta Mostram a estrutura interna de classes ou componentes em termos de seus objetos compostos e os seus relacionamentos.

07

Em contraste, outros diagramas mostram **interações** e **alterações de estado** que ocorrem como elementos no modelo de execução:

- Diagramas de atividades;
- Diagramas de comunicação;
- Diagramas de sequência;
- Diagramas de máquina de estados;
- Diagrama Visão Geral da Interação;
- Diagramas de temporização;
- Diagramas de Caso de Uso.

A ampla gama de diagramas da UML a torna uma opção atraente para a modelagem de todos os tipos de sistemas de *software*. No entanto, é de vital importância para arquitetos não ficarem presos à UML como a única solução de modelagem de arquiteturas, ou a superestimar as vantagens proporcionadas por seu uso.

A UML permanece principalmente focada no *design*, com um **foco inerente a sistemas orientados a objeto**. Alguns diagramas se estendem para outras atividades do ciclo de vida, captando alguns aspectos de requisitos. O apoio para atividades como testes, manutenção e gestão é mínimo. Decisões de arquitetura podem afetar qualquer um desses aspectos do desenvolvimento; é por isso que as notações adicionais devem ser utilizadas para capturar uma arquitetura completa.

Diagramas de atividades Utilizados para a definição de processos lógicos de um programa e do negócio.
Diagramas de comunicação Descrevem a sequência de chamadas entre objetos em tempo de execução.

Diagramas de sequência Mostram a sequência de mensagens trocadas entre os objetos.
Diagramas de máquina de estados Descreve os estados e eventos, e as condições que causam as transições de estado.
Diagrama Visão Geral da Interação São semelhantes aos diagramas de atividades, destinados a mostrar o fluxo de controle através de um número de cenários mais simples.
Diagramas de temporização Combinam essencialmente sequência e diagramas de estado para descrever vários estados de um objeto ao longo do tempo e de mensagens que alteram o estado do objeto.
Diagramas de Caso de Uso Tratam das interações de captura entre o sistema e seu ambiente, incluindo usuários e outros sistemas.

08

Compreender a semântica dos diagramas da UML é a chave para torná-los úteis para descrever as decisões de *design* de arquitetura. Por exemplo, a figura abaixo mostra um **diagrama de componentes** simples com dois componentes. A seta indica que o componente de cálculo é dependente do componente de dados da loja. No entanto, isso pode significar uma série de coisas, incluindo:

- Um elemento de cálculo chama-se Data Store.
- Instâncias de cálculo contêm um ponteiro ou referência a uma instância do Data Store.
- Cálculo exige Data Store para compilar.
- Implementação de cálculo tem um método que usa uma instância de implementação do Data Store como um parâmetro.
- Cálculo pode enviar mensagens para Data Store.

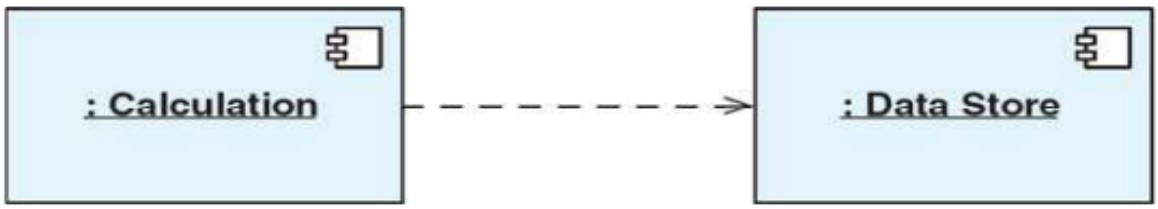


Diagrama de componentes mostrando uma dependência entre dois componentes

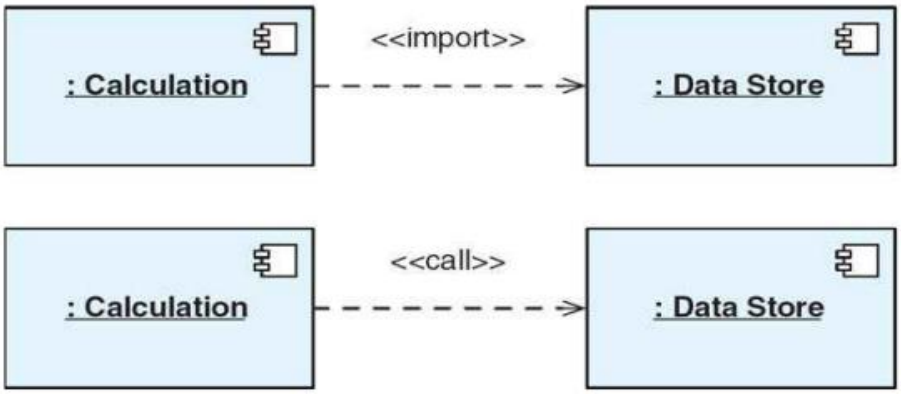
Esta abordagem dá à UML grande flexibilidade, mas limita a sua precisão semântica. Felizmente, a UML inclui instalações que permitem que seus usuários definam novos atributos e restrições que podem ser

aplicados a elementos existentes para especializá-los.

09

A figura a seguir mostra duas variantes do diagrama de componentes na figura anterior. Cada um deles usa um *estereótipo* para fornecer detalhes adicionais sobre o sentido da seta de dependência. Isso não torna a dependência completamente inequívoca, no entanto ainda depende de interpretação do leitor para os estereótipos de "importação" ou "chamada".

O diagrama superior indica que Cálculo importa Data Store. O outro diagrama indica que Cálculo chama Data Store.



Diagramas de componentes usando estereótipos

Diagramas UML sozinhos não carregam informações suficientes para interpretá-los completamente. Os participantes do projeto podem fazer acordos entre si sobre como interpretar aspectos particulares do diagrama UML em um projeto. Os participantes também devem considerar fortemente definir ou selecionar um perfil com interpretações documentadas para os estereótipos, valores atribuídos e restrições.

10

3.1- Resumo da avaliação da UML para a documentação de Arquitetura

- Escopo e Finalidade
- Elementos Básicos
- Estilo

- Aspectos Estáticos e Dinâmicos
- Modelagem Dinâmica
- Aspectos Não Funcionais
- Ambiguidade
- Acurácia
- Precisão
- Visões
- Visão de Consistência

Escopo e Finalidade Capturar decisões de *design* para um sistema de *software* usando até treze diferentes tipos de diagramas.
Elementos Básicos Classes, associações, estados, atividades, nós de composição, restrições entre outros.
Estilo Limitações estilísticas pode ser expressa sob a forma de restrições ou por fornecimento de modelos (parcial) em um dos muitos pontos de vista.
Aspectos Estáticos e Dinâmicos Inclui uma série de diagramas para modelar aspectos estáticos (por exemplo, diagrama de classe, diagrama de objeto, diagrama do pacote) e aspectos dinâmicos (por exemplo, diagrama de estado, diagrama de atividades) do sistema.
Modelagem Dinâmica Depende do ambiente de modelagem; na prática, muito poucos sistemas são vinculados diretamente a um modelo UML de tal forma que o modelo UML necessita ser atualizado ao longo do projeto.

Aspectos Não-Funcionais Sem o apoio direto.
Ambiguidade Em geral, elementos UML podem significar coisas diferentes em contextos diferentes. Ambiguidade pode ser reduzida através da utilização de princípios de perfis UML, incluindo estereótipos e restrições.
Acurácia Não existe um padrão para avaliar a precisão de um modelo UML.
Precisão Modeladores podem escolher um nível adequado de detalhe; UML oferece grande flexibilidade a este respeito.
Visões Cada tipo de diagrama representa, pelo menos, um ponto de vista possível; através de sobrecarga ou de particionamento, um tipo de diagrama pode ser usado para capturar múltiplos pontos de vista.
Visão de Consistência É fornecido muito pouco apoio para verificação da consistência entre os diagramas.

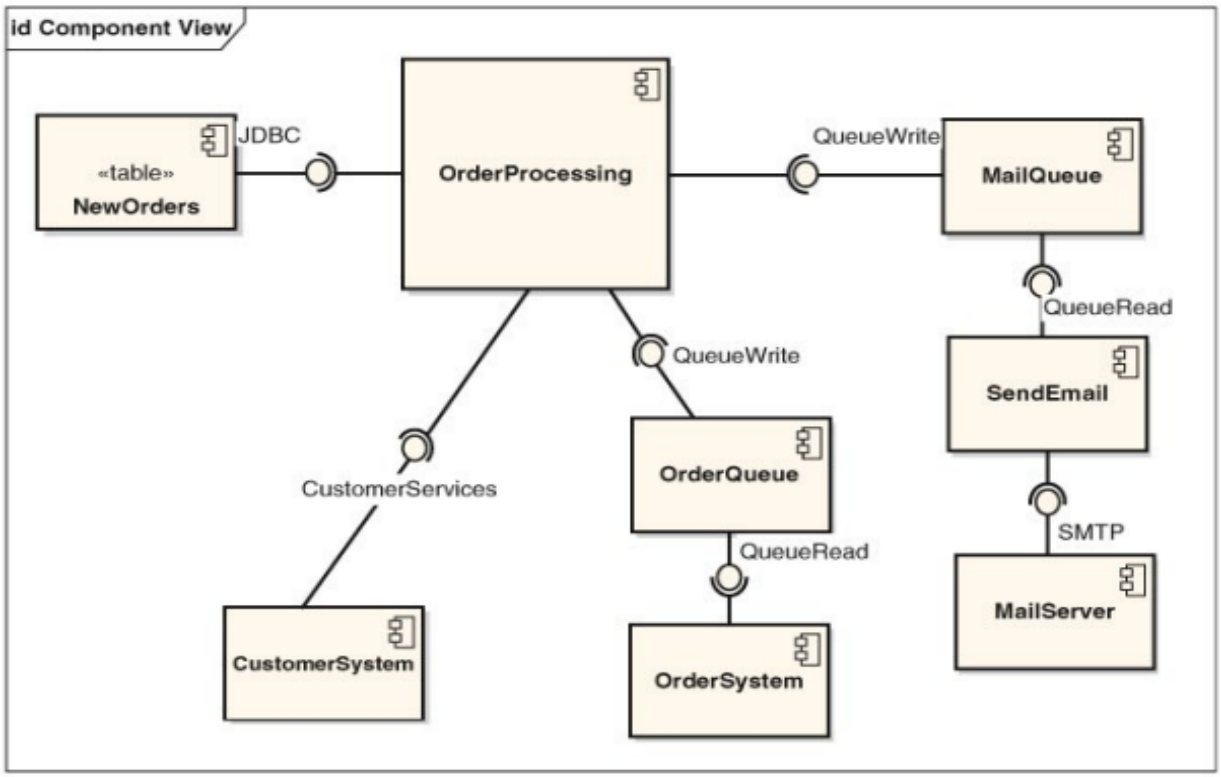
11

3.2 – A importância do Diagramas de Componentes para a documentação da Arquitetura

Os diagramas de componentes são muito úteis para esboçar a estrutura de uma arquitetura de aplicação. Eles mostram claramente as principais partes do sistema e podem mostrar que tecnologias serão utilizadas, bem como os novos componentes que necessitam ser construídos.

Notações introduzidas na UML 2.0 possuem melhorias para representação das interfaces de componentes. Vale lembrar que uma interface é um conjunto de métodos que suporta um componente. Estes estão ilustrados na figura a seguir.

Definição de interface é particularmente importante em uma arquitetura, já que permite que equipes independentes de desenvolvedores projetem e construam seus componentes de forma isolada, garantindo que eles apoiem os contratos definidos por suas interfaces.



Representação de interfaces no exemplo de processamento de pedidos

Ao ligar as interfaces fornecidas e necessárias, os componentes podem ser conectados entre si, conforme exibido na figura. As interfaces fornecidas são nomeadas e capturadas as dependências entre os componentes.

A UML 2.0 torna possível refinar ainda mais as definições de interface e descreve como elas são suportadas dentro do contexto de um componente.

12

3.3- Um exemplo de documentação de Arquitetura com UML

UML oferece diferentes visões possíveis para modelar a arquitetura de um sistema. Os participantes do projeto são responsáveis pela escolha de qual ponto de vista será usado e quão precisa será cada visão. Em nosso exemplo, vamos usar o diagrama de estado e de sequência para descrever um sistema.

O diagrama de componentes exemplo é exibido a seguir:

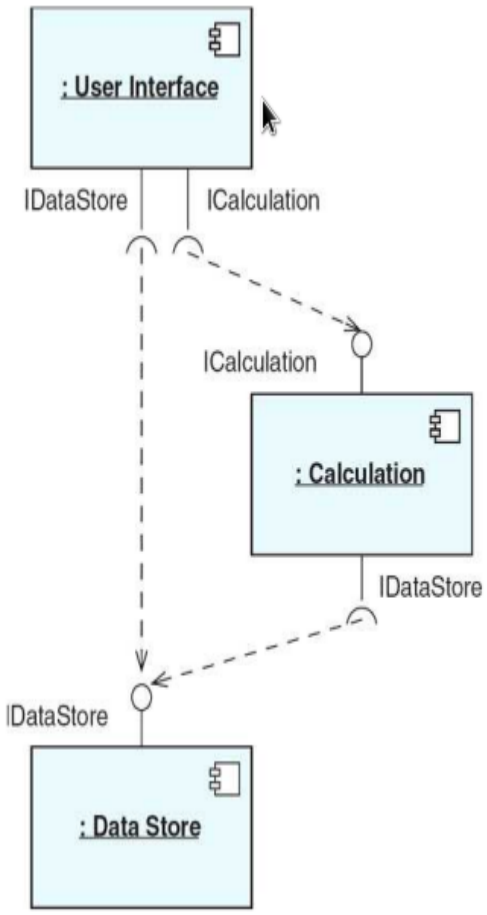


Diagrama componente em UML

Este diagrama tem sintaxe e semântica rigorosas. Os símbolos utilizados são documentados na especificação da UML, e têm significados específicos. A UML define símbolos específicos para os "componentes", que são usados neste exemplo. Não é possível afirmar, entretanto, que o diagrama não apresenta ambiguidade. Por exemplo, o diagrama nada diz sobre o que é um componente neste contexto ou quando e como as chamadas entre os componentes são feitas. Alguns desses detalhes podem ser especificados em outros diagramas UML.

13

O comportamento do sistema pode ser especificado em um **diagrama de estados UML**, como mostrado na figura abaixo. O estado inicial é indicado pelo círculo e o estado final é indicado pelo círculo delineado. Cada retângulo arredondado representa um estado do sistema, e as setas representam transições entre os estados. As condições em colchetes indicam condições de guardas que restringem quando pode ocorrer transições de estado.

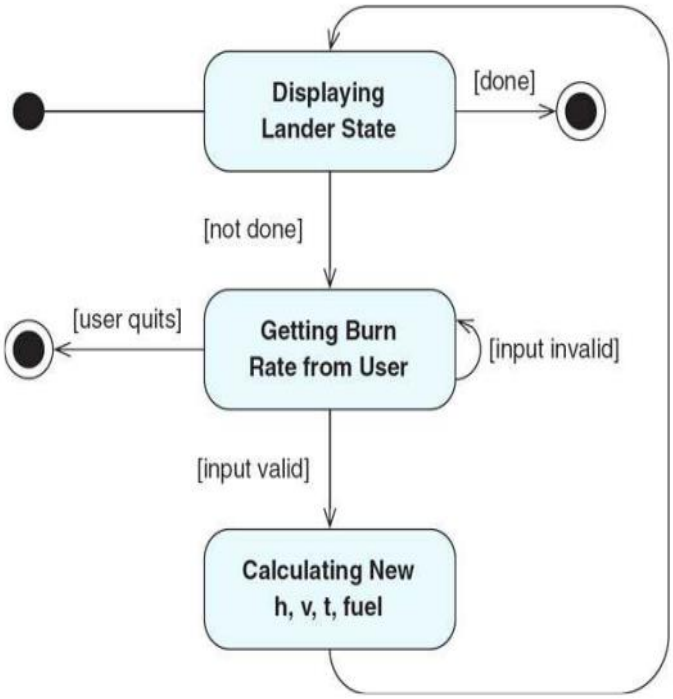


Diagrama de estados em UML

Este gráfico de estado indica que o sistema começa exibindo o estado "lander". Se a simulação é feita, o simulador irá parar. Caso contrário, o sistema irá solicitar uma velocidade de combustão a partir do utilizador. Aqui, o usuário pode optar por terminar o programa no início. Caso contrário, se a velocidade de combustão é válida, o programa irá calcular o novo estado simulador e exibi-lo. Este *loop* de controle irá repetir até que a simulação seja concluída.

Embora este diagrama de estados forneça uma descrição mais rigorosa e formal, ele deixa de fora alguns detalhes como quais componentes executam as ações especificadas. Esta informação pode ser captada num outro diagrama UML, tal como um diagrama de sequência UML.

14

Um diagrama de sequência para a aplicação exemplo é demonstrado na figura a seguir. Este diagrama mostra uma sequência particular de operações que pode ser realizada pelos três componentes.

Fique Atento!

Diagramas de sequência como este não se destinam a captar todo o comportamento de um sistema; ao contrário, eles são usados para documentar determinados cenários ou casos de uso.

O diagrama de sequência exemplo mostra o cenário de usuário que obtém uma velocidade de combustão do usuário, Calcula, Recupera o estado do lander do armazenamento de dados e atualizações das informações e, em seguida, retorna o estado do lander para a Interface do Usuário.

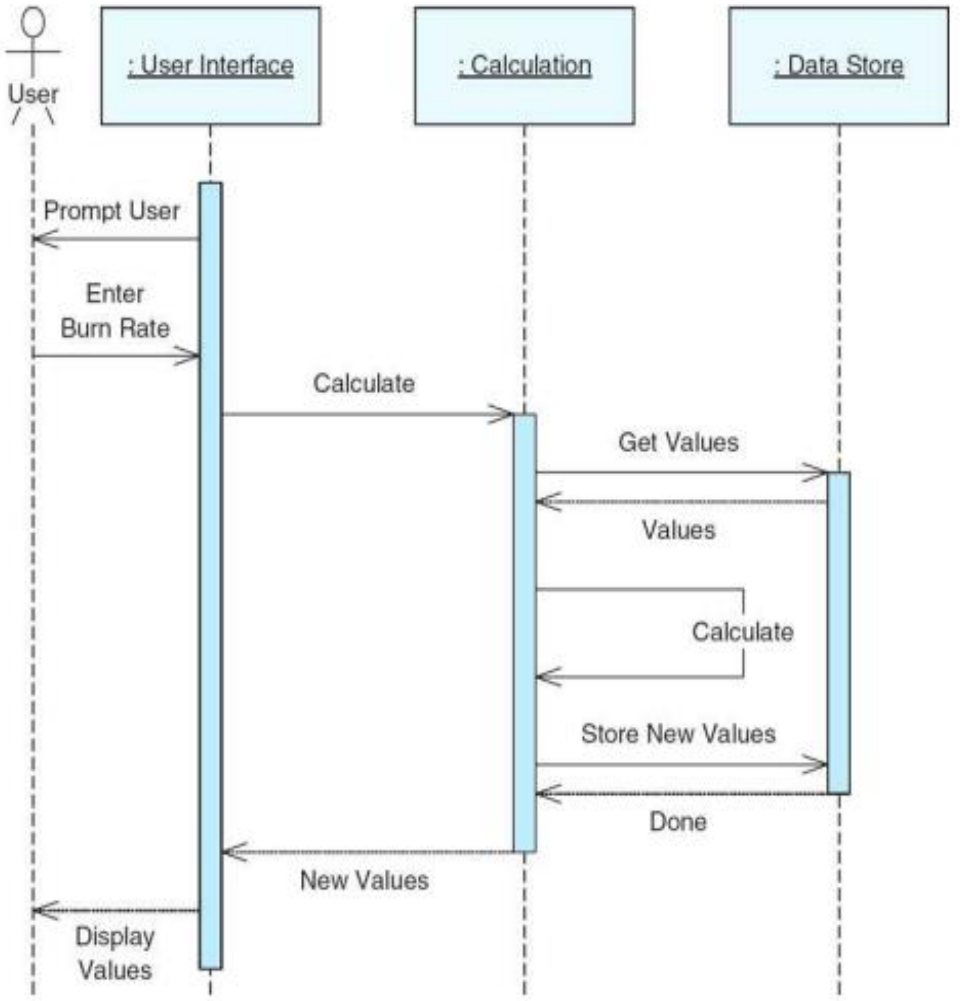


Diagrama de sequência em UML

15

As três figuras anteriores representam três diferentes pontos de vista da arquitetura do nosso sistema exemplo. Estes pontos de vista capturam aspectos tanto estáticos (estruturais) quanto comportamentais do sistema. A pergunta natural a fazer é se **essas visões são consistentes entre si**. Não há uma forma padrão para responder a esta pergunta, porque não há uma noção universal de consistência em UML. Em vez disso, temos que **estabelecer nossos próprios critérios de consistência** e analisar os diagramas com base nesses critérios.

Por exemplo, poderíamos verificar se cada componente no diagrama de componente é representado no diagrama de sequência, e se as chamadas no diagrama de sequência são permitidas pelas setas de dependência no diagrama de componentes.

A verificação da consistência dos diagramas de estados e de sequência é uma tarefa mais difícil. Apesar de ambos apresentarem aspectos comportamentais, estes diagramas estão relacionados com diferentes níveis da arquitetura. O diagrama de estado descreve o funcionamento geral do sistema sem levar em conta sua estrutura, enquanto o diagrama de sequência mostra um cenário em particular num contexto diferente: as interações entre componentes. Neste caso, as inspeções e acordos entre partes interessadas são, provavelmente, a melhor maneira de determinar a consistência dos diagramas.

16

4. Resumo

Documentar a arquitetura é quase sempre uma boa ideia. O truque é fazer apenas o esforço suficiente para produzir a documentação que será útil para as várias partes interessadas do projeto. Isso exige algum planejamento inicial e pensamento. Uma vez que um plano de documentação é estabelecido, os membros da equipe devem comprometer-se a manter a documentação razoavelmente atual, precisa e acessível.

A UML, especialmente a versão 2.0, faz com que seja bastante simples documentar diferentes visões estruturais e comportamentais de um projeto. Com esta linguagem se pode criar um *design* rápido e fácil, e também torna possível capturar grande parte da lógica *design*, as restrições de *design*, e outra documentação baseada em texto dentro do repositório ferramenta.

Além disso, é possível utilizar UML 2.0 de forma flexível em um projeto. Ela pode ser usada para esboçar uma representação abstrata da arquitetura, puramente para fins de comunicação e documentação. Ela também pode ser usada para modelar intimamente os componentes e objetos que vão ser realizados na implementação real.

Dentro da UML 2.0 um diagrama merece especial atenção, o diagrama de componente. Os diagramas de componentes são muito úteis para esboçar a estrutura de uma arquitetura de aplicação. Eles mostram claramente as principais partes do sistema e podem mostrar que tecnologias serão utilizadas, bem como os novos componentes que necessitam ser construídos.

Como muitas decisões de arquitetura, não há resposta certa ou errada no que se refere à UML na documentação da arquitetura. As soluções precisam ser avaliadas no contexto da sua definição do problema.

UNIDADE 4 – DOCUMENTANDO A ARQUITETURA DE SOFTWARE
MÓDULO 2 – MODELAGEM DA ARQUITETURA DE SOFTWARE

01

1- VISÃO GERAL

Cada sistema de *software* tem uma arquitetura, sendo boa ou não. A arquitetura é o conjunto das principais decisões de *design* sobre um sistema. Uma vez que as decisões de *design* foram feitas, elas devem ser documentadas. As decisões de projeto são capturadas em **modelos***.* O processo de criação de modelos é chamado de **modelagem***.*

1.1- Definições

Um **modelo de arquitetura** é um artefato que capta algumas ou todas as decisões de *design* que compõem a arquitetura de um sistema. A **modelagem arquitetônica** é a documentação dessas decisões de *design*.

Os modelos podem capturar as decisões de projeto de arquitetura com diferentes níveis de rigor e formalidade. Eles permitem que os usuários se comuniquem, visualizem, avaliem e evoluam uma arquitetura. Sem modelos, é quase impossível ter uma arquitetura.

Uma **notação de modelagem arquitetônica** é uma linguagem ou meio de captura de decisões de *design*.

Notações arquiteturais de modelagem variam entre os ricos e ambíguos aos semanticamente estreitos e altamente formais. Enquanto alguns modelos estão em conformidade com uma única anotação, um modelo também pode usar uma mistura de notações diferentes. Por exemplo, um único modelo pode usar a UML, que vimos anteriormente, diagrama de classes unificada para descrever as classes de um sistema.

02

2- Conceitos de modelagem

Uma das **decisões** mais importantes que arquitetos e outros interessados farão no desenvolvimento de uma arquitetura é escolher:

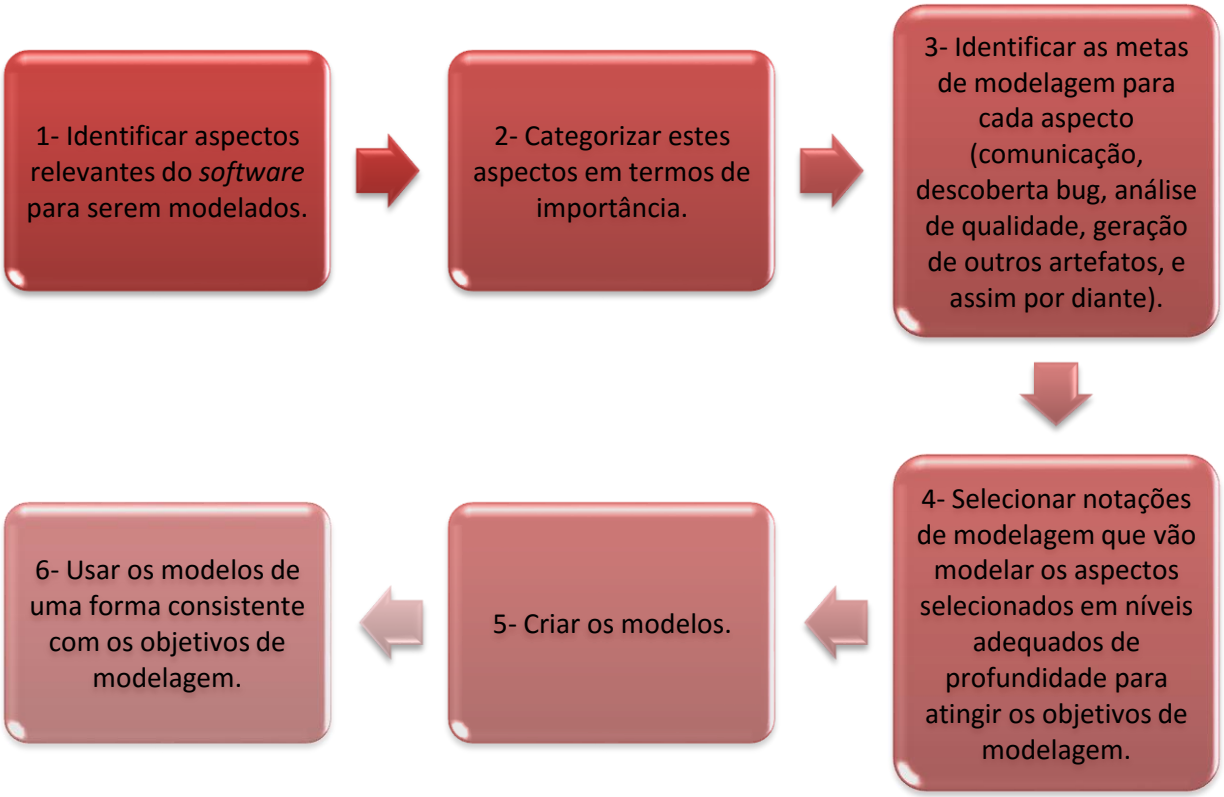
1. As decisões de arquitetura e conceitos que devem ser modelados.
2. O nível de detalhe.
3. A quantidade de rigor ou formalidade.

Estas decisões devem ser baseadas nos **custos** e **benefícios**. Os arquitetos devem equilibrar os benefícios de ter certos modelos em certas formas ou notações com os custos da criação e manutenção desses modelos. Assim, a escolha ocorre entre o *que* modelar, e em *que nível de detalhe*.

As boas práticas recomendam que os aspectos mais importantes ou críticos de um sistema devem ser os que são modelados no maior detalhe com os mais altos graus de rigor / formalidade.

03

As **atividades** básicas por trás da modelagem são:



Embora os passos descritos acima estejam em ordem cronológica, eles podem ser incorporados em um processo iterativo. Aspectos como quais são os objetivos da modelagem e se essas metas são realizáveis utilizando as notações, tecnologia, tempo e dinheiro disponíveis quase nunca são claros no início de um projeto. Assim, executar as tarefas por trás da modelagem de forma recorrente, reavaliando e refinando as estimativas se apresenta como uma boa prática.

2.1- Conceitos básicos de arquitetura

Enquanto cada arquitetura é diferente, conforme já estudado em módulos anteriores, certos elementos são de grande importância quando se fala de projetos arquitetônicos. A seguir vamos explorar estes elementos:

- **Componentes**;
- **Conectores**;
- **Interfaces**;
- **Configurações**.

Esses conceitos formam um ponto de partida para modelagem arquitetônica. No nível mais básico, a modelagem destes conceitos requer uma notação que pode expressar um gráfico de componentes e conectores, preferencialmente com pontos de ligação bem definidos (interfaces).

Numa modelagem para um projeto mais simples estes componentes de nível básico podem atender às necessidades, mas esses modelos não serão suficientes para projetos mais complexos. Dependendo da natureza e do domínio do sistema a ser desenvolvido, representá-lo através de elementos básicos pode não ser uma tarefa simples.

Aplicações grandes, dinâmicas e distribuídas podem ser mais difíceis de modelar. Em vez disso, esses modelos devem ser entendidos com uma série de outros conceitos:

- Como as funções estão divididas entre os componentes?
- Qual a natureza e os tipos de interfaces?
- O que significa para um componente e um conector ser “lincado”?
- Como ficam estas propriedades de uma mudança de sistema ao longo do tempo?

Estas questões estão no centro de modelagem arquitetônica.

Componentes

Os componentes são os blocos de construção de arquitetura que encapsulam um subconjunto de funcionalidades ou dados do sistema.

Conectores Os conectores são blocos de construção de arquitetura que regulam as interações entre componentes.
Interfaces As interfaces são os pontos em que os componentes e conectores interagem com o “mundo exterior”; em geral, outros componentes e conectores.
Configurações As configurações são um conjunto de associações específicas entre os componentes e conectores de arquitetura de um sistema de *software*. Tal associações podem ser capturados através de gráficos cujos nós representam componentes e conectores, e cujas arestas representam a sua interconectividade.

05

2.2- Elementos do estilo arquitetônico

Lembre-se de que um **estilo de arquitetura é uma coleção de decisões de projeto** de arquitetura que são aplicáveis em um determinado contexto de desenvolvimento, restringindo-se a decisões de projeto de arquitetura para um determinado sistema em um contexto específico, e provocam benefícios no sistema resultante. Além da modelagem de elementos arquitetônicos básicos, muitas vezes é útil modelar o estilo que rege a forma como esses elementos têm sido usados.

Note que os estilos arquitetônicos são constituídos por **decisões de *design***. Estas decisões de *design* também podem ser modeladas. Modelar a arquitetura pode ser útil por uma série de razões:

- Reduz a confusão sobre o que é e não é permitido na arquitetura.
- Ajuda a reduzir a dispersão de arquitetura.
- Torna mais fácil distinguir se uma decisão de projeto específico em uma arquitetura foi feita para estar em conformidade com a restrição de estilo ou por algum outro motivo.
- Ajuda a orientar a evolução da arquitetura.
- Pode ser mais viável e útil do que a modelagem estrutural (componentes e conectores gráficos) em sistemas grandes ou dinâmicos.

Os tipos de decisões de *design* encontrados em um estilo arquitetônico são geralmente mais abstratos do que aqueles encontrados em uma arquitetura. Alguns tipos de decisões de *design* que podem ser capturados em um modelo incluem:

- Elementos específicos;
- Componente, conector e interface Tipos;
- Restrições comportamentais;
- Restrições de simultaneidade.

Elementos específicos Um estilo pode determinar que componentes, conectores ou interfaces podem ser usados em situações específicas.
Componente, conector e interface Tipos Tipos específicos de elementos podem ser permitidos, necessários, ou proibidos na arquitetura. Muitas abordagens de modelagem são acompanhadas por um tipo de sistema, embora muitas vezes elas tenham semânticas diferentes.
Restrições comportamentais Restrições sobre o comportamento dos elementos arquitetônicos ou tipos de elementos podem ser incluídos. As restrições podem executar a gama de regras simples e as especificações comportamentais completas.
Restrições de simultaneidade Restrições sobre quais elementos desempenham suas funções em simultâneo e como eles sincronizam o acesso a recursos compartilhados também podem ser incluídos em um estilo arquitetônico.

06

2.3- Aspectos estáticos e dinâmicos

Decisões de *design* de arquitetura podem abordar tanto os aspectos estáticos quanto dinâmicos de um sistema.

Aspectos estáticos de um sistema são aqueles que não envolvem o comportamento do sistema durante sua execução.

Aspectos dinâmicos de um sistema envolvem o comportamento de tempo de execução do sistema.

Aspectos estáticos são geralmente mais fáceis de modelar, simplesmente porque eles não implicam alterações ao longo do tempo.

Aspectos dinâmicos de um sistema pode ser mais difíceis de modelar porque eles têm de lidar com a forma como um sistema se comporta ou com eventuais mudanças ao longo do tempo.

A distinção entre estático e dinâmico, muitas vezes não é uma linha clara. Por exemplo, a estrutura de um sistema pode ser relativamente estável, mas, ocasionalmente, pode alterar-se devido à falha de um componente, o uso de conectores flexíveis, ou dinamismo arquitetônico.

Apoiar modelos dinâmicos é mais difícil do que apoiar modelos estáticos. Uma vez desenvolvidos, modelos estáticos podem ser incorporados num sistema em um modo "somente leitura" que pode ser utilizado como uma base para comparação e análise. Modelos dinâmicos devem ser integrados na execução com o modo “leitura e escrita" do sistema, pois podem mudar. Isso requer suporte de ferramentas para manter o modelo e um sistema sincronizado e consistente.

07

2.4- Aspectos funcionais e não funcionais

Conforme você já estudou, a arquitetura pode capturar os aspectos funcionais e não funcionais de um sistema.

Aspectos funcionais relacionam-se com o que um sistema faz.

Aspectos Não funcionais relacionam-se com a forma com que um sistema executa suas funções.

Uma boa dica para entender essa distinção é que aspectos funcionais de um sistema podem ser descritos usando **frases declarativas**, sujeito-verbo. Exemplo: *O sistema imprime registros médicos.*

Aspectos não funcionais de um sistema podem ser descritos adicionando advérbios a estas frases. Exemplo: *O sistema imprime os registros médicos* ***de forma rápida e confidencial.***

Aspectos funcionais são geralmente mais concretos, mais fáceis de modelar e, muitas vezes, podem ser modelados de forma rigorosa e formal. Modelos funcionais de um sistema capturam os serviços que são prestados por diferentes componentes e conectores, bem como as interconexões que atingem as funções gerais do sistema. Em suma, eles podem capturar o comportamento dos componentes, conectores, ou subsistemas, descrevendo que funções esses elementos executam.

Aspectos não funcionais dos sistemas tendem a ser qualitativos e subjetivos. Modelos de aspectos não-funcionais dos sistemas podem ser mais informais e menos rigorosos do que os modelos funcionais, mas isso não significa que eles não devem ser capturados. Muitas vezes, os aspectos funcionais de sistemas são desenvolvidos especificamente para alcançar os objetivos não funcionais.

08

3- Ambiguidade, Exatidão e Precisão

Arquiteturas de sistemas são abstrações. Elas capturam informações sobre alguns aspectos do sistema e deixam de fora outros aspectos. Idealmente, os aspectos mais importantes de um sistema serão bem definidos pela arquitetura. As partes que são especificadas podem descrever o estado nominal do sistema e deixar de fora estados incomuns. Isso quer dizer que, até certo ponto, arquiteturas normais não são destinadas a contemplar todas as implementações de um sistema. Por conseguinte, as notações usadas para capturar arquiteturas não têm que ser completamente inequívocas e precisas.

Três conceitos-chaves podem ser usados para caracterizar modelos arquitetônicos:

- ambiguidade,
- exatidão e
- precisão.

Apresentaremos a seguir cada um desses conceitos.

09

3.1- Ambiguidade

Um modelo é ambíguo se está aberto a mais de uma interpretação.

Interpretações conflitantes de um modelo podem levar a equívocos, problemas e erros. Em geral, é desejável eliminar a ambiguidade em *design*. Incompletude é a principal razão para a ambiguidade em modelos: quando algum aspecto de um sistema não é determinado claramente, diferentes pessoas podem fazer diferentes suposições sobre como a lacuna deve ser preenchida. A arquitetura é necessariamente incompleta, pois abordam as ***principais*** decisões de *design* sobre um sistema, e não **todas** as decisões de *design*.

Por esta razão, é geralmente impossível eliminar completamente a ambiguidade em modelos arquitetônicos. Além disso, os custos de tentar fazer isso quase sempre superam os benefícios. Portanto, um equilíbrio deve ser atingido. Uma boa prática é permitir que os aspectos ambíguos tenham o consentimento das pessoas envolvidas; todos os envolvidos devem concordar que a arquitetura está "completa o suficiente" e decisões restantes podem ser feitas em atividades de desenvolvimento futuras. Embora esta avaliação prossiga, é útil identificar e documentar aspectos ambíguos da arquitetura como uma espécie de **justificativa de *design***.

10

3.2- Exatidão e Precisão

Muitas definições de exatidão e precisão confundem os dois termos. Aqui, vamos adotar uma interpretação para delineá-los mais claramente.

Exatidão	Precisão
• Um modelo é **exato** se ele estiver correto, conforme a verdade, ou se desvia da correção dentro dos limites aceitáveis.	• Um modelo é **preciso** se é específico, detalhado e exato.

Em termos arquitetônicos, um modelo é **preciso** se transmite a correta informação sobre o sistema modelado. Um modelo é preciso se transmite uma grande quantidade de informações detalhadas sobre o sistema modelado. Pode parecer que esses conceitos caminhem lado a lado, mas na verdade eles são um pouco antagônicos.

No desenvolvimento de uma arquitetura, a exatidão geralmente deve ser favorecida em detrimento da precisão. É desejável que todos os diagramas elaborados sejam exatos, ou seja, corretos, que representem corretamente a arquitetura do sistema. Não necessariamente precisam ser documentados todos os detalhes do sistema. O custo para se detalhar todo o sistema não compensa esta abordagem. Adicionalmente, no decorrer do projeto o detalhamento necessário para a implementação será realizado.

11

3.3- A ilusão de Qualidade

Em última análise, nós usamos arquitetura para garantir que o sistema que estamos a desenvolver alcança a qualidade desejada.

Muitas vezes, as pessoas envolvidas no projeto assumem que o uso de uma determinada abordagem de modelagem irá garantir que o sistema modelado alcance qualidade. Entretanto, na realidade, é possível modelar decisões de *design* corretas e incorretas. Por esta razão, é importante compreender as limitações das notações e abordagens selecionadas para um projeto.

12

4- Resumo

A arquitetura é o conjunto das principais decisões de *design* sobre um sistema. Um modelo de arquitetura é um artefato que capta as decisões de *design* que compõem a arquitetura de um sistema. Já a modelagem arquitetônica é a documentação das decisões de *design*.

Uma notação de modelagem arquitetônica é uma linguagem ou meio de captura de decisões de *design*. As decisões mais importantes que arquitetos farão no desenvolvimento de uma arquitetura são:

- As decisões de arquitetura e conceitos que devem ser modelados.
- O nível de detalhe.
- A quantidade de rigor ou formalidade.

Estas decisões devem ser baseadas nos custos e benefícios.

As atividades básicas por trás da modelagem são:

- Identificar aspectos relevantes do *software* para serem modelados;
- Categorizar estes aspectos em termos de importância;
- Identificar as metas de modelagens para cada aspecto;
- Selecionar notações de modelagem;
- Criar modelos;
- Usar os modelos de forma consistente com os objetivos da modelagem.

13

Certos elementos são de grande importância quando se fala de projetos arquitetônicos. São eles:

- Componentes;
- Conectores;
- Interfaces;
- Configurações.

Um estilo de arquitetura é uma coleção de decisões de projeto de arquitetura que são aplicáveis em um determinado contexto de desenvolvimento.

Os tipos de decisões de *design* encontrados em um estilo arquitetônico são geralmente mais abstratos do que aqueles encontrados em uma arquitetura. Decisões de *design* de arquitetura podem abordar aspectos estáticos e dinâmicos de um sistema, onde o primeiro não envolve comportamento do sistema

ao passo que o segundo envolve o comportamento do sistema.

A arquitetura pode capturar os aspectos funcionais (o que o sistema faz) e não funcionais (forma como o sistema executa suas funções) de um sistema.

Três conceitos-chaves podem ser usados para caracterizar modelos arquitetônicos:

- ambiguidade - um modelo é ambíguo se está aberto a mais de uma interpretação;
- exatidão – o sistema está correto;
- precisão – o sistema é específico, detalhado e exato.

Deve-se tentar diminuir as ambiguidades e a exatidão geralmente deve ser favorecida em detrimento da precisão.

É importante compreender as limitações das notações e abordagens selecionadas para um projeto para que o sistema possa alcançar a qualidade desejada.

UNIDADE 4 – DOCUMENTANDO A ARQUITETURA DE SOFTWARE
MÓDULO 3 – OUTRAS LINGUAGEM PARA DOCUMENTAÇÃO DE ARQUITETURA

01

1- A ARCHITECTURE ANALYSIS AND DESIGN LANGUAGE (AADL)

Adicionalmente à UML, existem outras formas de representar a arquitetura de um *software*. Nesta etapa do nosso estudo vamos explorar algumas linguagens que têm este objetivo. São elas:

- Architecture Analysis and *Design* Language (AADL)
- A Architecture Description Markup Language (ADML)
- xADL - XML Extensible Architecture Description Language.

A **Architecture Analysis and *Design* Language** (AADL) é uma linguagem de descrição para especificar arquiteturas de sistema.

A finalidade inicial da AADL era modelar sistemas de aviação, mas a notação não está ligada especificamente a esse domínio. Em vez disso, ela contém construções úteis e capacidades de modelagem de uma ampla variedade de sistemas embarcados e em tempo real, tais como sistemas automotivos e médicos.

A AADL pode descrever a estrutura de um sistema como um conjunto de componentes, embora esta linguagem tenha disposições especiais para descrever tanto *hardware* como elementos de *software*, e da atribuição de componentes de *software* para *hardware*. Ela também pode capturar aspectos não funcionais de componentes (tais como segurança e atributos de confiabilidade).

02

Sintaticamente, AADL é primariamente uma linguagem textual, acompanhada por uma visualização gráfica e um perfil UML para capturar informação de maneiras diferentes. A sintaxe da linguagem textual é definida usando Backus-Naur Form (BNF) - regras de produção.

AADL define uma série de categorias (ou tipos) de componentes; estes podem ser *hardware* (por exemplo, memória, dispositivo, processador, ônibus), *software* (por exemplo, dados, subprograma, linha, grupo de discussão, o processo), ou composto (por exemplo, sistema). A categoria de um componente prescreve que tipos de *propriedades* podem ser especificadas sobre um tipo de componente.

AADL é suportada por uma base crescente de ferramentas, incluindo um conjunto de *plug-ins* open-source para o ambiente de desenvolvimento de *software* como o Eclipse, que fornece suporte e de

importação / exportação através da Extensible Markup Language (XML). Um conjunto adicional de *plug-ins* está disponível para analisar vários aspectos das especificações da AADL, por exemplo, se todos os elementos estão ligados de forma adequada e se o uso de recursos pelos vários componentes excede recursos disponíveis.

Backus-Naur Form

Backus-Naur Form ou Backus Normal Form (BNF) é uma sintaxe usada para expressar gramáticas livres de contexto, ou seja, um modo formal de descrever linguagens formais.

03

1.1- Resumo da avaliação da AADL

Como vimos, a AADL é usada para o modelar a arquitetura de *software* e *hardware* de um sistema embarcado e em tempo real, como, por exemplo, os sistemas de aviação. Este modelo de arquitetura pode então ser usado como uma documentação do projeto, para as análises ou para geração de código, como por exemplo a linguagem UML. Para um melhor entendimento da AADL apresentamos, a seguir, um resumo de suas características:

- **Escopo e Finalidade**

Modelos em multinível de elementos de *hardware* e *software* interligados.

- **Elementos Básicos**

Vários elementos de *hardware* e *software*, como: redes, portas, processos, threads, e muitos outros.

- **Estilo**

Não há suporte explícito.

- **Aspectos Estáticos e Dinâmicos**

Capta aspectos principalmente estáticos de um sistema, embora propriedades possam capturar alguns aspectos dinâmicos.

- **Modelagem Dinâmica**

Não há suporte explícito.

- **Aspectos Não Funcionais**

Propriedades definidas pelo usuário podem capturar aspectos não funcionais.

04

1.2- Exemplo de AADL

A seguir vamos listar um exemplo de uma especificação de uma arquitetura utilizando o AADL:

```
data lander_state_data
```

```
end lander_state_data;
bus lan_bus_type
end lan_bus_type;
bus implementation lan_bus_type.ethernet
properties
Transmission_Time => 1 ms .. 5 ms;
Allowed_Message_Size => 1 b .. 1 kb;
end lan_bus_type.ethernet;
system calculation_type
features
network
: requires bus access lan_bus.calculation_to_datastore;
request_ge t
: out event port;
response_get
: in event data port lander_state_data;
request_store
: out event port lander_state_data;
response_store : in event port;
end calculation_type;
system implementation calculation_type.calculation
subcomponents
the_calculation_processor : processor calculation_processor_type;
```

```
the_calculation_process : process calculation_process_type.one_thread;
connections
bus access network -> the_calculation_processor.network;
event data port response_get -> the_calculation_process.response_get;
event port the_calculation_process.request_get -> request_get;
event data port response_store -> the_calculation_process.response_store;
properties
Actual_Processor_Binding => reference
the_calculation_processor applies to the_calculation_process;
end calculation_type.calculation;
processor calculation_processor_type
features
network : requires bus access lan_bus.calculation_to_datastore;
end calculation_processor_type;
process calculation_process_type
features
request_get
: out event port;
response_get : in event data port lander_state_data;
request_store : out event data port lander_state_data;
response_store : in event port;
end calculation_process_type;
thread calculation_thread_type
features
```

```
request_get
: out event port;
response_get : in event data port lander_state_data;
request_store : out event data port lander_state_data;
response_store : in event port;
properties
Dispatch_Protocol => periodic;
end calculation_thread_type;
process implementation calculation_process_type.one_thread
subcomponents
calculation_thread : thread client_thread_type;
connections
event data port response_get -> calculation_thread.response_get;
event port calculation_thread.request_get -> request_get;
event port response_store -> calculation_thread.response_store;
event data port request_store -> request_store;
properties
Dispatch_Protocol => Periodic;
Period => 20 ms;
end calculation_process_type.one_thread;
```

A primeira coisa a notar sobre este modelo é o **nível de detalhe com o qual a arquitetura é descrita**. Um componente (calculation_type.calculation) é executado em um processador físico (the_calculation_processor), que executa um processo (calculation process type.one thread), que por sua vez contém um único segmento de controle (thread de cálculo), todos os quais podem fazer dois tipos de solicitação-resposta através de portas (request get/response get, request store/response store)

sobre um barramento Ethernet (lan bus type.ethernet). Cada um destes diferentes níveis de modelagem está conectado através de composição, mapeamento de portas, e assim por diante. Este nível de detalhe enfatiza a importância de ferramentas, como editores gráficos, para modelar essas informações em uma forma mais compreensível.

A segunda coisa a notar sobre esta especificação é o **uso de propriedades para definir as características** específicas de alguns dos elementos (por exemplo, o processo de cálculo é executado periodicamente, a cada 20ms). Em uma especificação completa do sistema, essas propriedades seriam ainda mais detalhadas e elaboradas, e poderiam ser usadas por ferramentas para responder a questões sobre a disponibilidade e tempo que são críticos em um contexto em tempo real.

05

2 - A Architecture Description Markup Language (ADML)

A Architecture Description Markup Language (ADML) é uma linguagem de descrição de arquitetura baseada em XML.

Ela foi originalmente desenvolvida pela microeletrônica e Computer Technology Consortium (MCC) e é mantida como um padrão pelo Open Group. Na ADML, os usuários podem definir as propriedades presentes nos elementos do sistema que está sendo descrito. Esta definição é feita através de metapropriedades.

A utilização da Extensible Markup Language (XML) como a base para uma notação de modelagem de arquitetura apresenta alguns benefícios. Entre estes **benefícios**, podemos citar:

- A XML fornece uma estrutura padrão para expressar a sintaxe das notações.
- A XML é adequada para descrever organizações hierárquicas de conceitos, e
- Permite a utilização de referências internas para criar ponteiros de um elemento para outro.

Outra vantagem prática significativa da utilização de XML é a quantidade de ferramentas comerciais e de código aberto que estão disponíveis para análise, manipulação e visualização de documentos XML. Usando essas ferramentas pode-se reduzir significativamente a quantidade de tempo e esforço necessários para a construção de ferramentas centradas em arquitetura, tais como analisadores, editores e assim por diante.

06

2.1- Exemplo de ADML

A seguir vamos listar um exemplo de uma especificação de uma arquitetura utilizando a ADML:

```
<Componente ID = nome de "armazenamento de dados" = "Data Store">
<ComponentDescription>
<ComponentBody>
<Porta ID = "GetValues" name = "getValues" />
<Porta ID = "storeValues" name = "storeValues" />
</ ComponentBody>
</ ComponentDescription>
</ Componente>
</ programlisting>
```

O exemplo acima não fornece uma descrição completa da ADML para uma aplicação. O uso de XML como uma estrutura sintática padrão abre esta especificação em uma matriz muito maior de ferramentas, tais como analisadores e editores que não estariam necessariamente disponíveis de outra forma.

07

2.2- Como a ADML relaciona-se com UML?

Como sabemos, a UML é sobretudo uma linguagem de *design* do sistema, enquanto que a ADML atua principalmente no nível da arquitetura corporativa. No entanto, essa distinção não é simples, porque as diferenças entre arquitetura e *design* não são claras. Sistemas individuais de grande porte muitas vezes têm uma arquitetura, enquanto o desenvolvimento de arquitetura empresarial muitas vezes envolve *design* de alto nível, ou seja, uma forma mais superficial para validar a arquitetura.

Outra diferença é que UML é uma **linguagem gráfica**, enquanto ADML é uma **notação de marcação**, proporcionando uma textual (legível) notação para a descrição da arquitetura. XML está sendo desenvolvido como um meio de troca de modelos UML baseada em XML, e nós acreditamos que há potencial sinergia aqui.

Consideramos que UML e ADML são complementares, mas não existe atualmente um diálogo permanente entre a comunidade ADML dentro do *The Open Group* e da comunidade UML dentro OMG para ajudar ambos os lados entender essa questão melhor.

08

3- xADL: LINGUAGEM BASEADA EM XML EXTENSIBLE ARCHITECTURE DESCRIPTION LANGUAGE

xADL é uma tentativa de fornecer uma plataforma sobre a qual os recursos de modelagem comuns podem ser reutilizados de domínio para domínio e novos recursos podem ser criados e adicionados à linguagem como entidades de primeira classe.

Como ADML, xADL é uma linguagem baseada em XML. Isto é, todos os modelos xADL são um documento XML válido e bem estruturado. A principal diferença é que, ao contrário da ADML, xADL aproveita totalmente os mecanismos de extensibilidade da XML para suas extensões de linguagem.

Sintaticamente, a linguagem xADL é a composição de esquemas xADL. Cada esquema xADL adiciona um conjunto de recursos para a linguagem. Estes recursos podem ser uma nova estrutura de construção de mais alto nível ou podem ser extensões de estruturas já existentes.

A flexibilidade de poder utilizar novas estruturas ou extensões de estruturas já existentes fornece algumas **vantagens**. São elas:

- Permite a **adoção incremental:** os usuários podem criar novas estruturas gradativamente de acordo com a necessidade de seu domínio.
- Permite a **extensão divergente:** os usuários podem estender a linguagem para adequar às suas necessidades.
- Permite a **reutilização:** como os esquemas são definidos em módulos XML, eles podem ser compartilhados entre projetos que precisam de recursos comuns, sem que cada projeto desenvolva esquemas próprios.

De tempos em tempos, novos esquemas são adicionados ao xADL. Estes esquemas são desenvolvidos pelos criadores do xADL ou por seus colaboradores externos.

09

3.1- Resumo da avaliação da xADL

Como vimos, a xADL é uma tentativa de fornecer uma plataforma sobre a qual os recursos de modelagem podem ser reutilizados ou em que novos recursos possam ser criados e adicionados à linguagem.

Para um melhor entendimento da xADL apresentamos, a seguir, um resumo de suas características:

- Escopo e Propósito
- Elementos Básicos
- Estilo
- Estática e Aspectos dinâmicos
- Modelagem Dinâmica
- Aspectos não funcionais
- Ambiguidade
- Exatidão
- Precisão
- Pontos de Vista

Escopo e Propósito Modelando estrutura arquitetura, linhas de produtos, e implementações, com suporte para extensibilidade.
Elementos Básicos Componentes, conectores, interfaces, links, opções, variantes, versões, além de todos os elementos de base definidos nas extensões.
Estilo Aspectos estilísticos de uma arquitetura podem ser modelados por meio do uso de tipos e bibliotecas de tipos.
Estática e Aspectos dinâmicos Estrutura estática é modelada de forma nativa, propriedades dinâmicas podem ser capturada através de extensões.
Modelagem Dinâmica A biblioteca de vinculação de dados permite que especificações xADL sejam manipuladas de forma programática.

Aspectos não funcionais As extensões podem ser escritas para capturar aspectos não funcionais.
Ambiguidade A linguagem xADML é considerada permissiva à medida que permite que seus elementos possam ser utilizados, embora a documentação desta linguagem indique claramente como os elementos devem ser utilizados. Estão disponíveis ferramentas para verificar automaticamente restrições em documentos xADL e permitir que os usuários definam suas próprias restrições.
Exatidão As ferramentas são fornecidas para verificar a exatidão dos documentos xADL; restrições adicionais podem ser escritas para estas ferramentas para lidar com extensões.
Precisão Extensões podem ser utilizadas para definir sobre a utilização de elementos existentes ou sobre a criação de novos componentes.
Pontos de Vista Nativamente, pontos de vista estruturais (tanto tempo de execução e tempo de *design*) são suportados, bem como vistas da linha de produto; as extensões podem ser usadas para fornecer pontos de vista adicionais.

10

3.2- Exemplo de xADL

A seguir vamos listar um exemplo de uma especificação de uma arquitetura utilizando o xADL:

```
id = "userinterface";
description = "UserInterface";
interface{
id = "userinterface.getValues";
description = "User Interface obter valores
Interface";
direction = "out";
}
interface{
ID = "userinterface.calculate";
description = "User Interface Calcular Interface";
direction = "out";
}
}
```

```
ligação{
id = "cálculo-to-datastore-GetValues";
description = "Cálculo para armazenar dados de obter valores"
ponto{
anchorOnInterface {
type = "simples";
href = "# calculation.getValues";
}
}
ponto{
anchorOnInterface {
type = "simples";
href = "# datastore.getValues";
}
}
}
ligação{
id = "cálculo-to-datastore-storevalues";
description = "Cálculo para armazenar dados loja
Valores "
ponto{
anchorOnInterface {
type = "simples";
href = "# calculation.storeValues";
}
}
ponto{
anchorOnInterface {
type = "simples";
href = "# datastore.storeValues";
}
}
} ligação{
id = "ui-to-cálculo-Calculate";
description = "UI para calcular Cálculo"
ponto{
anchorOnInterface {
type = "simples";
href = "# userinterface.calculate";
}
}
ponto{
anchorOnInterface {
type = "simples";
href = "# calculation.calculate";
}
}
```

```
}
ligação{
id = "ui-to-datastore-GetValues";
description = "toto UI Data Store Obter Valores"
ponto{
anchorOnInterface {
type = "simples";
href = "# userinterface.getValues";
}
}
ponto{
anchorOnInterface {
type = "simples";
href = "# datastore.getValues";
}
}
}
}
}
```

11

4- RESUMO

Adicionalmente à UML, existem outras formas de representar a arquitetura de um *software*.

A Architecture Analysis and *Design* Language (AADL) é uma linguagem de descrição para especificar arquiteturas de sistema. A AADL pode descrever a estrutura de um sistema como um conjunto de componentes, embora esta linguagem tenha disposições especiais para descrever tanto *hardware* como elementos de *software*, e da atribuição de componentes de *software* para *hardware*.

A Architecture Description Markup Language (ADML) é uma linguagem de descrição de arquitetura baseada em XML. Na ADML, os usuários podem definir as propriedades presentes nos elementos do sistema que está sendo descrito. Esta definição é feita através de metapropriedades.

Sistemas individuais de grande porte muitas vezes têm uma arquitetura, enquanto o desenvolvimento de arquitetura empresarial muitas vezes envolve *design* de alto nível, ou seja, uma forma mais superficial para validar a arquitetura.

Uma diferença entre UML e ADML é que aquela é uma linguagem gráfica, enquanto esta é uma notação de marcação, proporcionando uma textual (legível) notação para a descrição da arquitetura.

Extensible Architecture Description Language (xADL) é uma tentativa de fornecer uma plataforma sobre a qual os recursos de modelagem comuns podem ser reutilizados de domínio para domínio e novos recursos podem ser criados e adicionados à linguagem como entidades de primeira classe.

UNIDADE 4 – DOCUMENTANDO A ARQUITETURA DE SOFTWARE
MÓDULO 4 – PADRÃO DE DOCUMENTAÇÃO DE ARQUITETURA

01

1- A DOCUMENTAÇÃO DA ARQUITETURA – VISÃO GERAL

É sempre útil para uma organização ter um modelo de documento disponível para a elaboração da documentação do projeto. Modelos de documentos ajudam a reduzir o tempo de início para um projeto, fornecendo estruturas prontas para que possam ser usadas pelos membros.

Uma vez que o uso dos modelos se torna institucionalizado, a familiaridade adquirida com a estrutura do documento ajuda na captura eficiente de detalhes do projeto de design. Os modelos de documentos também ajudam no treinamento de novos funcionários em especial os desenvolvedores que passam a pensar e questionar o desenvolvimento de seus sistemas.

A estrutura de documentação a seguir pode ser utilizada para a captura de uma arquitetura. Para implantar essa template em uma organização, deve-se elaborar um texto explicativo com exemplos do tipo de informação esperada em cada seção.

Template de Documentação de Arquitetura

Nome do Projeto: XXX
Contexto
1. Projeto
2. Requisitos de Arquitetura
    2.1 Visão Geral dos objetivos-chave
    2.2 Casos de Uso de Arquitetura
    2.3 Partes Interessadas Requisitos arquitetônicos
    2.4 Restrições
    2.5 Requisitos não funcionais
    2.6 Riscos
3. Solução
    3.1 Padrões de Arquitetura relevantes
    3.2 Visão Geral da Arquitetura
    3.3 Visão estrutural
    3.4 Visão de Comportamento
    3.5 Questões de Implementação
4. Análise Arquitetura
    4.1 Cenário
    4.2 Riscos

Para entendermos melhor como a documentação da arquitetura funciona, vamos mergulhar na documentação do projeto. Vamos chamar este projeto de **ICDE**.

02

2- REQUISITOS DE ARQUITETURA DO ICDE

Esta seção descreve o conjunto de requisitos do projeto da arquitetura do aplicativo ICDE.

2.1- Visão geral dos principais objetivos

O principal objetivo da arquitetura ICDE é fornecer uma infraestrutura para suportar uma interface de programação para ferramentas de cliente para que terceiros acessem o armazenamento de dados ICDE.

A ICDE deve oferecer:

- Flexibilidade em termos de plataforma e aplicação de implementação / configuração precisa de ferramentas de terceiros.
- Estrutura que permita integrar outras ferramentas ao ambiente do ICDE, obter *feedback* imediato sobre as atividades do usuário do ICDE e forneça informações aos analistas e às outras ferramentas no ambiente.
- Acesso de leitura / gravação simples e conveniente para o armazenamento de dados ICDE.

O segundo objetivo é evoluir a arquitetura ICDE para que ela possa suportar implementações de 100-150 usuários. Isso deve ser feito de forma a oferecer um baixo custo por implantação de estação de trabalho.

A abordagem adotada deve ser coerente com as necessidades dos envolvidos e as limitações e requisitos não funcionais detalhadas nas seções seguintes.

03

2.2- Casos de Uso de Arquitetura

Dois casos de uso básicos sobre o uso da API foram identificados a partir de discussões com um pequeno número de potenciais fornecedores de ferramentas de terceiros.

Estes estão brevemente descritos abaixo:

Acesso a dados	Armazenamento de dados
•Consultas de ferramentas de terceiros se concentram nas atividades de um único usuário. •A sequência de consultas começa por informações sobre trabalho atual do usuário. •Os eventos recuperados são pesquisados no tempo que eles ocorrem, e a lógica do aplicativo procura por itens específicos de dados nos registros recuperados. •Estes valores são usados para qualquer atividade de inicialização na ferramenta de análise de terceiros, ou criam uma saída informativa que aparece na tela do usuário.	•Ferramentas de terceiros precisam ser capazes de armazenar informações no banco de dados do ICDE, para que se possa compartilhar dados sobre suas atividades. •É necessário um mecanismo de notificação para comunicar sobre a disponibilidade de novos dados. •Os dados a partir de cada uma das ferramentas está diversificado em termos de estrutura e conteúdo.

04

2.3- Requisitos de arquitetura dos *stakeholders*

Os requisitos a partir das perspectivas das três principais partes interessadas no projeto são cobertos nas seções a seguir.

2.3.1- Desenvolvedores de ferramenta de terceiros

- **Facilidade de acesso a dados**

O armazenamento de dados do ICDE compreende um componente de *software* moderadamente complexo. O banco de dados relacional tem cerca de 50 tabelas, com algumas inter-relações complexas. No ambiente ICDE v1.0, esta complexidade torna as consultas SQL para recuperar dados não triviais para escrever e testar. Além disso, como os requisitos funcionais evoluem com cada versão, alterações no esquema de banco de dados são inevitáveis, e estes podem quebrar as consultas existentes. Por estas razões, é necessário um mecanismo que torne mais fácil a recuperação de dados úteis por ferramentas de terceiros, bem como uma abordagem para o isolamento das ferramentas de alterações da base de dados. Para que haja integração com ferramentas de terceiros os desenvolvedores não precisarão entender o esquema de banco de dados e escrever consultas complexas.

- **Suporte de plataforma heterogênea**

Várias ferramentas de terceiros desenvolvem tecnologias em plataformas diferentes do Windows. O ICDE *software* v1.0 está intimamente ligado ao Windows. Além disso, o banco de dados relacional utilizado só está disponível na plataforma Windows. A estratégia adotada para o ICDE v2.0 é torná-lo possível de ser executado em sistemas operacionais diferentes do Windows tanto para acesso aos dados, como para a integração com o ambiente.

- **Notificação instantânea de eventos**

As ferramentas de terceiros sendo desenvolvidas tem como objetivo fornecer *feedback* em tempo real para os analistas. A implicação direta disso é que essas ferramentas têm acesso aos eventos registados pelo sistema ICDE à medida que ocorrem. Consequentemente é necessário algum mecanismo para distribuir eventos gerados pelo usuário ICDE.

05

2.3.2- Programadores

Do ponto de vista do programador API ICDE, a API deve:

- Ser fácil e intuitiva.
- Ser fácil de compreender e modificar o código usado pela API.
- Fornecer um modelo de programação conveniente e conciso para a implementação de casos de uso comuns para acessar os dados do ICDE.
- Fornecer uma API para escrever dados específicos de ferramentas e metadados para o armazenamento de dados ICDE. Isso permitirá que múltiplas ferramentas troquem informações através da plataforma ICDE.
- Fornecer a capacidade de atravessar dados ICDE em caminhos de navegação incomuns ou imprevistas. A equipe de *design* não pode prever exatamente como os dados do banco de dados serão utilizados, de modo que a API deve ser flexível e não inibir a criatividade dos desenvolvedores na utilização da ferramenta.
- Fornecer desempenho adequado, de forma ideal uma consulta deve apresentar seu resultado em torno de 5 segundos em uma implantação de *hardware* padrão. Isso vai permitir que os desenvolvedores de ferramentas criem produtos com tempos de resposta previsíveis.
- Ser flexível em termos de opções de implantação e distribuição de componentes. Isso irá torná-la rentável para estabelecer instalações da ICDE em pequenos grupos de trabalho, ou grandes departamentos.
- Ser acessível através de uma API Java.

06

2.3.3- Equipe de Desenvolvimento

Do ponto de vista da equipe de desenvolvimento do ICDE, a arquitetura deve:

- Isolar a estrutura de banco de dados, o mecanismo de implementação das ferramentas de terceiros e a estrutura de armazenamento de dados ICDE e alterações para a estrutura de armazenamento de dados ICDE.
- Oferecer facilidade de modificação com o mínimo impacto sobre o código do cliente ICDE existente que usa a API.
- Apoiar o acesso simultâneo de vários segmentos ou aplicativos ICDE em execução em diferentes processos e / ou em diferentes máquinas.
- Possuir uma documentação capaz de transmitir claramente o uso da API para os desenvolvedores.
- Fornecer desempenho escalável. À medida que a carga aumenta no pedido simultâneo de uma implantação ICDE, deve ser possível dimensionar o sistema sem alterações para a implementação da API. A escalabilidade seria alcançada pela adição de novos recursos de *hardware*, quer aumentando ou dimensionando a implantação.
- Reduzir significativamente ou remover a capacidade de ferramentas de terceiros em causar falhas no servidor, consequentemente, reduzindo o esforço de suporte. Isso significa que a API deve garantir que suas chamadas não irão consumir todos os recursos (memória, CPU) do servidor ICDE, travando assim as outras ferramentas.
- Não ser indevidamente cara para testar. A equipe de teste deve ser capaz de criar um conjunto de testes abrangente que pode automatizar o teste da API ICDE.

07

2.4- Restrições

- O esquema de banco de dados ICDE v1.0 deve ser utilizado.
- O ambiente v2.0 ICDE deve ser executado em plataformas Windows.

2.5- Requisitos Não Funcionais

- **Desempenho**

O ambiente ICDE v2.0 deve fornecer tempo de resposta de 5 segundos para as consultas realizadas através das APIs que recuperam até 1.000 linhas de dados, conforme medido sobre uma plataforma de hardware de implantação "típico".

- **Confiabilidade**

A arquitetura v2.0 ICDE deve ser resiliente a falhas induzidas por ferramentas de terceiros.

- **Simplicidade**

Como requisito para a API temos a simplicidade no *design*, com base na flexibilidade da arquitetura. Isso ocorre porque projetos simples são mais baratos para construir, mais confiáveis e mais fáceis de evoluir para atender às exigências concretas à medida que surgem.

08

2.6 Riscos

Os principais riscos associados com a concepção são os seguintes:

Flexível em termos de facilidade de evolução, ampliação e melhorias, e não incluindo mecanismos que facilitem a adoção de uma arquitetura com diferentes estratégias.

Flexível em termos da gama de recursos sofisticados oferecidos na API para recuperar dados.

09

3- SOLUÇÃO

As seções a seguir descrevem o *design* da arquitetura ICDE.

3.1- Padrões de Arquitetura

Os seguintes padrões de arquitetura são utilizados na concepção:

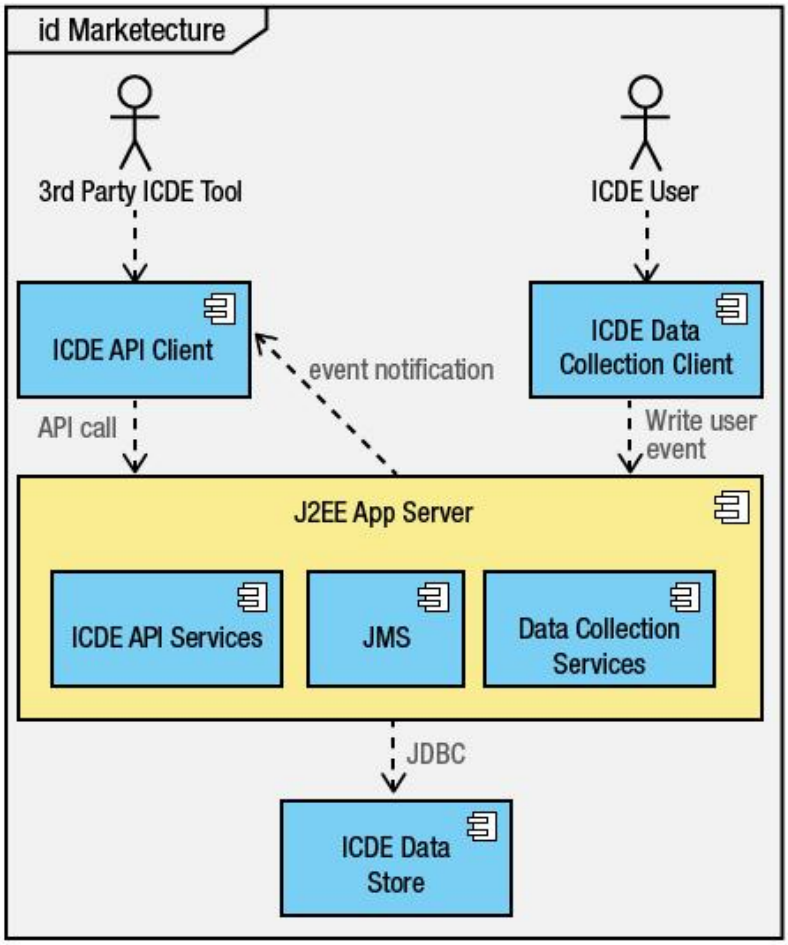
Três camadas	*Publish-subscribe*	*Layered*
•Ferramentas de terceiros são clientes comunicando-se com a implementação da API na camada intermediária, que consulta os dados armazenados pelo ICDE v2.0.	•A camada intermediária contém um recurso de publicação-assinatura.	•Tanto o cliente e camada intermediária empregam camadas internamente para estruturar o *design*.

10

3.2- Visão Geral da Arquitetura

A visão geral da arquitetura ICDE v2.0 está representado na figura abaixo. Clientes ICDE usam o

componente *ICDE cliente API* para fazer chamadas para *a* API do *ICDE*. Este é hospedado por um servidor de aplicações JEE e traduz chamadas de API através de JDBC para o armazenamento de dados.



ICDE arquitetura API

A notificação de eventos é atingida usando uma infraestrutura de publicação-assinatura baseada no Java Messaging Service (JMS).

Usando JEE como uma infraestrutura de aplicativos, o ICDE pode ser implantado de modo que um armazenamento de dados possa suportar:

- Vários usuários interagindo com os componentes de coleta de dados.
- Várias ferramentas de terceiros que interagem com os componentes da API.

11

3.3- Visualizações estruturais

Um diagrama de componentes para a criação de API é mostrado na figura abaixo.

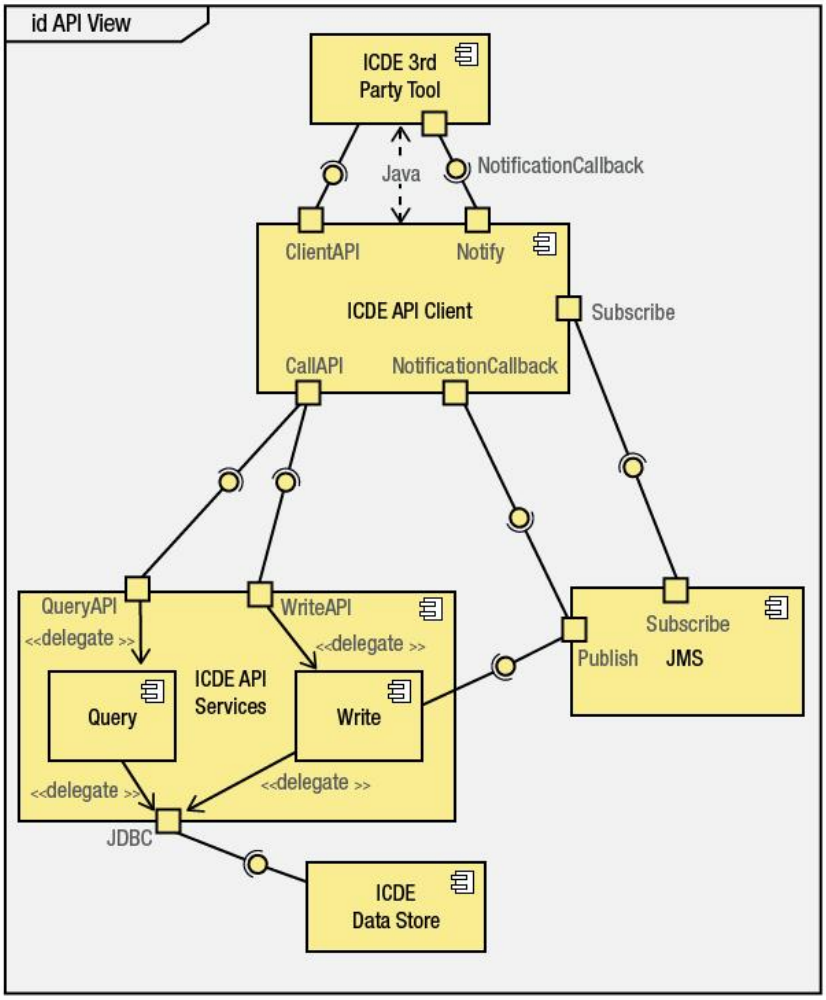


Diagrama de componentes para a arquitetura API ICDE

O diagrama mostra as interfaces e dependências de cada componente, a saber:

- **ICDE ferramenta de terceiros;**
- **ICDE cliente API;**
- **Serviços API ICDE;**
- **ICDE Data Store;**
- **JMS.**

ICDE ferramenta de terceiros

Esta ferramenta usa a interface do componente da API ICDE. A interface da API suporta os serviços necessários para consulta, gravação e alteração de dados no banco de dados

ICDE cliente API

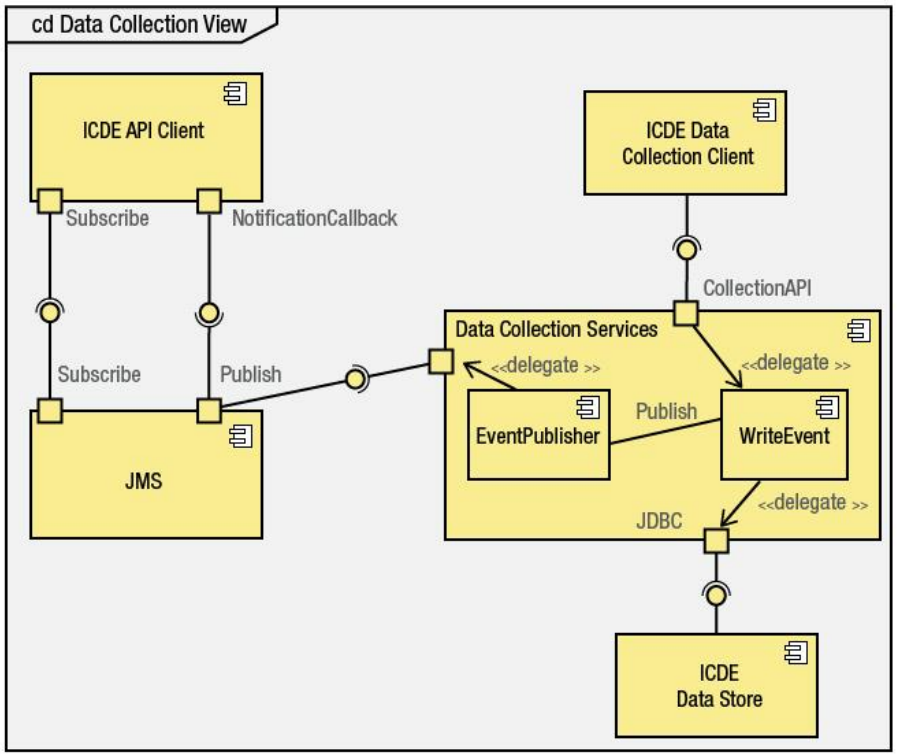
Este implementa a parte do cliente da API. Esta API recebe a requisição das ferramentas de terceiros, traduz estas chamadas para EJB e chama os componentes da API para ler ou escrever dados do banco de dados.

Serviços API ICDE O componente de serviços API compreende EJBs de sessão sem estado para acessar a loja ICDE de dados usando JDBC.
ICDE Data Store Este é o banco de dados do ICDE v2.0.
JMS Este é um padrão JEE Java Messaging Service, e suporta uma variedade de tópicos utilizados para a notificação de eventos utilizando o JMS publicar-subscrever interfaces.

12

As responsabilidades dos componentes são:

- **Coleta de Dados ICDE Cliente;**
- **A coleta de dados serviços;**
- **EventPublisher.**



Componentes de coleta de dados

Coleta de Dados ICDE Cliente Isso é parte do ambiente de aplicativo cliente ICDE. Ele recebe dados de eventos do aplicativo cliente, e chama o método necessário no CollectionAPI para armazenar esse evento. Ele encapsula todo o conhecimento sobre a interação entre o servidor JEE com o aplicativo cliente ICDE.
A coleta de dados serviços Este compreende EJBs de sessão sem estado que gravam os dados do evento passados para eles como parâmetros para o ICDE Armazenamento de Dados. Alguns tipos de eventos também causam uma notificação de eventos a serem passados para o EventPublisher.
EventPublisher Este publica dados de evento sobre os JMS utilizando um conjunto de temas pré-configurados para eventos que devem ser publicados (nem todos os eventos gerados pelos usuários são publicados, como por exemplo, o mover do mouse). Estes eventos são entregues para os componentes da API ICDE).

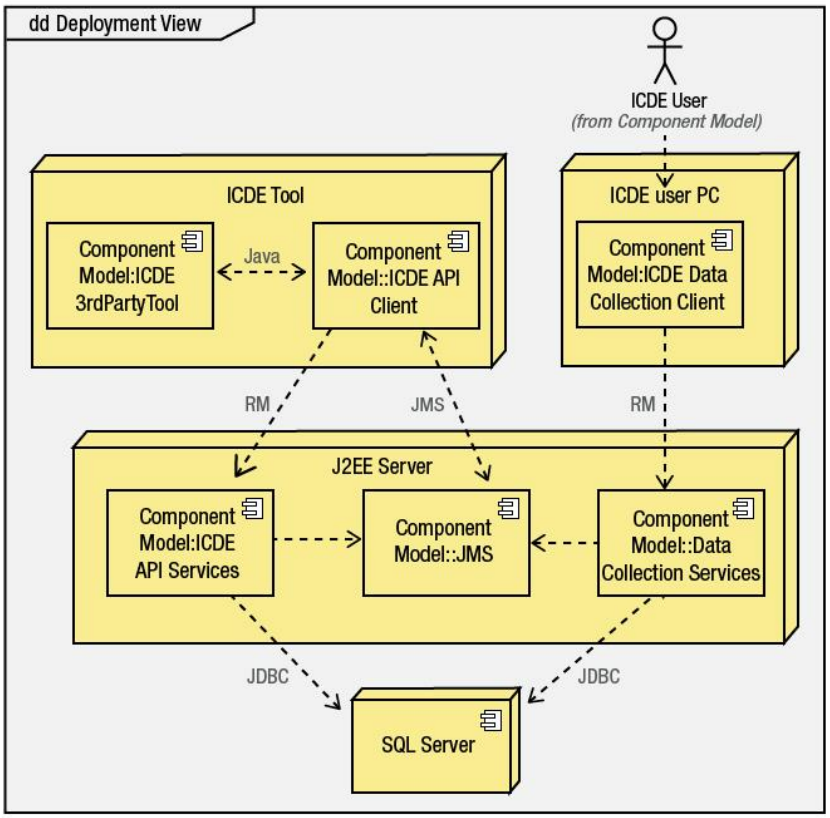
13

Um diagrama de implantação para a arquitetura ICDE é mostrado na figura abaixo. Ele mostra como os vários componentes são atribuídos aos nós. Apenas um único usuário e uma única ferramenta de terceiros são mostrados, mas o servidor JEE pode suportar vários clientes de qualquer tipo.

Algumas questões a serem observadas:

- Embora as ferramentas de terceiros estejam mostradas num nó de execução diferente para a estação de trabalho ICDE utilizador, este não é necessariamente o caso. Ferramentas ou componentes específicos de ferramentas podem ser implantados na estação de trabalho do usuário. Esta é uma decisão de configuração que depende da ferramenta.

- Há um componente *de cliente API ICDE* para cada ferramenta de terceiros. Este componente é construído como um arquivo JAR que está incluído na ferramenta construir.

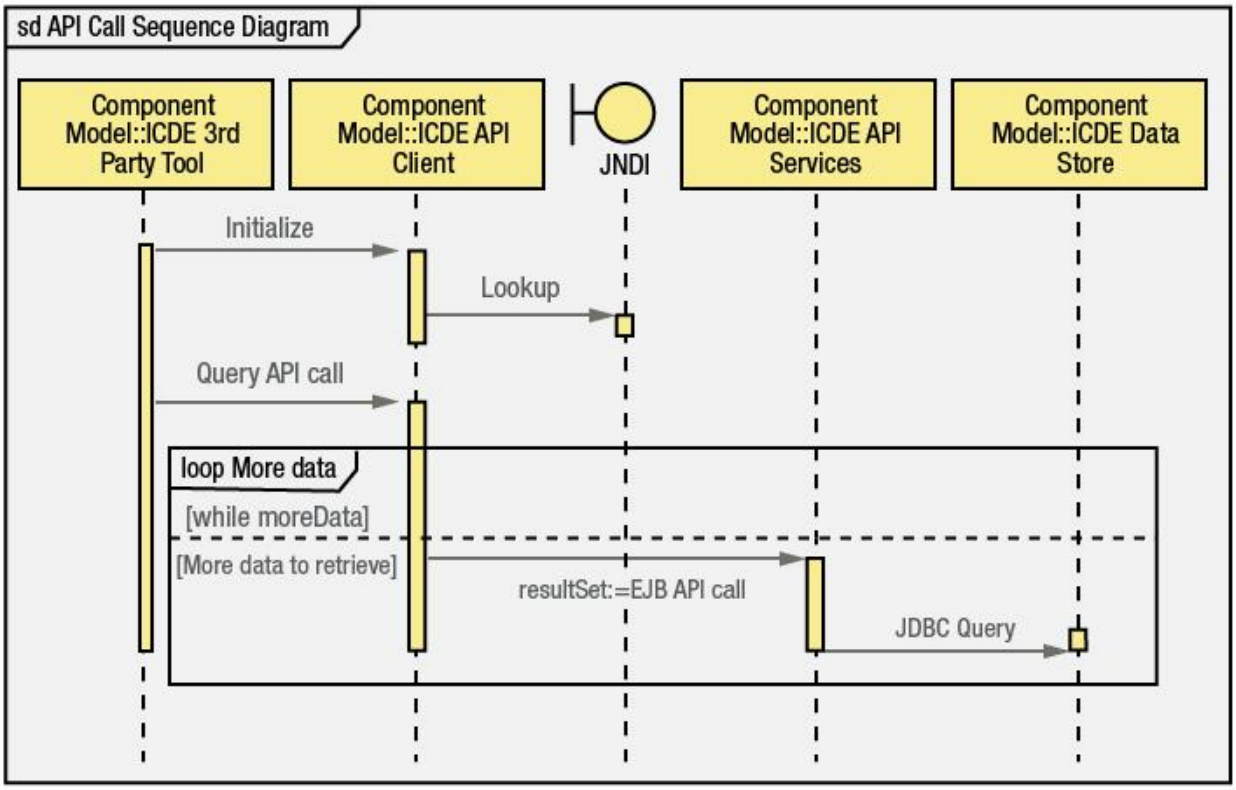


ICDE diagrama de implantação

14

3.4- Visões Comportamentais

Um diagrama de sequência para uma chamada API é mostrado na figura a seguir. A API fornece uma chamada explícita "Inicializar", que deve invocar ferramentas. Este faz com que o *cliente API ICDE* estabeleça referências para os beans de sessão sem estado EJB usando o serviço de diretório JEE (JNDI).



Consulta API diagrama de sequência chamada

Uma vez que a camada de API é inicializada, a ferramenta de terceiros chama uma das APIs de consulta disponíveis para recuperar dados do evento. Essa solicitação é passada para uma instância EJB que implementa a consulta e emite a chamada JDBC para obter os eventos que satisfazem a consulta.

Todas as APIs ICDE que retornam coleções de eventos potencialmente podem obter grandes conjuntos de resultados do banco de dados. Isso cria o potencial para esgotamento de recursos no servidor JEE, especialmente se várias consultas retornam coleções de eventos grandes simultaneamente.

15

Um diagrama de sequência que descreve o comportamento de uma chamada de escrita API é mostrado na figura abaixo. A chamada API de gravação contém os valores dos parâmetros que permitem à ICDE cliente API especificar se um evento deve ser publicado depois de uma gravação bem-sucedida e, em caso afirmativo, em que tema o evento deve ser publicado.

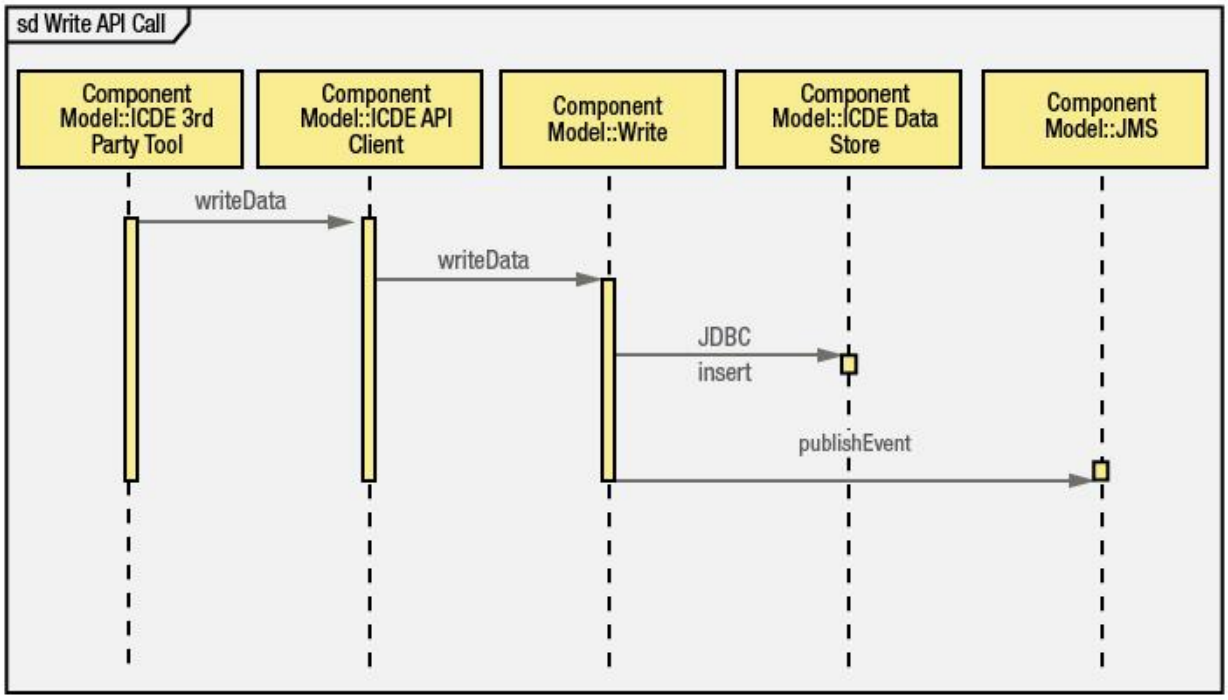


Diagrama de sequência para a API de gravação

16

Um diagrama de sequência para o armazenamento de um evento gerado pelo utilizador ICDE é mostrado na figura abaixo. Um tipo de evento pode exigir múltiplas inserções JDBC e instruções a serem executadas para armazenar os dados de eventos; portanto, deve-se utilizar os serviços de transação de contêiner.

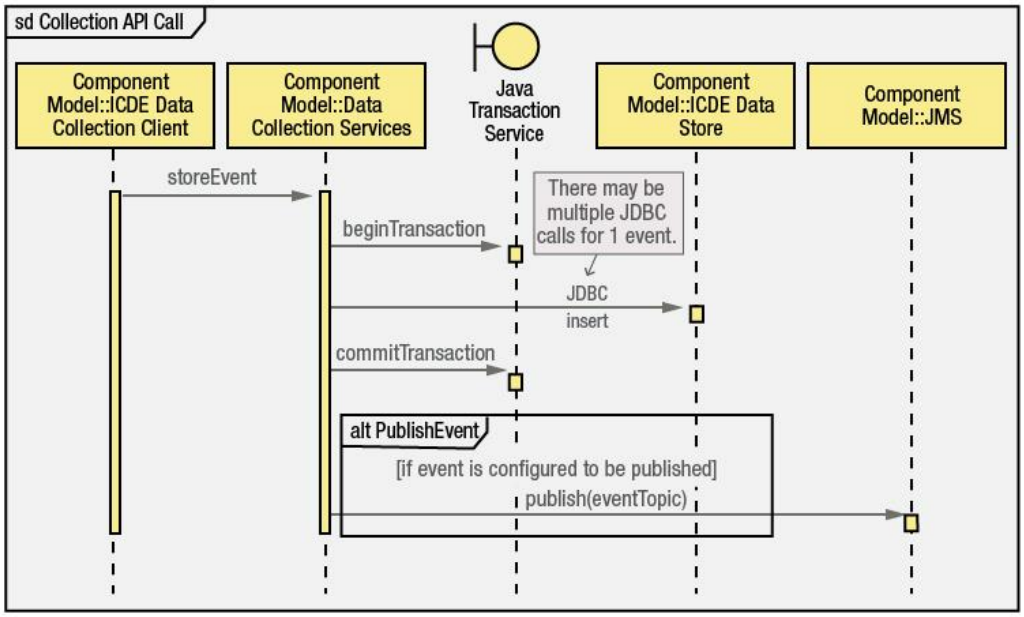


Diagrama de sequência para o armazenamento de eventos gerados pelo usuário

17

3.5- Questões de Implementação

A plataforma Java 2 Enterprise Edition foi selecionada para implementar o sistema ICDE v2.0. Java é uma plataforma neutra, satisfazendo a requisito para a heterogeneidade da plataforma. Há também versões de código-aberto disponíveis para implantação de baixo custo, bem como alternativas comerciais de alto desempenho que podem ser mais indicadas para clientes de sites de missão crítica. Além disso, dispõe de suporte JEE para sistemas baseados em componentes distribuídos e para acesso ao banco de dados.

Questões adicionais de implementação a considerar são:

Segurança: ferramentas ICDE são autenticadas por meio de um nome de usuário e senha. A API suporta uma função *de login,* que valida a combinação usuário/senha contra as credenciais no armazenamento de dados ICDE, e permite o acesso a um conjunto específico de dados do usuário ICDE. Este é o mesmo mecanismo utilizado na v1.0.

O valor de "tamanho de página" podem ser ajustados para cada tipo de evento para tentar maximizar o desempenho do servidor e rede. Um valor típico é de 1.000.

18

4- Análise de Arquitetura

As seções a seguir fornecem uma análise da arquitetura ICDE em termos de cenários e riscos.

4.1 Cenário

Os seguintes cenários são considerados:

- **Modificar organização do Data Store.**
- **Mover a arquitetura ICDE para outro fornecedor JEE.**
- **Escalar uma implantação de 150 usuários.**

4.2 Riscos

Os seguintes riscos devem ser observados no projeto ICDE.

O planejamento de capacidade envolve descobrir o quanto de *hardware* e *software* é necessário para suportar uma instalação específica ICDE, com base no número de usuários simultâneos, velocidades de rede e *hardware* disponíveis.

Modificar organização do Data Store Alterações na entidade de banco de dados irão exigir alterações de código no componente EJB. As mudanças estruturais que não adicionam novos atributos de dados estão contidas totalmente dentro desses componentes e não se propagam à API ICDE.
Mover a arquitectura ICDE para outro fornecedor JEE Enquanto a aplicação ICDE está codificada com os padrões JEE e não usa quaisquer classes de extensão de fornecedores, a experiência da indústria de aplicações mostra que JEE são portáteis de um servidor de aplicativos para outro com pequena quantidade de esforço.
Escalar uma implantação de 150 usuários Isso exige planejamento de capacidade com atenção especial para a especificação do *hardware* disponível e capacidade de redes. A camada do servidor JEE pode ser replicada e agrupada facilmente devido ao uso de beans de sessão sem estado.

18

5- RESUMO

Esse módulo descreveu a importância para uma empresa de definir um padrão de documentação de arquitetura. Para exemplificar um documento de arquitetura, algumas das decisões de *design* tomadas para uma aplicação exemplo ICDE foram apresentadas. O objetivo foi o de transmitir o pensamento de análise que é necessário para projetar uma arquitetura como tal, e demonstrar o nível da documentação de projeto que deve ser suficiente nos muitos projetos.

Note-se que alguns dos detalhes mais delicados do projeto não foram tratados. Mas o exemplo ICDE representa um pedido de média complexidade e, portanto, proporciona um excelente exemplo da obra de um arquiteto de *software*.