Apontamentos de ESW

Índice

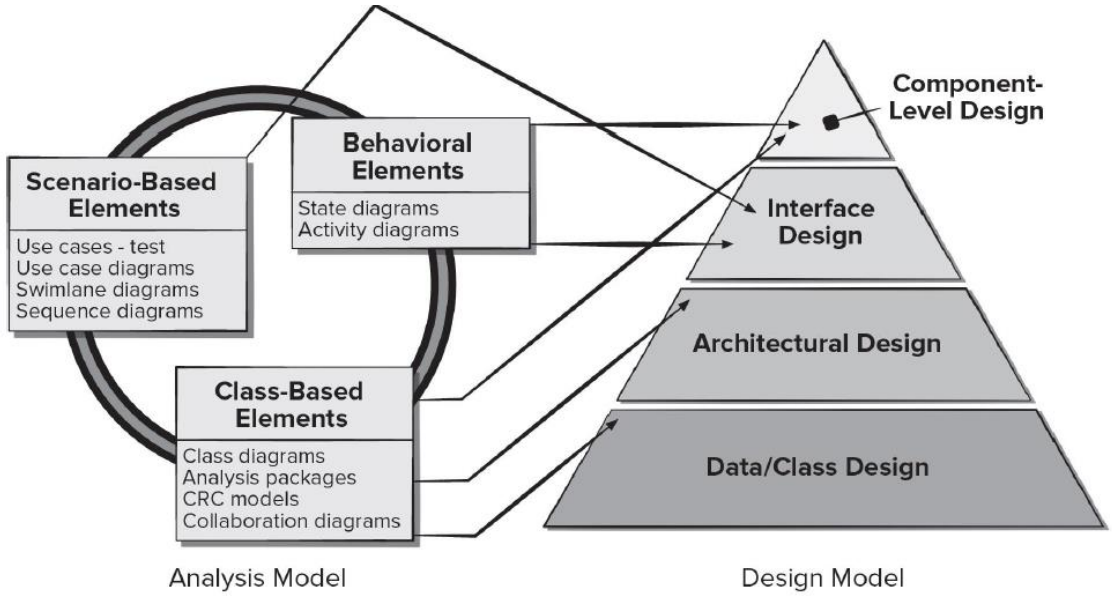
Design de Software

- Abrange o conjunto de princípios, conceitos e práticas que levam ao desenvolvimento de um sistema ou produto de alta qualidade;
- Os princípios do design estabelecem e prevalecem sobre uma filosofia que orienta o designer à medida que o trabalho é realizado;
- Os conceitos do design devem ser entendidos antes dos mecanismos da prática do design sem aplicados;
- As práticas de conceção de software mudam continuamente à medida que novos métodos, melhores análises, e uma compreensão mais ampla evoluem.

Design de Engenharia de Software

- **Data/Class design** – transforma as classes de análise em classes de implementação e estruturas de dados.
- **Architectural design** – define uma relação entre os principais elementos estruturais do software.
- **Interface design** – define como os elementos de software, elementos de hardware e os utilizadores finais comunicam.
- **Component-level design** – transforma elementos estruturais em descrições processuais de componentes de software.

Modelo de requisitos de mapeamentos ao modelo de desenho



Design e qualidade

- O design deve implementar todos os requisitos explícitos contidos no modelo de análise, e deve acomodar todos os requisitos implícitos desejas pelo cliente.

- O design deve ser um guia legível e compreensível para aqueles que geram código e para aqueles que testam subsequentemente apoiam o software.
- O design deve fornecer uma imagem completado do software, abordando os dados, domínios funcionais, e comportamentais numa perspetiva de implementação.

Diretrizes de qualidade

1. Um design deve exibir uma arquitetura (a) criada utilizando estilos ou padrões arquitetónicos reconhecíveis, (b) composta por componentes bem concebidos (c) implementados numa arquitetura evolutiva.
2. Um design deve ser modular.
3. Um design deve conter representações distintas de dados, arquitetura, interfaces e componentes.
4. Um design deve conduzir a estruturas de dados que são extraídas de padrões de dados reconhecíveis.
5. Um design deve conter componentes funcionalmente independentes.
6. Um design deve conduzir a interfaces que reduzam a complexidade das ligações entre componentes e o ambiente externo.
7. Uma conceção deve ser derivada utilizando um método repetível que seja impulsionado pela análise dos requisitos de software.
8. Um desenho deve ser representado usando uma notação significativa.

Características comuns do design

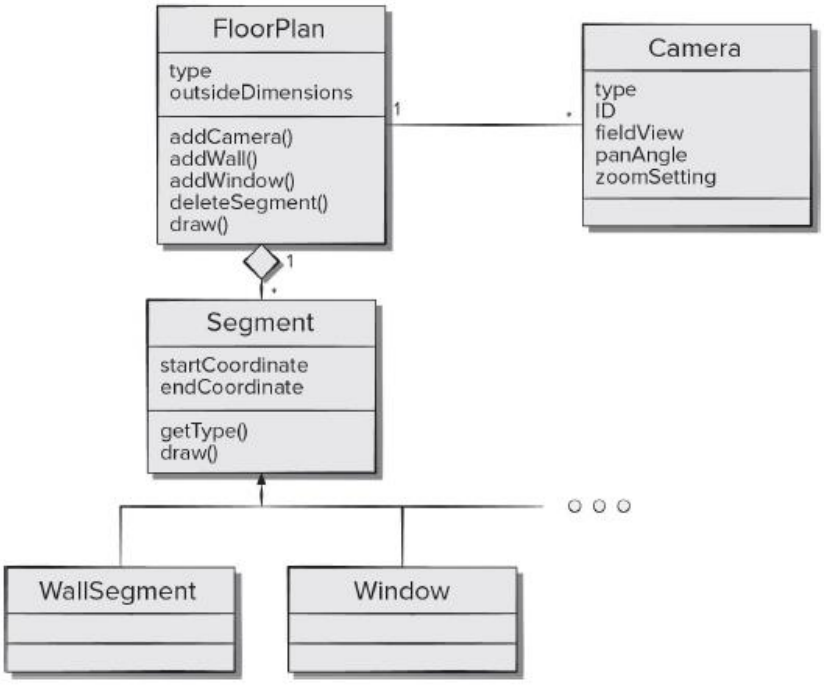
Cada nova metodologia de conceção de software introduz heurísticas e noções únicas, mas cada uma delas contém:

1. Um mecanismo para a tradução do modelo de requisitos para uma representação do design.
2. Uma notação para representar os componentes funcionais e as interfaces.
3. Heurística para refinamento e partição.
4. Diretrizes para a avaliação da qualidade.

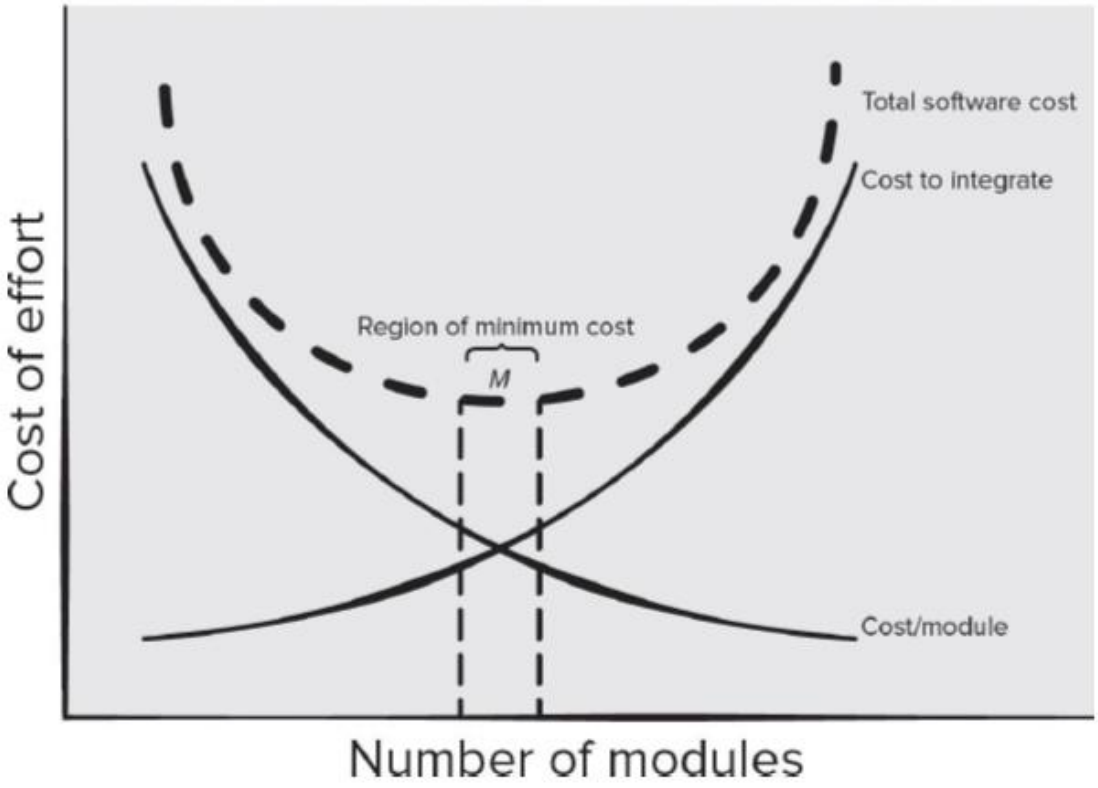
Conceitos do design

- **Abstração** – dados (recolha nomeada de dados que descrevem o objeto de dados), procedimentos (sequência de nomes de instruções com função específica e limitada).
- **Arquitetura** – estrutura global ou organização dos componentes de software, formas de interação dos componentes, e estrutura dos dados utilizados pelos componentes.
- **Padrões de desenho** – descrevem uma estrutura de desenho que resolve um problema de desenho bem definidos dentro de um contexto específico.
- **Separação de preocupações** – qualquer problema complexo pode ser mais facilmente tratado se for subdividido em pedaços.
- **Modularidade** – compartimentação de dados e função.
- **Esconder informação** – interfaces controladas que definem e fazem cumprir o acesso aos detalhes processuais da componente e a qualquer estrutura de dados local utilizada pela componente.
- **Independência funcional** – componentes de mente única (alta coesão) com aversão à interação excessiva com outros componentes (baixo acoplamento).
- **Refinamento por etapas** – elaboração incremental de detalhes para todas as abstrações.
- **Refactoring** – uma técnica de reorganização que simplifica o desenho sem alterar a funcionalidade.
- **Classes de desenho** – fornece detalhes de desenho que permitirão a implementação de classes de análise.

Exemplo de uma classe de desenho



Modularidade e custo de software



Características da classe de desenho

- **Completo** – inclui todos os atributos e métodos necessários e suficiente (contém apenas os métodos necessários para alcançar a intenção de classe)
- **Primitividade** – cada método de classe centra-se na prestação de um serviço.
- **Elevada coesão** – classes pequenas, focadas, de mente única.
- **Baixo acoplamento** – colaboração de classe mantida ao mínimo.

Esconder informação

- Reduz a probabilidade de “efeitos secundários”.
- Limita o impacto global das decisões do design locais.
- Enfatiza a comunicação através de interfaces controladas.
- Desencoraja a utilização de dados globais.
- Conduz ao encapsulamento de um atributo do design de alta qualidade.
- O resultado é um software de maior qualidade.

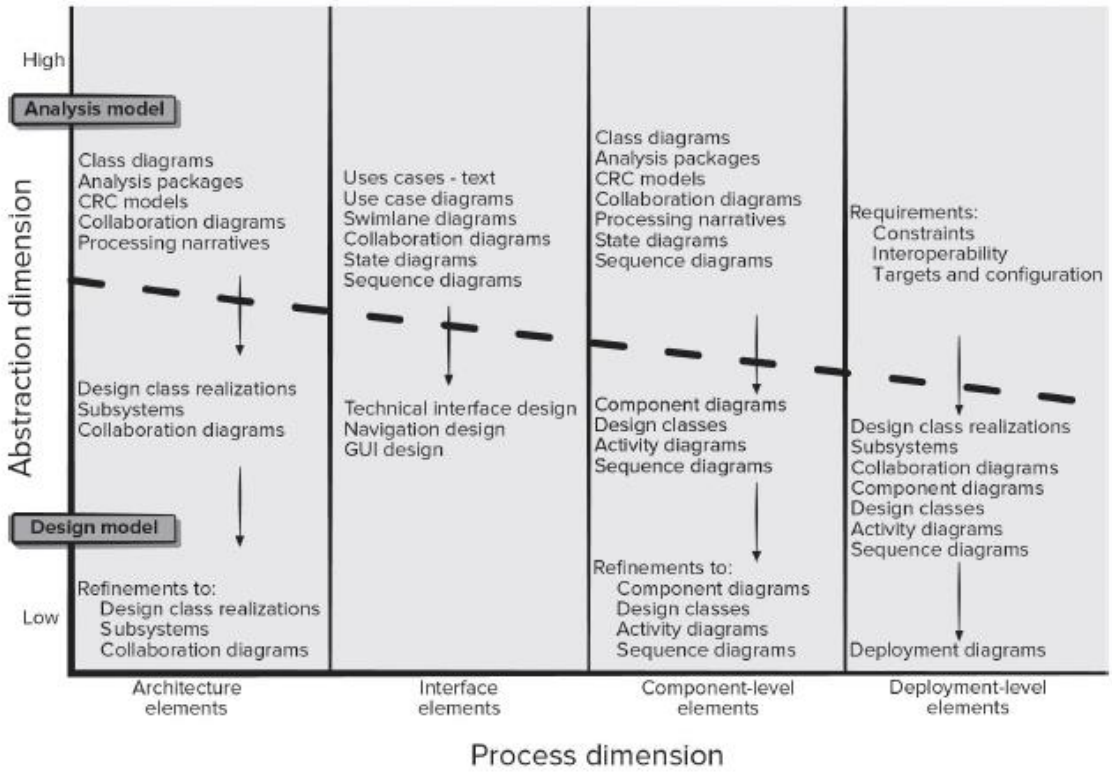
Propriedades arquitetónicas

- **Propriedades estruturais**. Este aspeto da representação do projeto arquitetónico define os componentes de um sistema (por exemplo, módulos, objetos, filtros) e a forma como os componentes são embalados e interagem uns com os outros.
- **Propriedades funcionais extra**. A descrição do projeto arquitetónico deve abordar a forma como a arquitetura de projeto atinge os requisitos de desempenho, capacidade, fiabilidade, segurança, adaptabilidade e outras características.
- **Famílias de sistemas relacionados.** A conceção arquitetónica deve basear-se em padrões repetíveis (blocos de construção) frequentemente encontrados na conceção de sistemas semelhantes.

Modelo do padrão do design

- **Nome do padrão** – descreve a essência do padrão num nome curto, mas expressivo.
- **Intenção** – descreve o padrão e o que ele faz.
- **Também conhecido como** – lista quaisquer sinónimos para o padrão.
- **Motivação** – fornece um exemplo do problema.
- **Aplicabilidade** – anota situações específicas de desenho em que o padrão é aplicável.
- **Estrutura** – descreve as classes que são necessárias para implementar o padrão.
- **Participantes** – descreve as responsabilidades das turmas que são necessárias para implementar o padrão.
- **Colaborações** – descreve como os participantes colaboram para cumprir as suas responsabilidades.
- **Consequências** – descreve as “forças de conceção” que afetam o padrão e as potenciais trocas comerciais que devem ser consideradas quando o padrão é implementado.
- **Padrões relacionados** – padrões do design relacionados com referências cruzadas.

Modelo do design



Princípios da modelação do design

1. O design deve ser rastreável até ao modelo de requisitos
2. Considerar sempre a arquitetura do sistema a ser construído.
3. A conceção dos dados é tão importante como a conceção das funções de processamento.
4. As interfaces (tanto internas como externas) devem ser concebidas com cuidados.
5. A conceção da interface do utilizador deve ser sintonizada com as necessidades do utilizador final e salientar a facilidade de utilização.
6. A conceção do nível de componentes de ser funcionalmente independente.
7. Os componentes devem ser acoplados uns aos outros de forma solta, para além do ambiente.
8. As representações do design (modelos) devem ser facilmente compreensíveis.
9. O desenho deve ser desenvolvido iterativamente.
10. A criação de um modelo de desenho não impede a utilização de uma abordagem ágil.

Elementos de desenho de dados

Modelo de dados – objetos de dados e arquiteturas da base de dados.

- Examina objetos de dados independentemente do processamento.
- Concentra a atenção no domínio dos dados.
- Cria um modelo ao nível de abstração do cliente.
- Indica a forma como os objetos de dados se relacionam uns com os outros.

Os objetos de dados podem ser uma entidade externa, uma coisa, um evento, um lugar, um papel, uma unidade organizacional, ou uma estrutura.

Os objetos de dados contêm um conjunto de atributos que atuam como qualidade, características, ou descritor do objeto.

Os objetos de dados podem estar ligados uns aos outros de muitas maneiras diferentes.

Elementos do design arquitetónico

Desenho arquitetónico para software - equivalente à planta de uma casa.

O modelo arquitetónico é derivado de três fontes:

- Informação sobre o domínio de aplicação para o software a ser construído.
- Elementos do modelo de requisitos específicos, tais como classes de análise de fluxo de dados e as suas relações (colaborações) para o problema em questão.
- Disponibilidade de padrões e estilos arquitetónicos.

Elementos de desenho de interface

A interface é um conjunto de operações que descreve o comportamento observável externamente de uma classe e dá acesso às suas operações públicas.

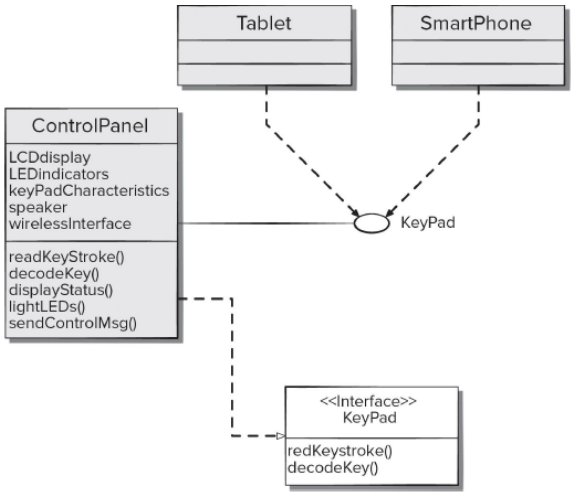
Elementos importantes:

- Interface do utilizador (UI).
- Interfaces externas entre vários componentes do design.
- Interfaces internas entre vários componentes do design.

A UI ou Experiência do Utilizador (UX) é uma ação de engenharia importante para assegurar a criação em produtos de software utilizáveis.

As interfaces internas e externas devem incorporar tanto a verificação de erros como características de segurança adequadas.

Modelo de interface para o Painel de Controlo



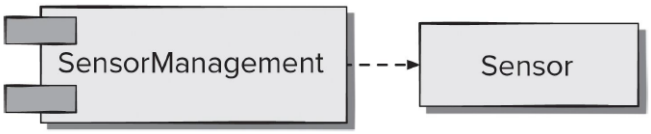
Elementos de desenho ao nível dos componentes

Descreve os detalhes internos de cada componente do software.

Define:

- Estruturas de dados para todos os objetos de dados locais.
- Detalhe algorítmico para todas as funções de processamento de componentes.
- Interface que permite o acesso a todas as operações componentes.

Modelado através de diagramas de componentes UML.



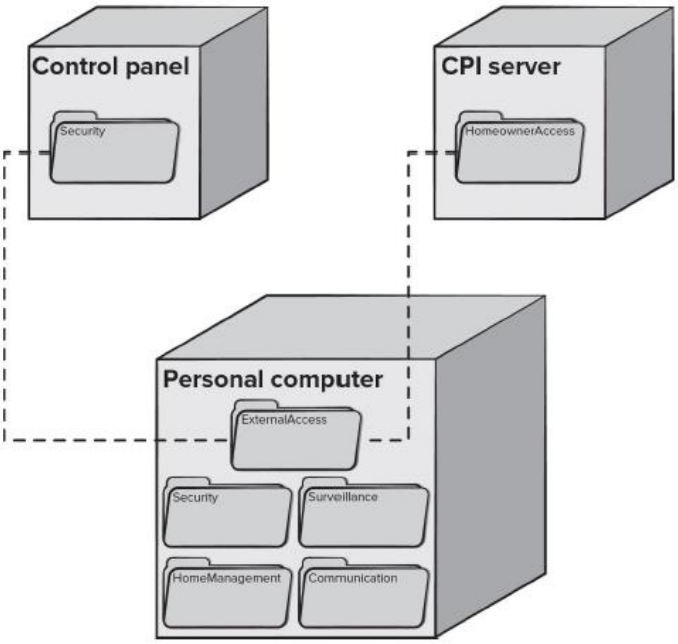
Elementos de desenho de implantação

Indica como as funcionalidades e subsistemas de software serão atribuídos dentro do ambiente de computação física.

Modelado através de diagramas de implantação UML.

- Os diagramas de implantação dos formulários descritivos mostram o ambiente informático, mas não indicam detalhes de configuração.
- Os diagramas de implantação de instâncias identificam as configurações de hardware específicas e são desenvolvidos nas últimas fases de conceção.

Diagrama da Instância de Implementação da UML



O que é a Arquitetura de Software?

A arquitetura não é o software operacional, é uma representação que permite a um engenheiro de software fazer:

1. Analisar a eficácia do desenho para satisfazer os seus requisitos declarados
2. Considerar alternativas arquitetónicas numa fase e que fazer alterações do design ainda é relativamente fácil
3. Reduzir os riscos associados com a construção do software.

Porque é que a Arquitetura de Software é importante?

- A arquitetura de software fornece uma representação que facilita a comunicação entre os intervenientes interessados no desenvolvimento de um sistema baseado em computador.
- A arquitetura destaca as primeiras decisões do design que terão um impacto profundo em todo o trabalho de engenharia de software que se segue.
- A arquitetura constitui um modo relativamente pequeno e intelectualmente compreensível de como o sistema é estrutura e de como os seus componentes funcionam em conjunto.

Descrições Arquitetónicas

Descreve a utilização de pontos de vista arquitetónicos, quadros arquitetónicos e linguagens de descrição arquitetónica como um meio de codificação das convenções e práticas comuns de descrição arquitetónica.

A norma IEEE define uma descrição arquitetónica (AD) como "uma coleção de produtos para documentar uma arquitetura".

- Uma descrição da arquitetura deve identificar os intervenientes no sistema que tenham preocupações consideradas fundamentais para a arquitetura do sistema de interesse.
- Estas preocupações serão consideradas quando aplicáveis e identificadas na descrição da arquitetura: finalidade do sistema, adequação da arquitetura, viabilidade da construção e implantação do sistema, riscos e impactos do sistema, e a capacidade de manutenção e evolução do sistema.

Documentação da Decisão da Arquitetura

1. Determinar os elementos de informação necessário para cada decisão.
2. Definir ligações entre cada decisão e os requisitos apropriados.
3. Fornecer mecanismos para alterar o estatuto quando decisões alternativas precisam de ser avaliadas,
4. Definir relações de pré-requisitos entre as decisões para apoiar a rastreabilidade.
5. Associar decisões significativas a pontos de vista arquitetónicos resultantes de decisões.
6. Documentar e comunicar todas as decisões à medida que são tomadas.

Agilidade e Arquitetura

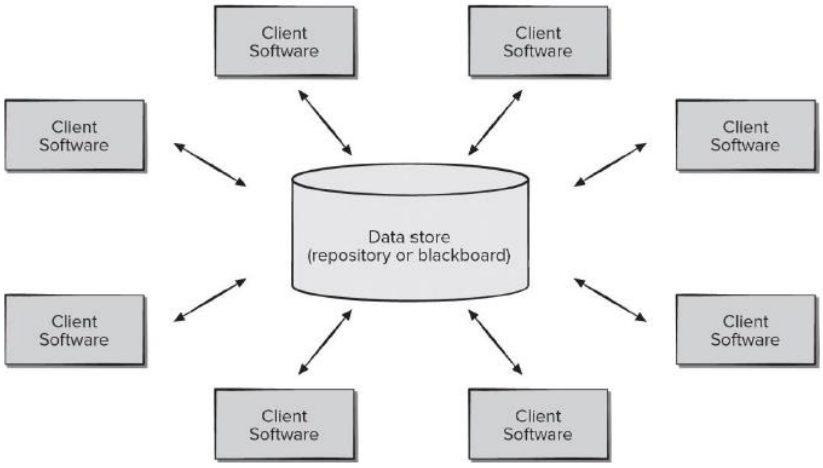
- Para evitar retrabalho, as histórias de utilizadores são utilizadas para criar e desenvolver um modelo arquitetónico (esqueleto de caminhada) antes de iniciar qualquer codificação.
- Utilizar modelos que permitam aos arquitetos de software adicionar histórias de utilizadores à storyboard em evolução e trabalha com o proprietário do produto para priorizar as histórias arquitetónicas como "sprints" (unidades de trabalho) são planeados.
- Os projetos bem geridos e ágeis incluem a entrega de documentação arquitetónica durante cada sprint.
- Após a conclusão do sprint, o arquiteto revê o protótipo de trabalho em termos de qualidade antes de a equipa o apresentar às partes interessadas numa revisão formal do sprint.

Estilos arquitetónicos

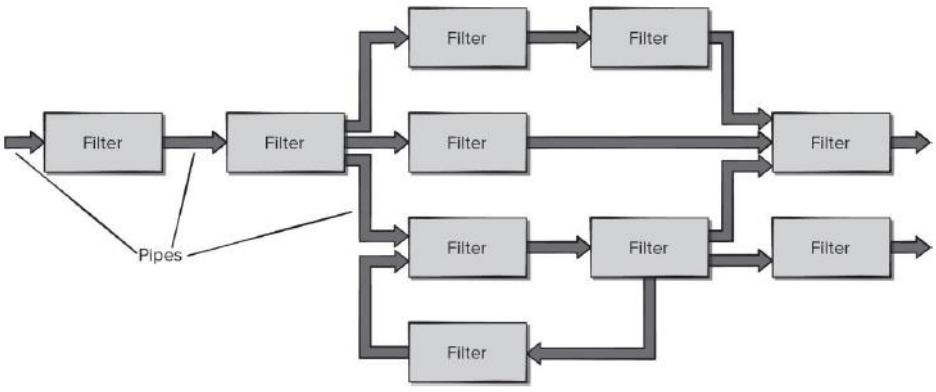
Cada estilo descreve uma categoria do sistema que engloba:

1. Um conjunto de componentes (por exemplo: uma base de dados, módulos computacionais) que desempenha uma função requerida por um sistema.
2. Um conjunto de conetores que permitem "comunicação, coordenação e cooperação" entre componentes.
3. Restrições que definem como os componentes podem ser integrados para formar o sistema.
4. Modelos semânticos que permitem a um designer compreender as propriedades globais de um sistema através da análise das propriedades conhecidas das suas partes constituintes.

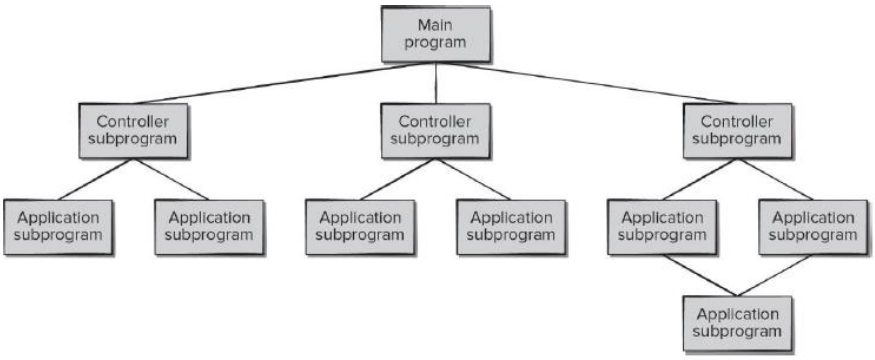
Arquitetura centrada em dados



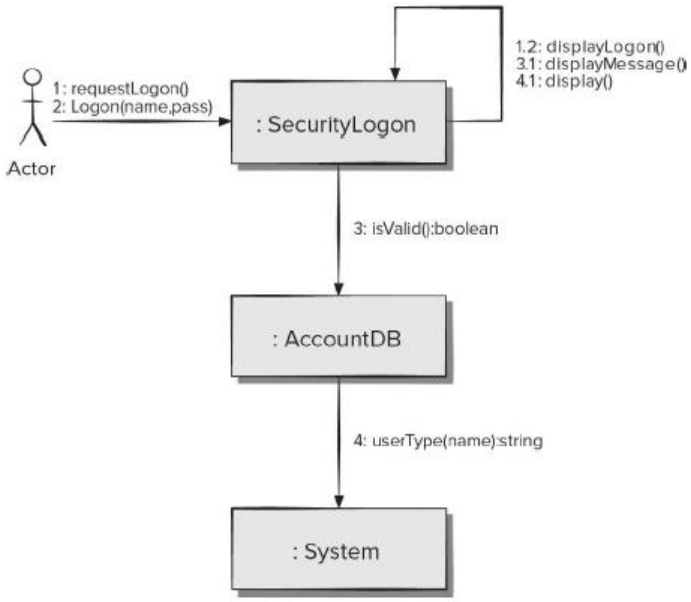
Arquitetura do fluxo de dados



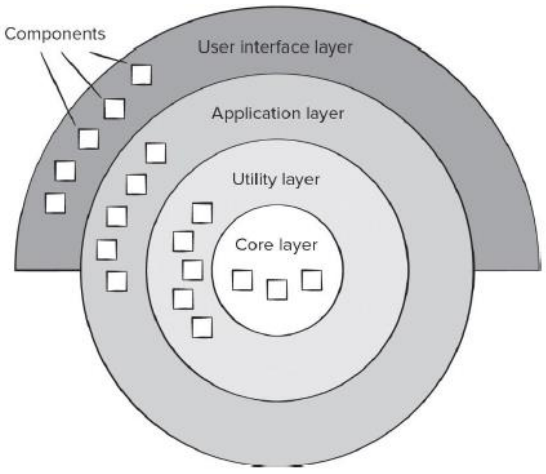
Arquitetura de Retorno de Chamada



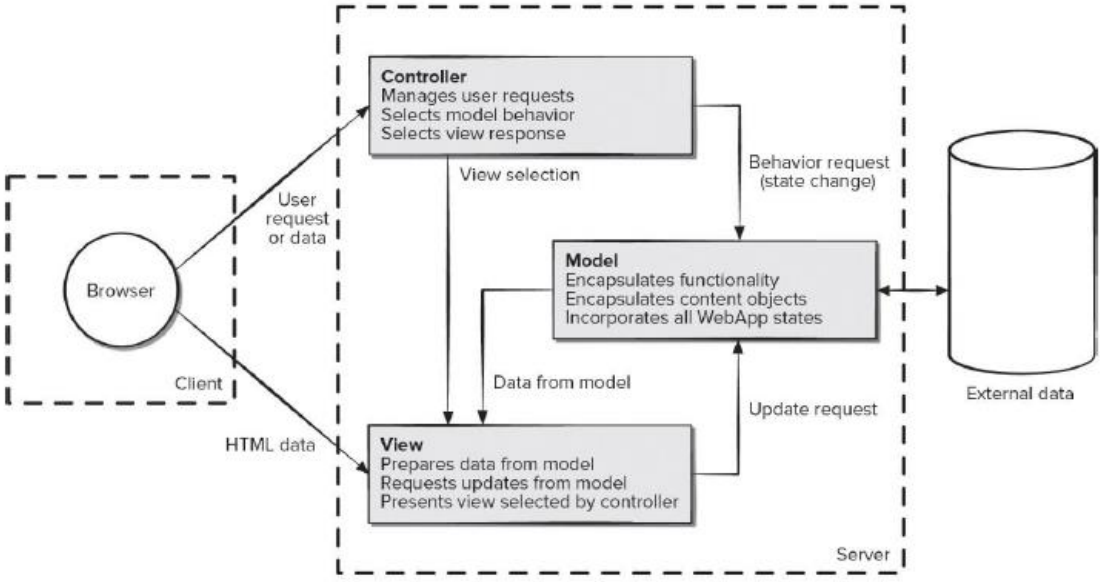
Arquitetura Orientada a Objetos



Arquitetura em camadas



Arquitetura Model-View-Controller



Organização e Aperfeiçoamento Arquitetónico

Controlo

- Como é gerido o controlo dentro da arquitetura?
- Existe uma hierarquia de controlo distinta e, em caso afirmativo, qual é o papel dos componentes dentro desta hierarquia de controlo?
- Como é que os componentes transferem o controlo dentro do sistema?
- Como é o controlo partilhado entre os componentes?
- Qual é a topologia de controlo (ou seja, a forma geométrica que o controlo assume)?
- O controlo é sincronizado, ou os componentes funcionam de forma assíncrona?

Dados

- Como são comunicados os dados entre componentes?
- O fluxo de dados é contínuo, ou os objetos de dados são esporadicamente passados para o sistema?
- Qual é o modo de transferência de dados?
- Existem componentes de dados, e em caso afirmativo, qual é o seu papel?
- Como é que os componentes funcionais interagem com os componentes de dados?
- Os componentes dos dados são passivos ou ativos?
- Como é que os dados e o controlo interagem no sistema?

Considerações arquitetónicas

- **Economia** – o software é desordenado e depende da abstração para reduzir detalhes desnecessários.
- **Visibilidade** – as decisões arquitetónicas e as suas justificações devem ser óbvias para os engenheiros de software que as revejam.
- **Espaço** – separação de preocupações num desenho sem introduzir dependências ocultas.
- **Simetria** – a simetria arquitetónica implica que um sistema seja consistente e equilibrado nos seus atributos.
- **Emergência** – comportamento e controlo emergentes e auto-organizados são a chave para criar arquiteturas de software escaláveis, eficientes e económicas.

Design arquitetónico

O software deve ser colocado em contexto.

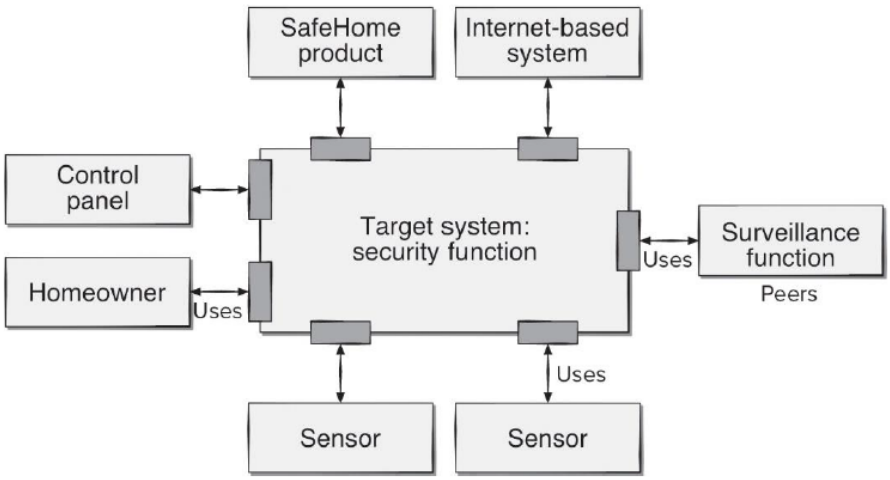
- A conceção deve definir as entidades externas (outros sistemas, dispositivos, pessoas) com as quais o software interage com a natureza da interação.

Deve ser identificado um conjunto de arquétipos arquitetónicos.

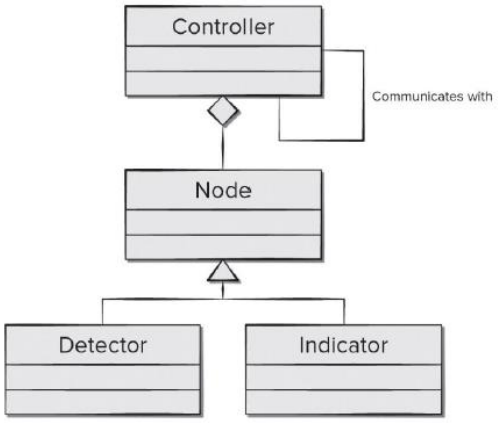
- Um arquétipo é uma abstração (semelhante a uma classe) que representa um elemento do comportamento do sistema.

O designer especifica a estrutura do sistema definindo e refinando os componentes de software que implementam cada arquétipo.

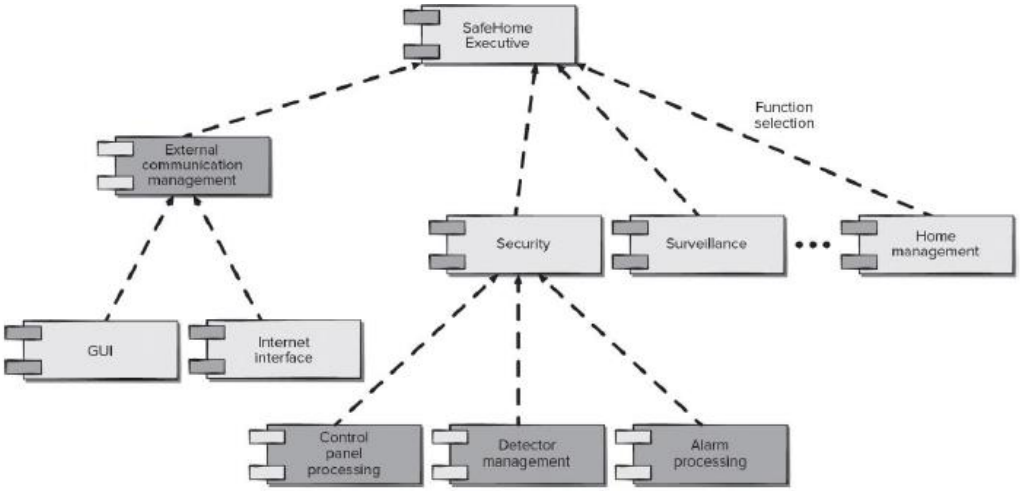
Diagrama de Contexto de Arquitetura



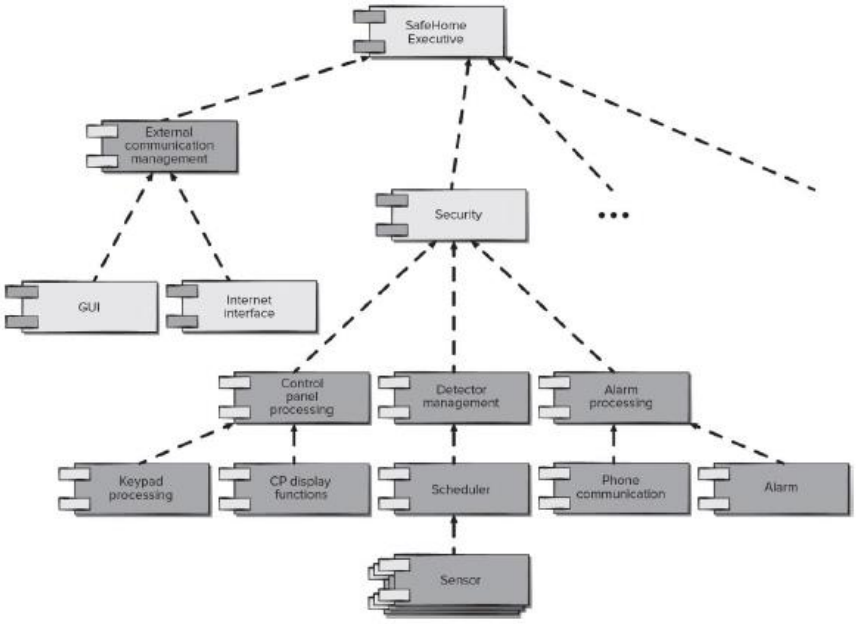
Arquétipo de função de segurança SafeHome



Arquitetura de componentes de alto nível SafeHome



Arquitetura de Componentes Refinados SafeHome



Análise arquitetónica Tradeoff

1. Recolher cenários
2. Exigências, restrições, descrição do ambiente
3. Descrever os estilos/padrões arquitetónicos que foram escolhidos para abordar os cenários e requisitos utilizando uma destas views: módulo, processo, fluxo de dados
4. Avaliar os atributos de qualidade, considerando cada um deles isoladamente
5. Identificar a sensibilidade dos atributos de qualidade a vários atributos arquitetónicos para um estilo arquitetónico específico
6. Arquiteturas candidatas à crítica (desenvolvidas no passo 3) utilizando a análise de sensibilidade (realizada no passo 5)

Revisões arquitetónicas

- Avaliar a capacidade da arquitetura de software para satisfazer os requisitos de qualidade dos sistemas e identificar potenciais riscos.
- Ter o potencial de reduzir os custos do projeto através da deteção precoce de problemas de conceção.
- Fazem frequentemente uso de revisões baseadas na experiência, avaliação de protótipos, revisões de cenários e listas de verificação.

Revisões arquitetónicas baseadas em padrões

1. Identificar e discutir os atributos de qualidade caminhando por os casos de utilização.
2. Discutir um diagrama de arquitetura do sistema em relação aos requisitos.
3. Identificar os padrões de arquitetura utilizados e adequar a estrutura do sistema à estrutura dos padrões.
4. Utilizar a documentação existente e utilizar casos para determinar o efeito de cada padrão nos atributos de qualidade.
5. Identificar todas as questões de qualidade levantadas pelos padrões de arquitetura utilizados no desenho.
6. Desenvolver um breve resumo das questões descobertas durante a reunião e fazer revisões do esqueleto de caminhada.

Abordagem estratégica aos testes

- Deve realizar revisões técnicas eficazes, o que pode eliminar muitos erros antes do início dos testes.
- Os testes começam ao nível dos componentes e trabalham "para fora" no sentido da integração de todo o sistema.
- Diferentes técnicas de testes são apropriadas para diferentes abordagens de engenharia de software e em diferentes pontos no tempo.
- Os testes são conduzidos pelo desenvolvedor do software e (para grandes projetos) por um grupo de teste independente.
- Os testes e a depuração são atividades diferentes, mas a depuração deve ser acomodada em qualquer estratégia de testes.

Verificação e Validação

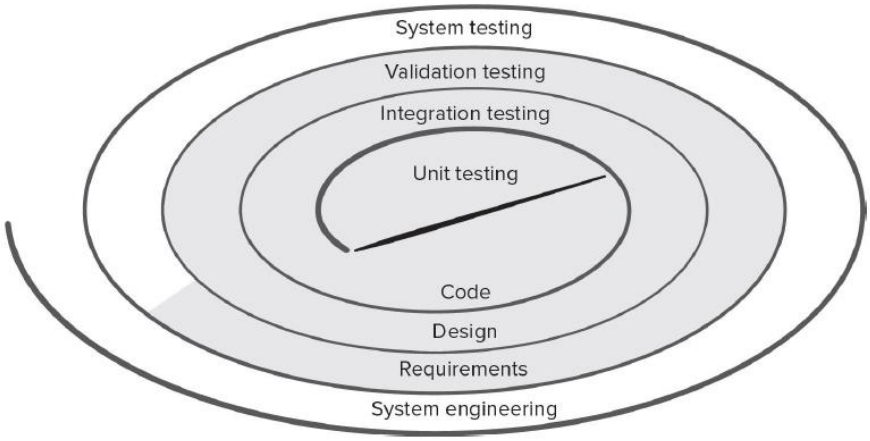
A verificação refere-se ao conjunto de tarefas que asseguram que o software implementa corretamente uma função específica. Ex: "Estamos a construir bem o produto?"

A validação refere-se a um conjunto diferente de tarefas que asseguram que o software que foi construído é rastreável às exigências do cliente. Ex: "Estamos a construir produto correto?"

Organização para testes

- Os programadores de software são sempre responsáveis por testar os componentes individuais do programa e por assegurar que cada um deles desempenha a sua função ou comportamento de conceção.
- Só depois da arquitetura do software estar completa é que um grupo de teste independente se envolve.
- O papel de um grupo de teste independente (ITG) é remover os problemas inerentes a deixar o construtor testar a coisa que foi construída.
- O pessoal do ITG é pago para encontrar error.
- Os programadores e o ITG trabalham de perto ao longo de um projeto de software para assegurar a realização de testes exaustivos.

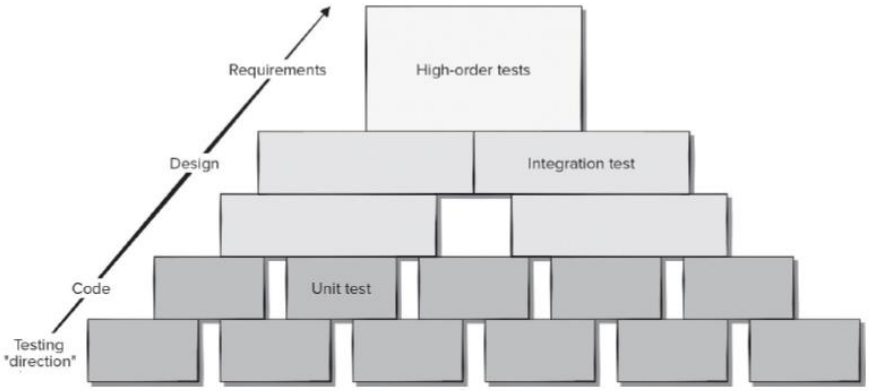
Estratégia de testes



Testar a Big Picture

- Os testes unitários começam no centro da espiral e concentram-se em cada unidade (por exemplo, componente, classe, ou objeto de conteúdo) à medida que são implementados no código fonte.
- Os testes progridem para testes de integração, onde o foco é o design e a construção da arquitetura do software.
- O teste de validação, é onde os requisitos estabelecidos como parte da modelação de requisitos são validados em relação ao software que foi construído.
- Nos testes do sistema, o software e os outros elementos do sistema são testados como um todo.

Passos de Teste de Software



Quando é que os testes são feitos?

Critérios para Realizado

- Nunca se termina os testes; o fardo passa simplesmente do engenheiro de software para o utilizador final. (Errado).
- Os testes terminam quando acaba o tempo ou o dinheiro. (Errado).
- A abordagem de garantia de qualidade estatística sugere a execução de testes derivados de uma amostra estatística de todas as execuções possíveis do programa por todos os utilizadores alvo.
- Ao recolher métricas durante os testes de software e ao fazer uso dos modelos estatísticos existentes, é possível desenvolver orientações significativas para responder à pergunta: “Quando é que terminamos os testes?”

Planeamento de testes

1. Especificar os requisitos do produto de uma forma quantificável muito antes do início dos testes.
2. Objetivos dos testes do Estado explicitamente.
3. Compreender os utilizadores do software e desenvolver um perfil para cada categoria de utilizadores.
4. Desenvolver um plano de testes que dê ênfase aos “testes de ciclo rápido”.
5. Construir software “robusto” que é concebido para se testar a si próprio.
6. Utilizar revisões técnicas eficazes como um filtro antes dos testes.
7. Realizar revisões técnicas para avaliar a estratégia do teste e os próprios casos de teste.
8. Desenvolver uma abordagem de melhoria contínua para o processo de ensaio.

Manutenção de registos de testes

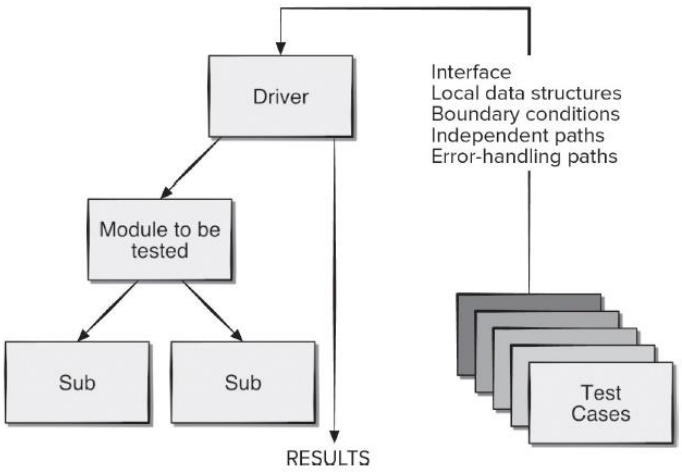
Os casos de teste podem ser registados na folha de cálculo do Google Docs:

- Descreve sucintamente o caso teste.
- Contém um ponteiro para o requisito que está a ser testado.
- Contém os resultados esperados dos dados dos casos de teste ou os critérios de sucesso.
- Indicar se o teste foi aprovado ou reprovado.
- Datas em que o caso de teste foi realizado.
- Deve haver espaço para comentários sobre as razões pelas quais um teste pode ter falhado (ajudas na depuração).

Papel do Andaime

- Os componentes não são um programa autónomo, é necessário algum tipo de andaime para criar uma estrutura de teste.
- Como parte desta estrutura, o driver e/ou software de stub deve ser frequentemente desenvolvido para cada teste de unidade.
- Um condutor não é mais do que um "programa principal" que aceita dados de casos de teste, passa esses dados para o componente (a ser testado), e imprime resultados relevantes.
- Os Stubs (subprograma dummy) servem para substituir os módulos invocados pelo componente a ser testado.
- Um stub utiliza a interface do módulo, pode fazer uma manipulação mínima dos dados, verifica a entrada das impressões, e devolve o controlo ao módulo em teste.

Ambiente de teste da unidade



Testes de custo-eficácia

- Os testes exaustivos exigem que todas as combinações e encomendas de valores de entrada possíveis sejam processadas pelo componente de teste.
- O retorno de testes exaustivos não vale muitas vezes o esforço, uma vez que os testes por si só não podem ser utilizados para provar que um componente é corretamente implementado.
- Os testadores devem trabalhar de forma mais inteligente e atribuir os seus recursos de teste a módulos cruciais para o sucesso do projeto ou àqueles que se suspeite serem suscetíveis de erro como o foco dos seus testes unitários.

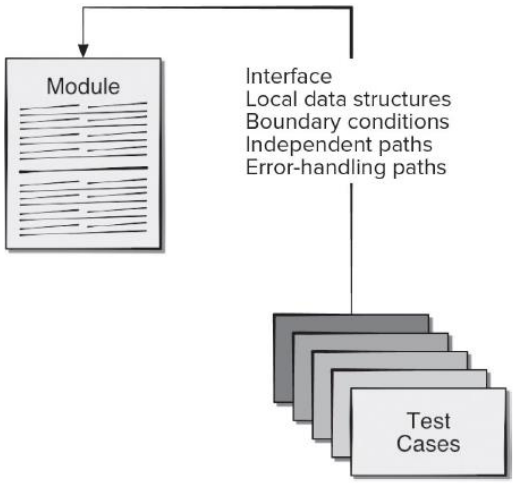
Desenho de casos de teste

Desenhe casos de teste de unidades antes de desenvolver o código de um componente para garantir o código que passará nos testes.

Os casos de teste são concebidos para cobrir as seguintes áreas:

- A interface do módulo é testada para assegurar que a informação flui corretamente para dentro e para fora da unidade de programa.
- As estruturas de dados locais são examinadas para assegurar que os dados armazenados mantêm a sua integridade durante a execução.
- São exercidos caminhos independentes através de estruturas de controlo para assegurar que todas as declarações são executadas pelo menos uma vez.
- As condições-limite são testadas para assegurar que o módulo funciona corretamente nos limites estabelecidos para limitar ou restringir o processamento.
- Todas as vias de tratamento de erros são testadas.

Testes de módulo



Tratamento de erros

- Uma boa conceção antecipa as condições de erro e estabelece caminhos de manipulação de erros que devem ser testados.
- Entre os erros potenciais que devem ser testados quando se avalia o tratamento de erros são:
    - A descrição do erro é ininteligível.
    - O erro assinalado não corresponde ao erro encontrado.
    - A condição de erro provoca a intervenção do sistema antes do tratamento do erro.
    - O processamento de condições de exceção é incorreto.
    - A descrição do erro não fornece informação suficiente para ajudar na localização da causa do erro.

Rastreabilidade

- Para assegurar que o processo de teste é auditável, cada caso de teste tem de ser rastreável até requisitos funcionais ou não funcionais específicos ou requisitos anti-funcionais.
- Muitas vezes, os requisitos não funcionais têm de ser restreáveis a requisitos empresariais ou arquitetónicos específicos.
- Muitas falhas no processo de teste podem ser rastreadas até percursos de rastreabilidade em falta, dados de teste inconsistentes, ou cobertura de teste incompleta.
- Os testes de regressão requerem um novo teste de componentes selecionados que podem ser afetados por alterações feitas a outros componentes de software colaborantes.

Teste da caixa branca

Utilizando o método de teste da caixa branca, pode derivar casos de teste que:

1. Garantem que todos os caminhos independentes dentro de um módulo foram exercidos pelo menos uma vez.
2. Exercem todas as decisões lógicas sobre os lados verdadeiro e falso.
3. Executam todos os loops nos seus limites e dentro dos seus limites operacionais.
4. Exercem estruturas de dados internas para assegurar a sua validade.

Teste do caminho de base

Determina o número de caminho independentes no programa através da computação da Complexidade Ciclomática:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as

   $V(G) = E - N + 2$

   E is the number of flow graph edges

   N is the number of nodes.

3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

   $V(G) = P + 1$

   P is number of predicate nodes contained in the flow graph G.

Cyclomatic Complexity of the flow graph is 4

1. The flow graph has four regions.
2. $V(G)$ = 11 edges − 9 nodes + 2 = 4.
3. $V(G)$ = 3 predicate nodes + 1 = 4.

An ***independent path*** is any path through the program that introduces at least one new set of processing statements or a new condition (we need 4 independent paths to test)

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

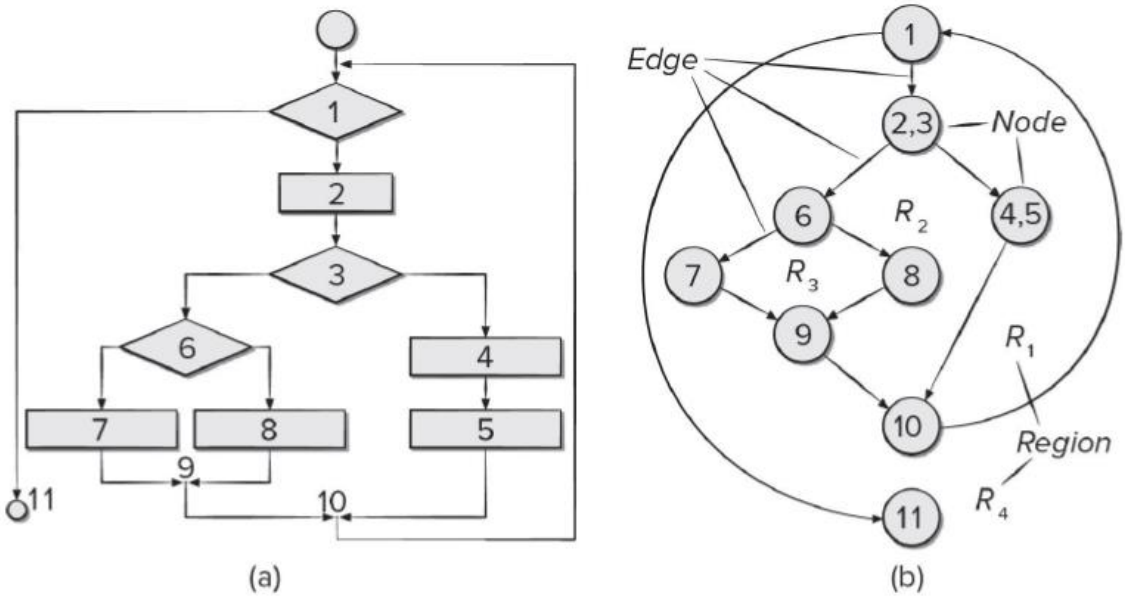
Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Designing Test Cases

- Using the design or code as a foundation, draw a corresponding flow graph.
- Determine the cyclomatic complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.

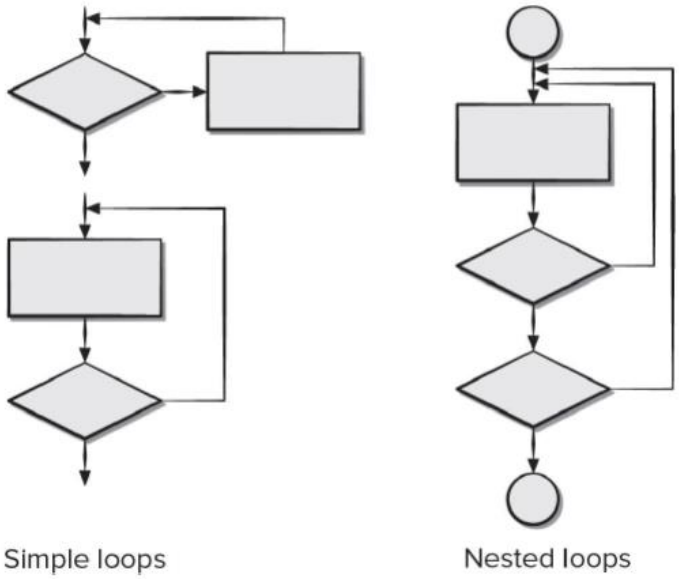
Flowchart (a) e Flow Graph (b)



Teste da Estrutura de Controlo

- O teste das condições é um método de conceção de caso de teste que exerce as condições lógicas contidas num módulo de programa.
- Os testes de fluxo de dados selecionam os caminhos de teste de um programa de acordo com os locais das definições e usos das variáveis do programa.
- O teste de loop é uma técnica de teste de caixa branca que se concentra exclusivamente na validade das construções de laço.

Classes de Loop



Teste de loop

Test cases for simple loops:

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where $m < n$.
5. $n - 1$, n, $n + 1$ passes through the loop.

Test cases for nested loops:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (for example, loop counter) values.
3. Add other tests for out-of-range or excluded values.
4. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
5. Continue until all loops have been tested.

Teste da caixa negra

A caixa negra (funcional) de testes tenta encontrar erros nas seguintes categorias:

1. Funções incorretas ou em falta.
2. Erros de interface.
3. Erros nas estruturas de dados ou no acesso a bases de dados externas.
4. Erros de comportamento ou de desempenho.
5. Erros de inicialização e erros de terminação.

Ao contrário do teste da caixa branca, que é realizado no início do processo de testes, o teste da caixa preta tende a ser aplicado durante fases posteriores do teste.

Os casos de teste da caixa negra são criados para responder a perguntas como:

- Como é que é testada a validade funcional?
- Como é que são testados o comportamento e o desempenho do sistema?
- Quais são as classes de inputs que farão bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como é que são isolados os limites de uma classe de dados?
- Que taxas de dados e volume de dados pode o sistema tolerar?
- Que efeito terão as combinações específicas de dados no funcionamento do sistema?

Caixa negra – Teste de interface

- Os testes de interface são utilizados para verificar se um componente do programa aceita a informação que lhe é passada na ordem e tipos de dados adequados e devolve a informação na ordem e formato de dados adequados.
- Os componentes não são programas isolados que testam interfaces de teste de programas, requerendo o uso de tocos e condutores.
- Os canos e condutores incorporam por vezes casos de teste a serem passados ao componente ou acedidos pelo componente.
- O código de depuração pode ter de ser inserido dentro do componente para verificar se os dados passados foram recebidos corretamente.

Testes orientados a objetos (OOT)

Para testar adequadamente os sistemas OO, três coisas devem ser feitas:

- A definição de testes de ver alargada para incluir técnicas de descoberta de erros aplicadas à análise orientada para objetos e modelos de conceção.
- A estratégia para os testes de unidade e de integração deve mudar significativamente.
- A conceção dos casos de teste deve ter em conta as características únicas do software OO.

Caixa negra – Análise do Valor Limite (BVA)

- ***Boundary value analysis*** leads to a selection of test cases that exercise bounding values.
- Guidelines for BVA:

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the min and max numbers as well as values just above and below min and max.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries (for example, array with max index of 100) be certain to design a test case to exercise the data structure at its boundary.

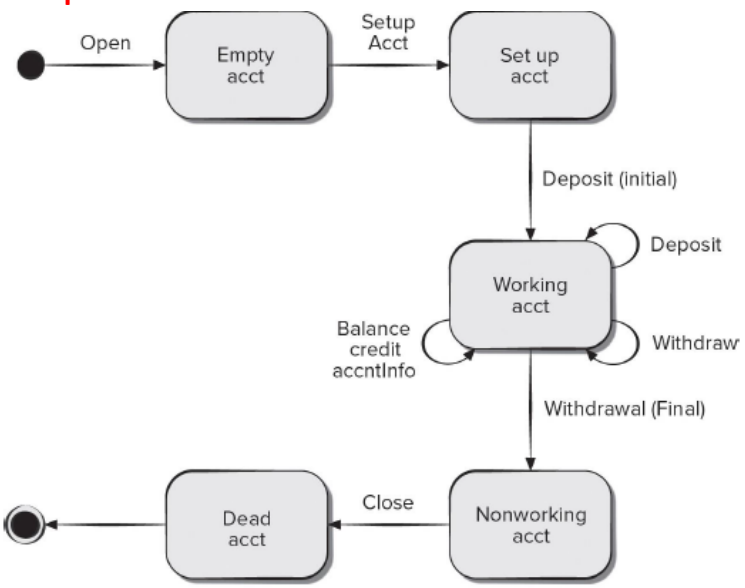
Teste de classe OOT

- O teste de classe para software orientado a objetos (OO) é o equivalente ao teste unitário para softwares convencionais.
- Ao contrário dos testes unitários de software convencional, que tendem a concentrar-se nos detalhes algorítmicos de um módulo e nos dados que fluem através da interface do módulo.
- Os testes de classe para o software OO são impulsionados pelas operações encapsuladas pela classe e pelo comportamento do estado da classe.
- As sequências válidas de operações e as suas permutações são utilizadas para testar que a compartimentação de comportamentos de classe pode reduzir as sequências numéricas necessárias.

OOT – Teste de comportamento

- Um diagrama de estados pode ser utilizado para ajudar a derivar uma sequência de teste que irá exercer um comportamento dinâmico da classe.
- Os testes a serem concebidos devem alcançar uma cobertura total, utilizando sequências de operação que provoquem transições através de todos os estados admissíveis.
- Quando o comportamento de classe resulta numa colaboração com várias classes, podem ser utilizados múltiplos diagramas de estado para acompanhar o fluxo comportamental do sistema.
- Um modelo de estado pode ser percorrido de uma forma ampla, primeiro através de um único exercício de teste de transição e quando uma nova transição deve ser testada, só são utilizadas transições previamente testadas.

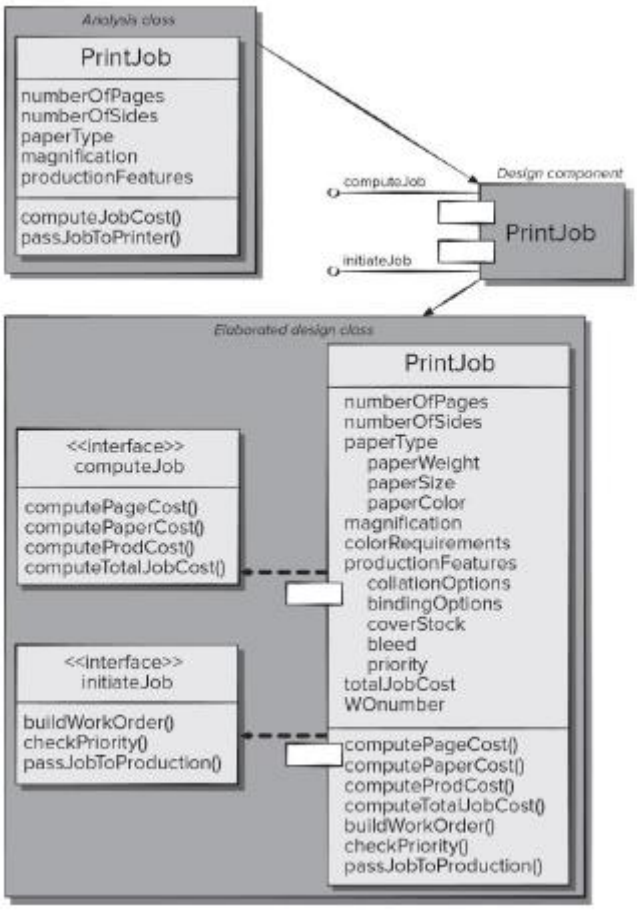
Diagrama de Estado para Classe de Conta



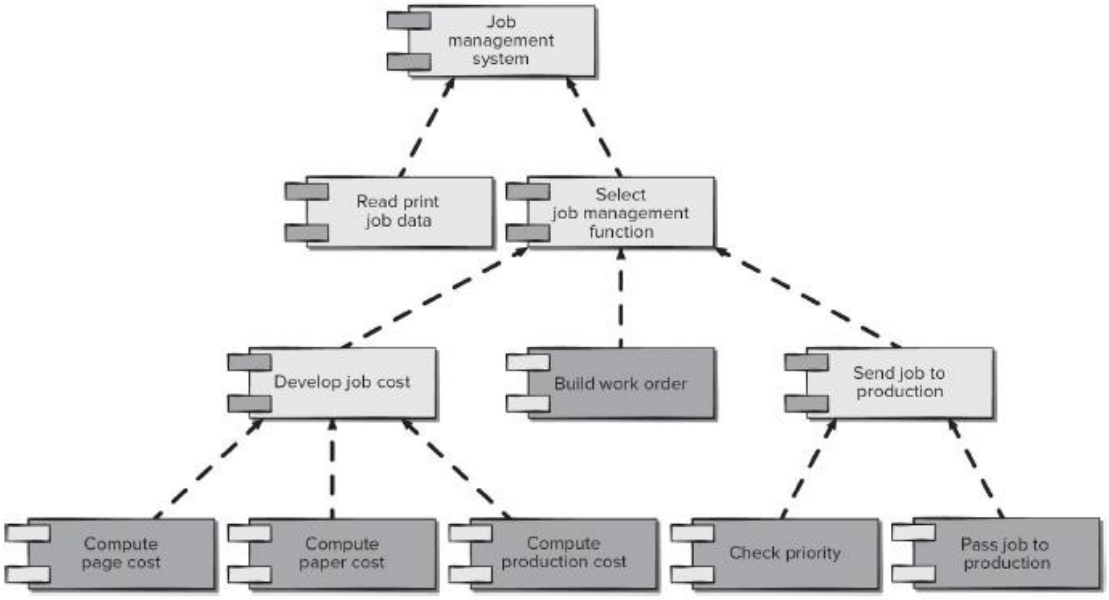
O que é um componente?

- OMG Unified Modeling Language Specification (Especificação Unificada de Linguagem de Modelação OMG) define um componente como
  - "… uma parte modular, destacável e substituível de um sistema que encapsula a implementação e expõe um conjunto de interfaces".
- Visão orientada para os objetos: um componente contém um conjunto de classes colaboradoras.
- Visão tradicional: um componente contém lógica de processamento, estruturas de dados internas necessárias para implementar a lógica de processamento, e uma interface que permite que o componente seja invocado e que os dados lhe sejam transmitidos.
- Visão relacionada com o processo: construção de sistemas a partir de componentes de software reutilizáveis ou padrões de conceção selecionados a partir de um catálogo (engenharia de software baseada em componentes).

Design de nível de componentes com base na classe



Desenho tradicional em nível de componentes



Princípios básicos de desenho de componentes

- Princípio Aberto Fechado (OCP). “Um módulo [componente] deve ser aberto para extensão, mas fechado para modificação”.
- Princípio de Substituição de Liskov (LSP). “As subclasses devem ser substituíveis pelas suas classes base.
- Princípio da Inversão de Dependência (DIP). “Depende das abstrações. Não depende de concreção”.
- Princípio de Segregação de Interfaces (ISP). “Muitas interfaces específicas de clientes são melhores do que uma interface de uso geral”.
- Princípio de Equivalência de Reutilização de Libertação (REP). “O grânulo de reutilização é o grânulo de libertação”.
- Princípio Comum de Encerramento (CCP). “As classes que mudam juntas pertencem umas às outras”.
- Princípio Comum de Reutilização (CRP). “As classes que não são reutilizadas em conjunto não devem ser agrupadas”.

Diretrizes de desenho ao nível dos componentes

- As convenções de nomeação de componentes devem ser estabelecidas para componentes que são especificados como parte do modelo arquitetónico e depois refinados e elaborados como parte do modelo a nível de componentes.
- As interfaces fornecem informações importantes sobre comunicação e colaboração (assim como nos ajudam a alcançar o OPC).
- Dependências e herança – Para a legibilidade, é uma boa ideia modelar dependências da esquerda para a direita e herança da base (classes derivadas) para o topo (classes base).

Coesão

Visão tradicional – a "mentalidade única" de um módulo.

Visão orientada para objetos – a coesão implica que um componente encapsula apenas atributos e operações que estão intimamente relacionados entre si e com o próprio componente.

Níveis de coesão:

- **Funcional** – o módulo efetua um e apenas um cálculo.
- **Camada** – ocorre quando uma camada superior acede aos serviços de uma camada inferior, mas as camadas inferiores não acedem às camadas superiores.
- **Comunicacional** – Todas as operações que acedem aos mesmos dados são definidas dentro de uma classe.

Acoplamento

Visão tradicional – grau em que um componente está ligado a outros componente e ao mundo exterior.

Visão orientada para o objeto – medida qualitativa do grau em que as classes estão ligadas umas às outras.

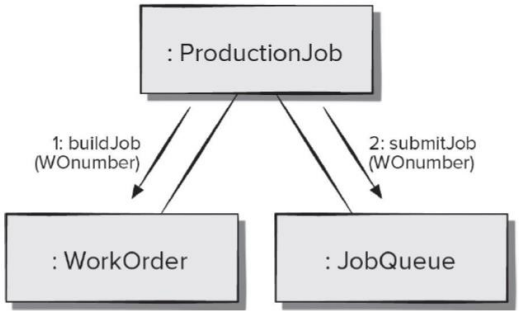
Níveis de acoplamento:

- **Conteúdo** – ocorre quando um componente "modifica sub-repticiamente dados que são internos a outro componente".
- **Controlo** – ocorre quando a bandeira de controlo é passada aos componentes para solicitar comportamentos alternativos quando invocada.
- **Externo** – ocorre quando um componente comunica ou colabora com componentes de infraestrutura.

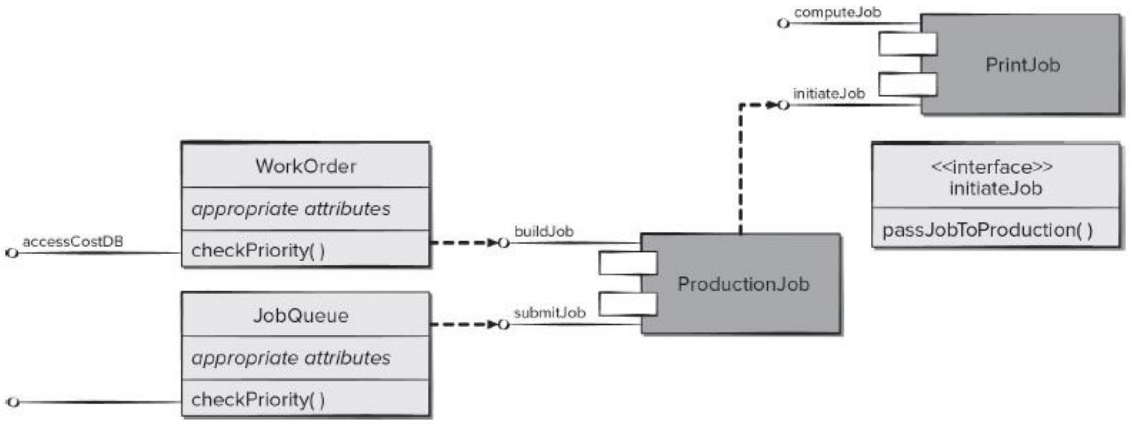
Desenho ao nível dos componentes

- Passo 1. Identificar todas as classes de desenho que correspondem ao domínio do problema.
- Passo 2. Identificar todas as classes de desenho que correspondem ao domínio das infraestruturas.
- Passo 3. Elaborar todas as classes de desenho que não são adquiridas como componentes reutilizáveis.
- Passo 3a. Especificar os detalhes da mensagem quando as classes ou componentes colaboram.
- Passo 3b. Identificar interfaces apropriadas para cada componente.
- Passo 3c. Elaborar atributos e definir tipos de dados e estruturas de dados necessários para os implementar.
- Passo 3d. Descrever em pormenor o fluxo de processamento dentro de cada operação.
- Passo 4. Descrever fontes de dados persistentes (bases de dados e ficheiros) e identificar as classes necessárias para a sua gestão.
- Passo 5. Desenvolver e elaborar representações comportamentais para uma classe ou componente.
- Passo 6. Elaborar diagramas de implantação para fornecer detalhe de implementação.
- Passo 7. Fator de cada representação de design ao nível de componentes e considerar sempre alternativas.

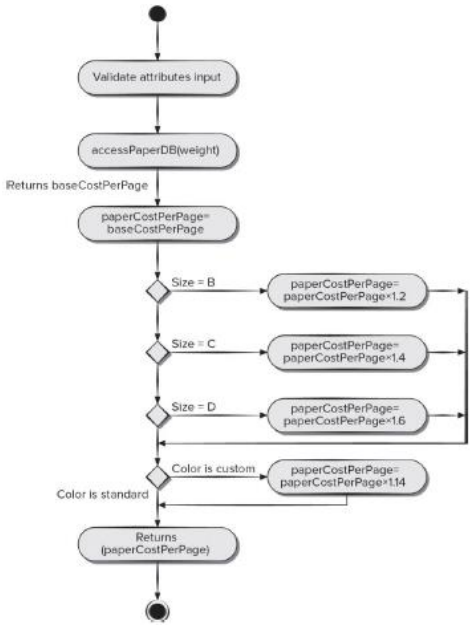
Diagrama de Colaboração com Detalhe da Mensagem



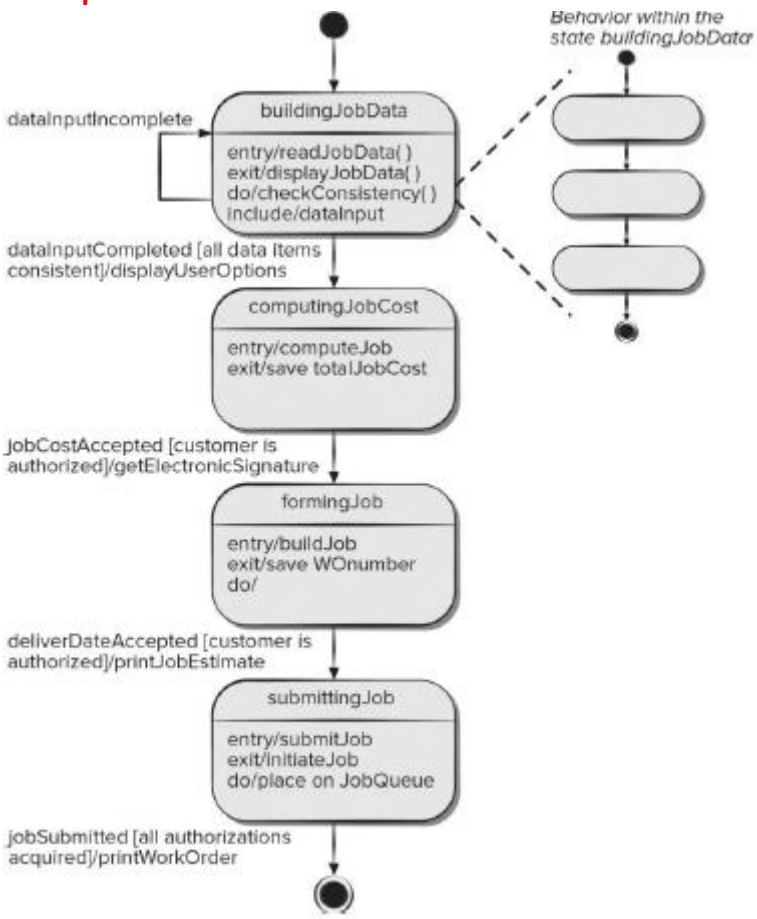
Definir Interfaces



Descrever o fluxo de processamento



Representações Comportamentais Elaboradas



Design de nível de componentes para WebApps

A componente WebApp é:

1. Uma função coesiva bem definida que manipula o conteúdo ou fornece processamento computacional ou de dados para um utilizador final-
2. Um pacote coesivo de conteúdo e funcionalidade que fornece ao utilizador final alguma capacidade necessária.

A conceção a nível de componentes para WebApps incorpora frequentemente elementos de conceção de conteúdo e de conceção funcional.

Conceção do conteúdo da WebApp

Concentra-se nos objetos de conteúdo e na forma como estes podem ser embalados para apresentação a um utilizador final da WebApp.

Considera uma capacidade de videovigilância baseada na Web no âmbito de SafeHomeAssured.com. podem ser definidos componentes potenciais de conteúdo para a capacidade de vídeo de vigilância.

1. Os objetos de conteúdo que representam a disposição do espaço (a planta) com ícones adicionais que representam a localização dos sensores e das câmaras de vídeo.
2. A coleção de capturas de vídeo em miniatura (cada um deles um objeto de dados separado)
3. A Janela de vídeo streaming para uma câmara específica.

Cada um destes componentes pode ser nomeado e manipulador separadamente como um pacote.

Desenho Funcional WebApp

As aplicações Web modernas oferecem funções de processamento cada vez mais sofisticadas que:

1. Executam processamento localizado para gerar conteúdo e capacidade de navegação de uma forma dinâmica.
2. Forneçam capacidade de cálculo ou processamento de dados que seja apropriada para o domínio comercial da WebApp.
3. Forneçam consulta e acesso a base de dados sofisticadas
4. Estabelecem interfaces de dados com sistemas corporativos externos.

Para alcançar estas (e muitas outras) capacidades, irá conceber e construir componentes funcionais WebApp que são idênticos em forma aos componentes de software para software convencional.

Design de nível de componentes para aplicações móveis

Cliente fino baseado na web.

- Camada de interface apenas no dispositivo.
- Camadas de negócios e de dados implementadas utilizando serviços web ou em nuvem.

Cliente rico.

- As três camadas (interface, negócio, dados) implementadas no dispositivo.
- Sujeito a limitações de dispositivos móveis.

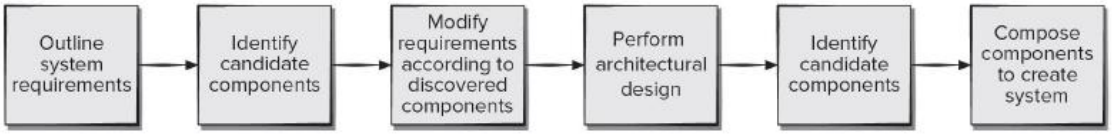
Desenho tradicional em nível de componentes

- A conceção da lógica de processamento é governada pelos princípios básicos da conceção de algoritmos e programação estruturada.
- A conceção das estruturas de dados é definida pelo modelo de dados desenvolvido para o sistema.
- A conceção de interfaces é governada pelas colaborações que um componente deve efetuar.

Engenharia de Software Baseado em Componente (CBSE)

A equipa de software pergunta:

- Estão disponíveis componentes comerciais fora da prateleira (COTS) para implementar o requisito?
- Estão disponíveis componentes reutilizáveis para implementar o requisito?
- As interfaces para os componentes disponíveis são compatíveis dentro da arquitetura do sistema a ser construído?



Benefícios do CBSE

- Redução do tempo de espera. É mais rápido construir aplicações completas a partir de um conjunto de componentes existentes.
- Maior retorno do investimento (ROI). Por vezes, é possível realizar poupanças através da compra de componentes, em vez de se desenvolver novamente a mesma funcionalidade internamente.
- Alavancou os custos de desenvolvimento de componentes. A reutilização de componentes em múltiplas aplicações permite que os custos sejam repartidos por múltiplos projetos.
- Qualidade melhorada. Os componentes são reutilizados e testados em muitas aplicações diferentes.
- Manutenção de aplicação baseadas em componentes. Com uma engenharia cuidadosa, pode ser relativamente fácil substituir componentes obsoletos por componentes novos ou melhorados.

Riscos CBSE

- Riscos de seleção de componentes. É difícil prever o comportamento do componente da caixa negra, ou pode haver um mapeamento deficiente dos requisitos do utilizador para a conceção arquitetónica dos componentes.
- Riscos de integração de componentes. Há uma falta de normas de interoperabilidade entre componentes; isto requer muitas vezes a criação de "código de embalagem" para a interface dos componentes.
- Riscos de qualidade. Assunções de conceção desconhecidas feitas para os componentes tornam os testes mais difíceis, e isto pode afetar a segurança, o desempenho e a fiabilidade do sistema.
- Risco de segurança. Um sistema pode ser utilizado de forma não intencional, e as vulnerabilidades do sistema podem ser causadas pela integração de componentes em combinações não testadas.
- Riscos de evolução do sistema. Os componentes atualizados podem ser incompatíveis com os requisitos do utilizador ou conter características adicionais não documentadas.

Refactoring de componentes

- A maioria dos criadores concordariam que a refactoria de componentes para melhorar a qualidade é uma boa prática.
- É difícil convencer a gestão a gastar recursos a fixar componentes que estão a funcionar corretamente em vez de lhes acrescentar novas funcionalidades.
- Mudar o software e não documentar as alterações pode levar a um aumento da dívida técnica.
- A redução desta dívida técnica envolve frequentemente a refaturação arquitetónica, que é geralmente vista pelos promotores como dispendiosa e arriscada.
- Os desenvolvedores podem criar ferramentas para examinar os históricos de mudança para identificar as oportunidades de rafactoring mais rentáveis.

Arquiteturas de software

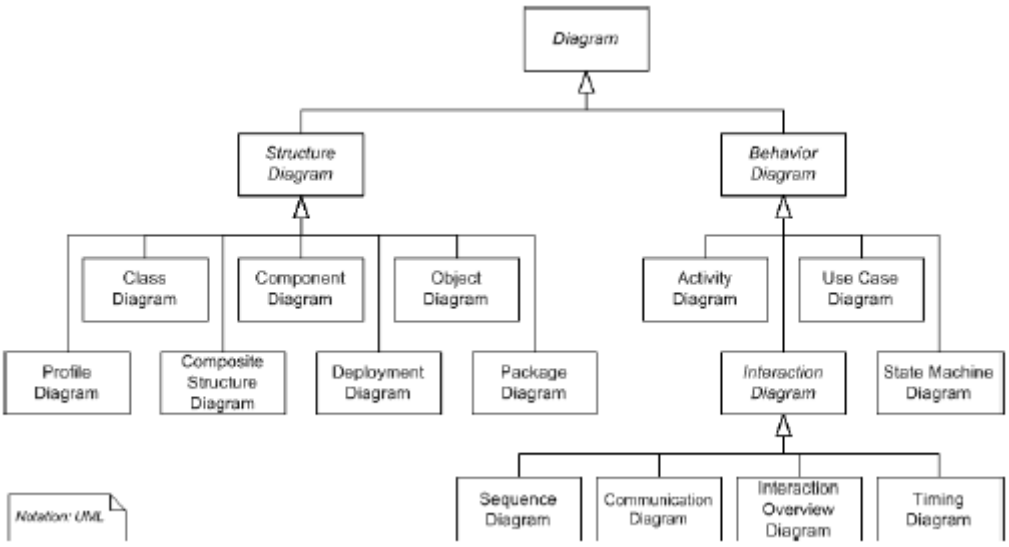
Visões a nível de arquiteturas de software

Uma Visão Arquitetural pode ser definida como uma coleção de padrões, modelos e convenções para construir um tipo de visão. Ela define os stakeholders que têm os seus interesses refletidos, assim como os princípios, modelos e formas de se construir as suas visões.

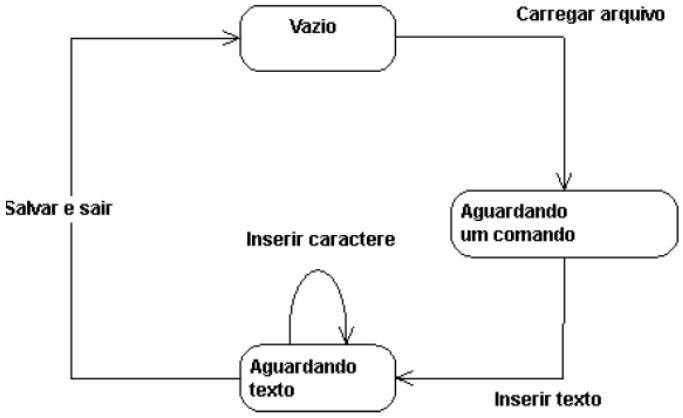
Diferentes tipos de visões

Visão de Projeto (ou Lógica):

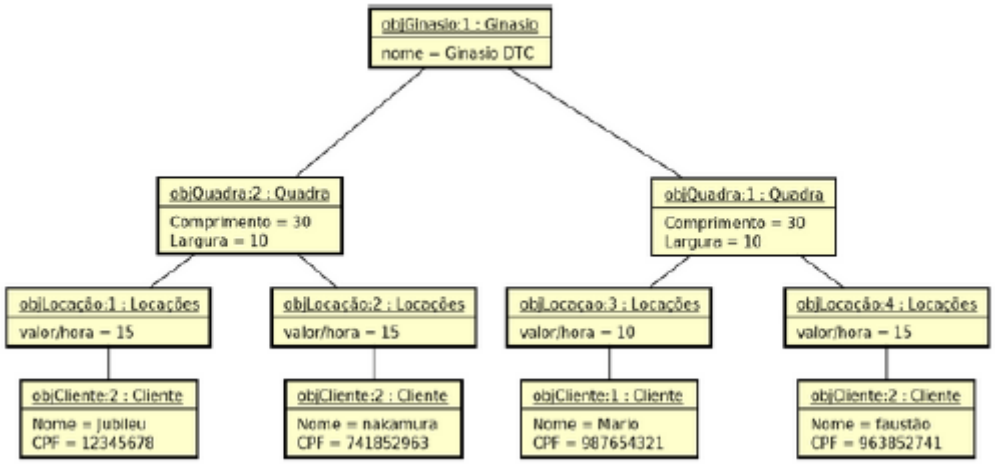
- Suporte aos requisitos funcionais (o que o sistema deve fornecer em termos de serviços aos seus utilizadores).
- Foco nas funcionalidades;
- Ligado ao domínio do problema;
- Descreve e específica a estrutura estática do sistema e as colaborações dinâmicas entre objetos;
- Exemplos de diagramas a apresentar:
    - Diagrama de Classe: mostra as classes e representa as relações lógicas entre ambas;



- Diagrama de Estado;

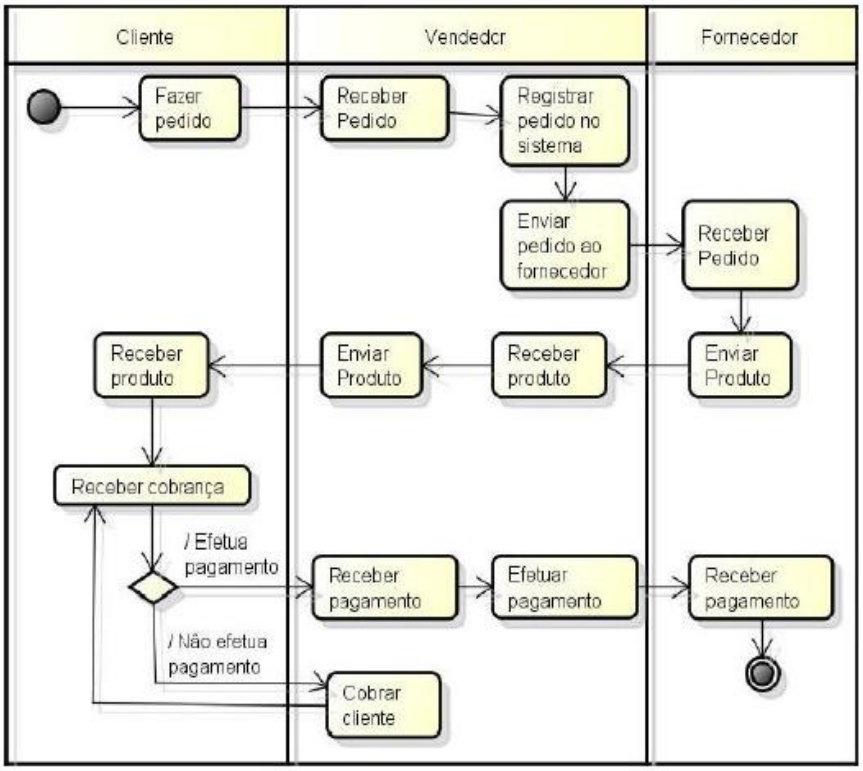


- Diagramas de Objetos.

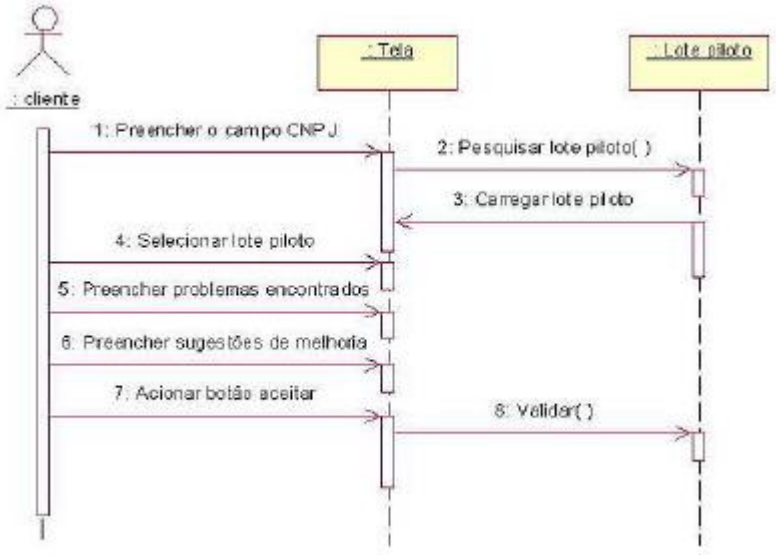


Visão de Processo (ou Concorrência):

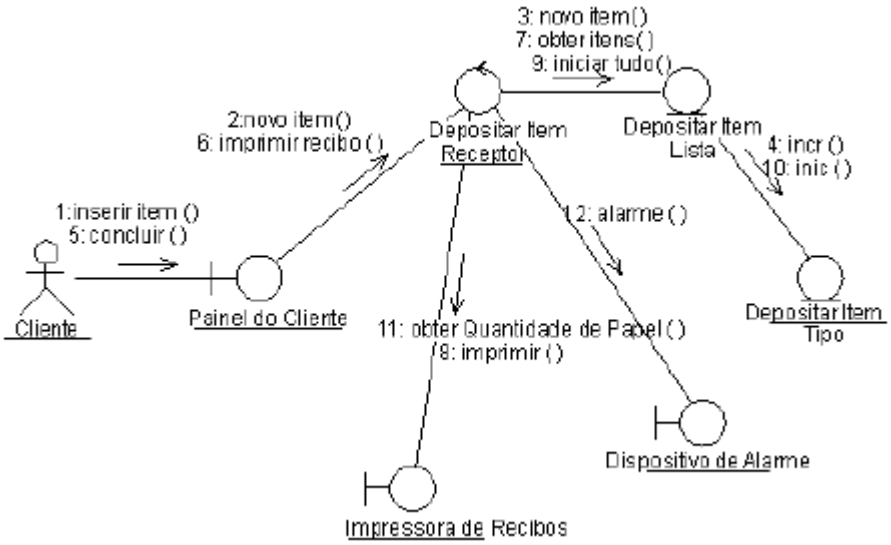
- O sistema é dividido em linhas de execução de processos concorrentes (threads).
- Esta visão de concorrência deverá mostrar como se dá a comunicação e a concorrência destas threads.
- Considera questões de performance e disponibilidades.
- Exemplos de diagramas a apresentar:
    - Diagramas de Atividade;



    - Diagramas de Sequência;

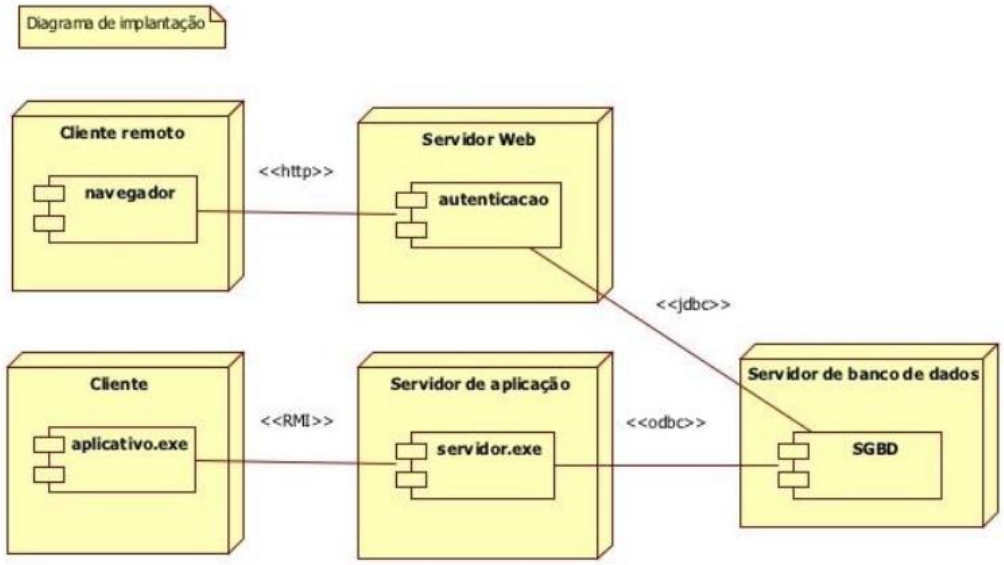


    - Diagrama de Comunicação.


3: novo item()
7: obter itens()
9: iniciar tudo()
2:novo item()
6: imprimir recibo()
Depositar Item
Recepto
Depositar Item
Lista
4: incr()
10: inic()
1:inserir item()
5: concluir()
12: alarme()
Cliente
Painel do Cliente
11: obter Quantidade de Papel()
8: imprimir()
Depositar Item
Tipo
Dispositivo de Alarme
Impressora de Recibos

Visão Física:

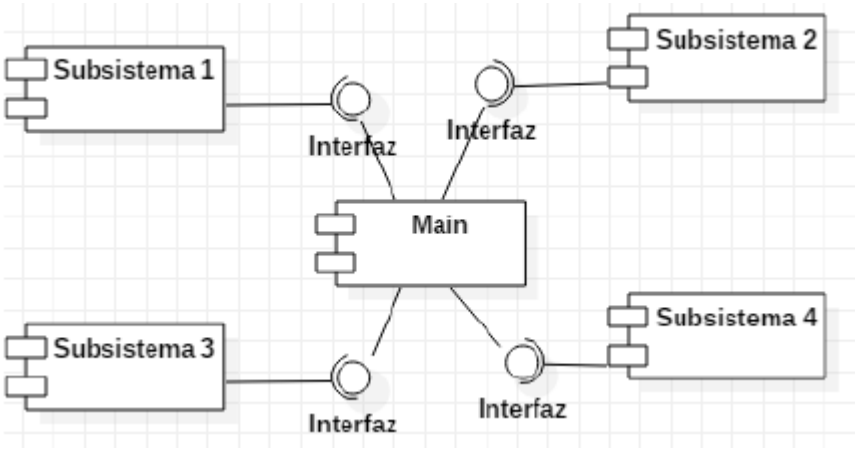
- A arquitetura física tem em consideração principalmente os requisitos não funcionais do sistema, como disponibilidade, confiabilidade (tolerância a falhas), desempenho e escalabilidade.
- Tem em conta a topologia dos componentes de software na camada física e as conexões físicas entre esses componentes.
- Exemplo de diagrama a apresentar:
    - Diagrama de Instalação.



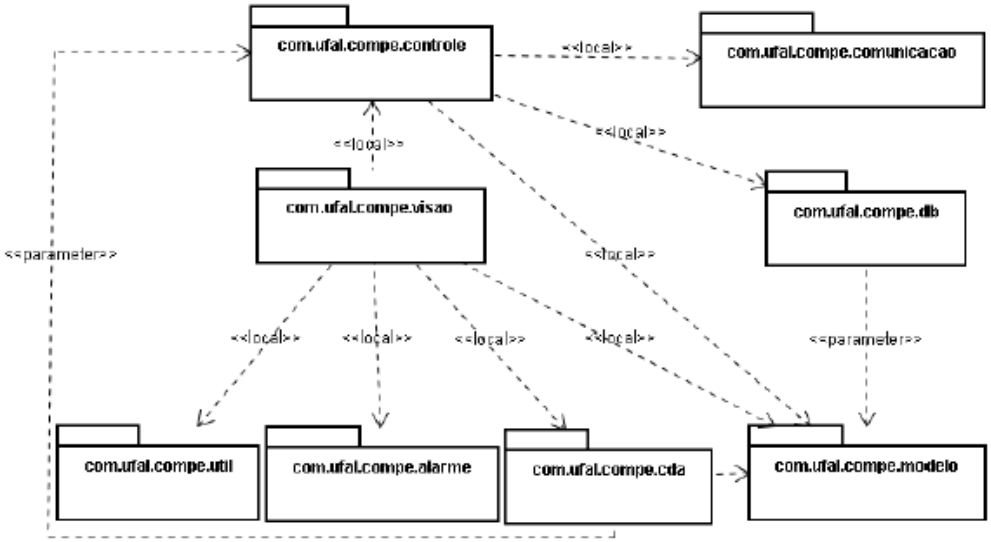
Visão de desenvolvimento:

- Concentra-se na organização real do módulo de software no software ambiente de desenvolvimento.
- O software é colocado em pequenos blocos – bibliotecas ou subsistemas.
- Exemplos de diagramas a apresentar:

    - Diagramas de Componentes;

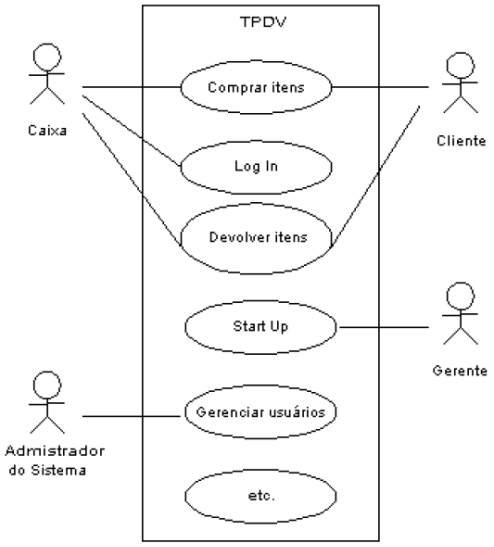


    - Diagrama de Pacotes.



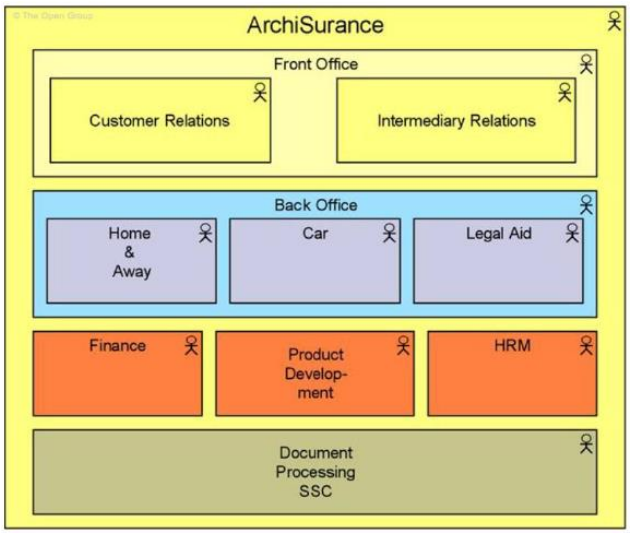
Visão de ação do utilizador:

- Apresenta uma visão próxima do utilizador.
- Descreve os cenários de uso da aplicação.
- Os cenários descrevem sequências de interações entre objetos e entre processos.
- Normalmente é a primeira visão construída.
- Exemplo de diagrama a apresentar:
    - Diagramas de Use Case.

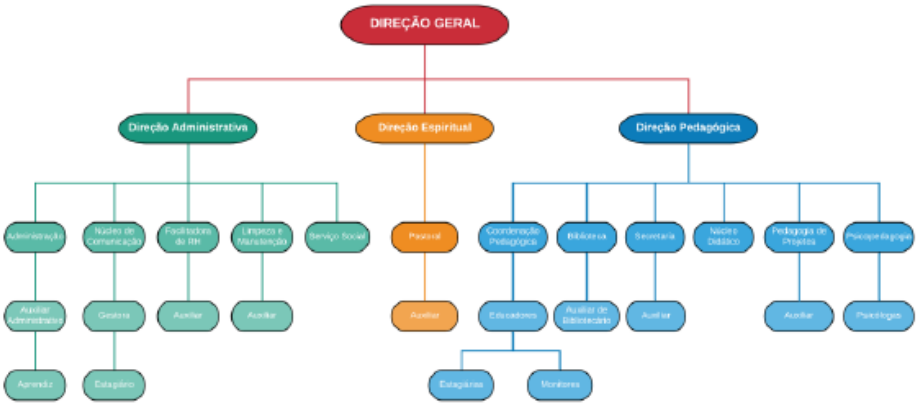


Visão de organização:

- Centra-se na organização (interna) de uma empresa, um departamento, uma rede de empresas ou de outra entidade organizacional
- Identifica competências, autoridade e responsabilidades em uma organização.
- Exemplos de diagramas a apresentar:
    - Diagrama de blocos aninhados;

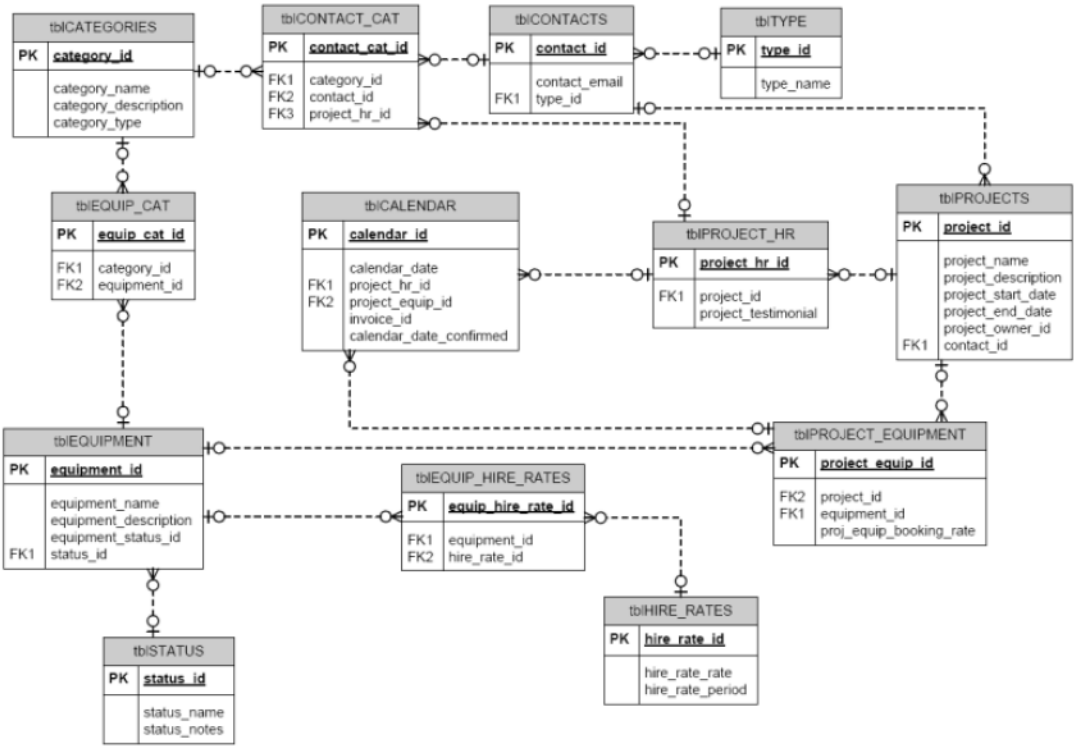


    - Organograma.



Visão dados e informação:

- Mostra a arquitetura dos dados e como a informação está organizada.
- Visualiza a estrutura de dados.
- Exemplo do diagrama a apresentar:
    - Entity Relationship Diagram

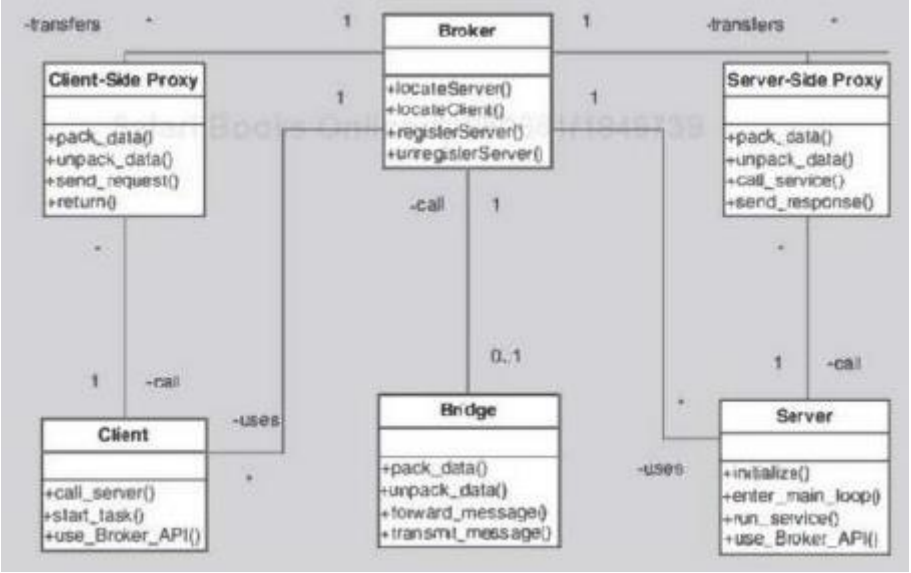


Padrões de arquitetura de software

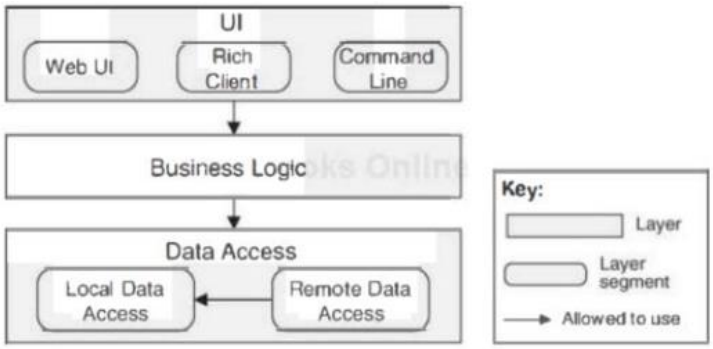
- São formas de captar boas estruturas de desenho, de forma a que estas possam ser reutilizáveis.
- É um conjunto de decisões de Desenho que são utilizadas repetidamente.
- Tem propriedades bem definidas, que podem ser reutilizadas.
- Descreve uma classe de arquiteturas.

Padrões de arquitetura software – componente e conector

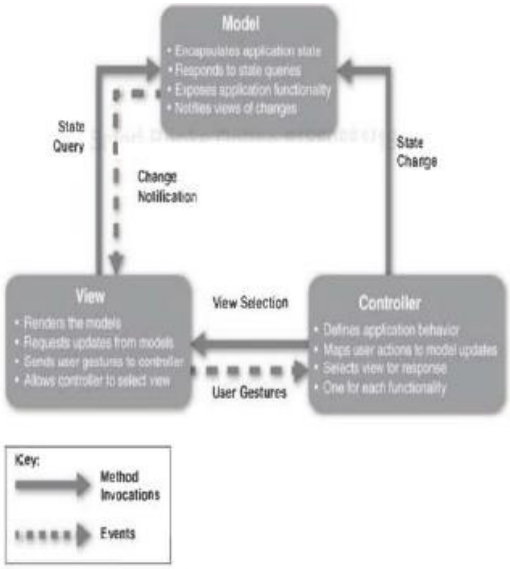
- **Padrão Broker**
  - Utiliza um intermediário para localizar um servidor apropriado para responder ao pedido de um cliente, encaminha o pedido para o servidor, e retorna o resultado ao cliente.



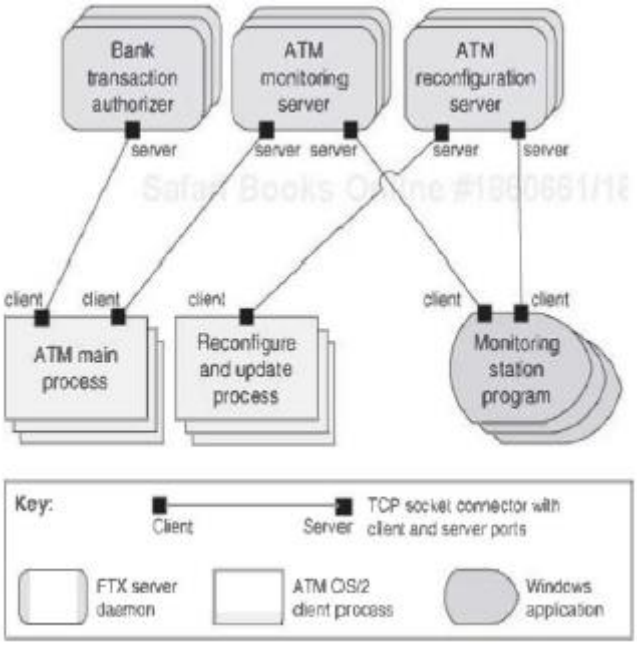
- **Padrão por camadas**
  - Define camadas (conjuntos de módulos que oferecem um conjunto de serviços coeso) e uma relação unidirecional "allowed-to-use" entre camadas.



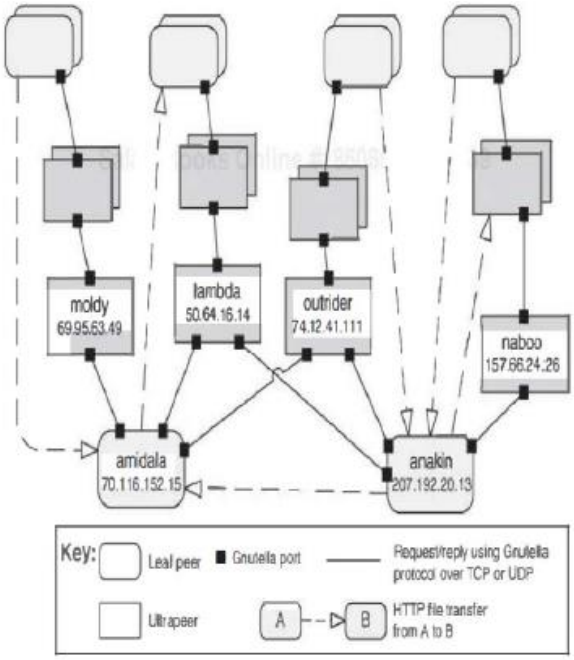
- **Padrão MVC**
  - Separa as funcionalidades da aplicação em 3 componentes:
    - Model – contém os dados da aplicação
    - View – apresente os dados e interage com o utilizador
    - Controller – faz a mediação entre o Model e a View, e gere as notificações e alterações de estado.
  - Os componentes no MVC estão ligados entre eles através de algum tipo de notificação, tal como eventos ou callbacks.



- **Padrão Cliente Servidor**
  - Os clientes iniciam interação com o servidor invocando serviços conforme necessário, e esperando pelos resultados desses pedidos. O melhor exemplo é a World Wide Web.

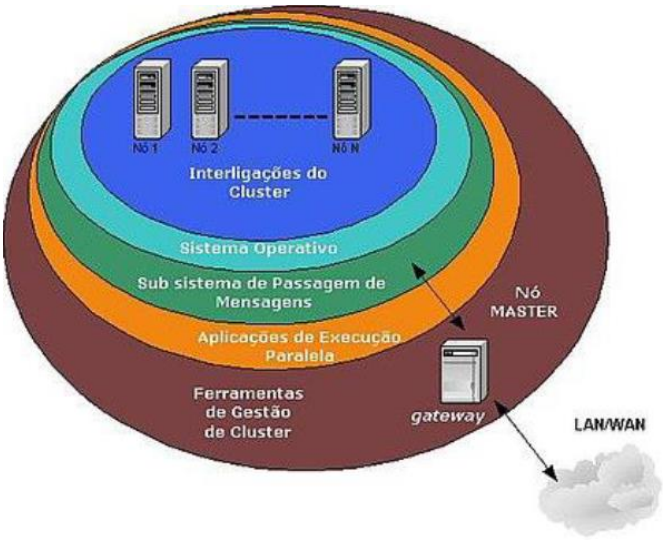


- **Padrão Peer-to-peer**
    - Neste padrão, a computação é atingida pela cooperação de peers que pedem e oferecem serviços entre si numa rede.
    - É utilizado maioritariamente em aplicações de computação distribuída, tal como partilha de ficheiros (BitTorrent p.ex), mensagens instantâneas (Skype).
        - Neste tipo de padrão, gerir a segurança, a consistência dos dados, e a disponibilidade de serviços é mais complexo.



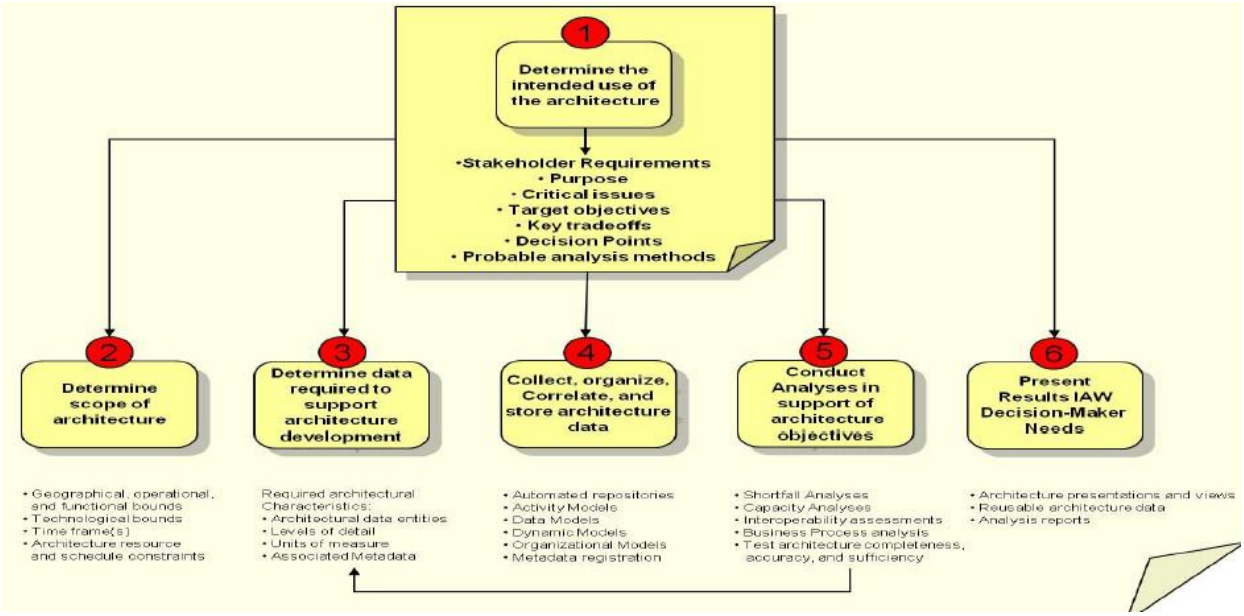
Padrões de arquitetura de software – computação distribuída

- Um sistema de processamento distribuído ou paralelo é um sistema que interliga vários nós de processamento (computadores individuais, não necessariamente homogéneos) de maneira que um processo de grande consumo seja executado no nó “mais disponível”, ou mesmo subdividido por vários nós.
- Conseguindo-se, portante, ganhos óbvios nestas soluções: uma tarefa qualquer, se divisível em várias subtarefas pode se realizada em paralelo.



DODAF – Department of Defense Architecture Framework

- Como o nome indica, é uma framework de desenvolvimento de arquitetura, desenhada especificamente para utilização por parte do departamento de defesa dos Estados Unidos da América.
- O seu desenvolvimento segue 6 passos
    1. Determinar a utilização pretendida da arquitetura
    2. Determinar o contexto da arquitetura
    3. Determinar os dados necessários para suportar o desenvolvimento da arquitetura
    4. Recolher, organizar, correlacionar e armazenar dados arquiteturais
    5. Realizar análises de suporte aos objetivos arquiteturais
    6. Documentar os resultados de acordo com as necessidades do tomador de decisões

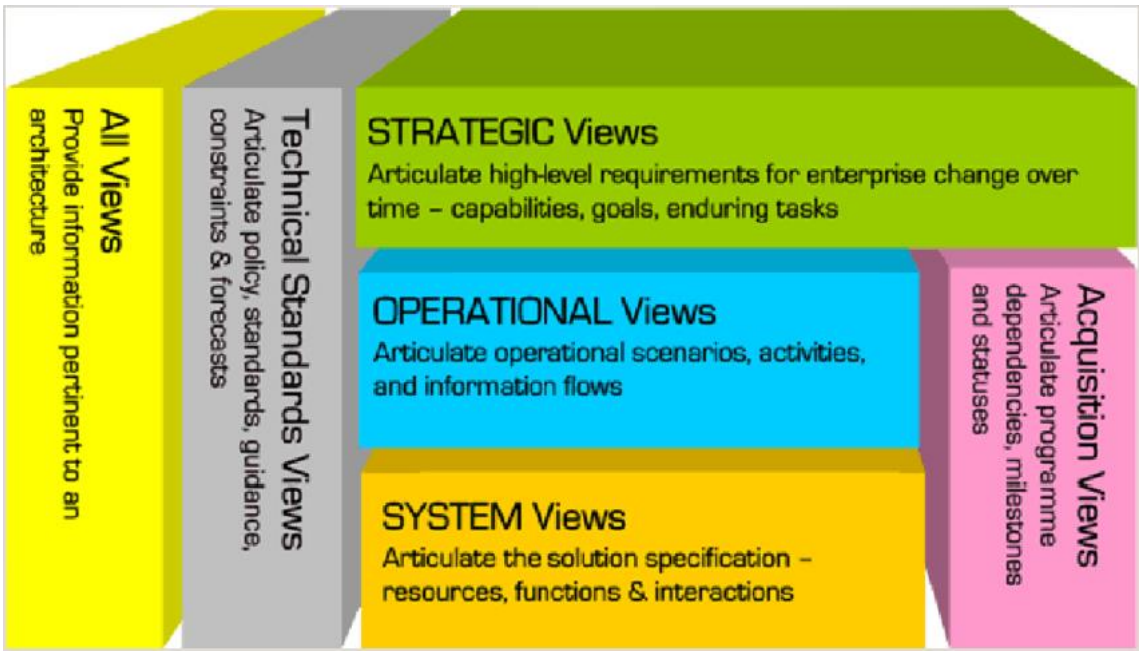


- Estes passos guiam a equipa de desenvolvimento de arquitetura num processo que segue uma abordagem orientada aos dados invés do produto, enfatizando o foco nos dados, e nas relações entre os dados.
- Esta abordagem assegura que todas as relações entre os dados são captadas para suportar uma variedade de tarefas de análise.
- As views criadas com base nos resultados do desenvolvimento da arquitetura demonstram visualmente os dados da arquitetura, e informação de interesse da Descrição arquitetural, que preencha as necessidades de utilizadores específicos, tomadores de decisões, ou comunidades específicas.
- O desenvolvimento da Descrição Arquitetural é um processo único e iterativo.

All Viewpoint
Underarching aspects of architecture context that relate to all views
Data and Information Viewpoint
Articulate the data relationships and alignment structures in the architecture content
Standards Viewpoint
Articulate applicable Operational, Business, Technical, and Industry policy, standards, guidance, constraints, and forecasts
Capability Viewpoint
Articulate the capability requirement, delivery timing and deployed capability
Operational Viewpoint
Articulate operational scenarios, processes, activities & requirements
Services Viewpoint
Articulate the performers, activities, services, and their exchanges providing for, or supporting, DoD functions
Systems Viewpoint
Articulate the legacy systems or independent systems, their composition, interconnectivity and context providing for, or supporting, DoD functions
Project Viewpoint
Describes the relationships between operational and capability requirements and the various projects being implemented. Details dependencies between capability management and the Defense Acquisition System process

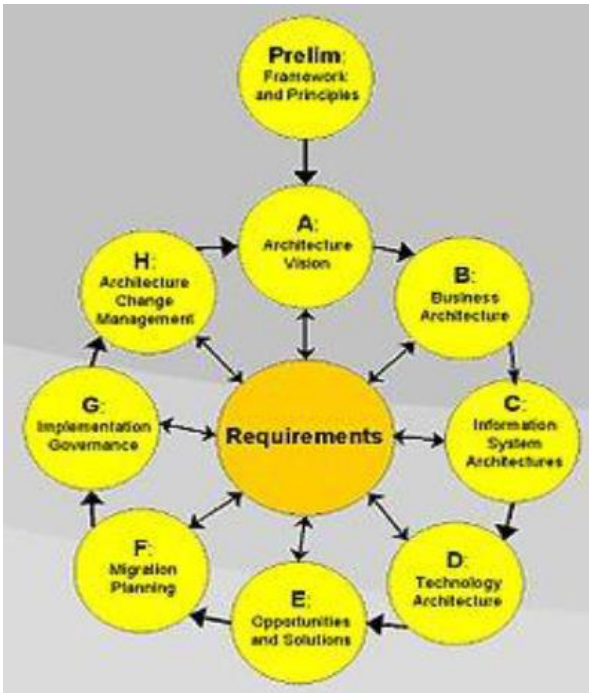
MODAF – Ministry of Defense Architecture Framework

- É uma framework de arquitetura que define um padrão de desenvolvimento de arquiteturas, estabelecido originalmente pelo Ministério da Defesa do Reino Unido.
- Fornece um conjunto de templates (designados por "Views"), que oferecem uma notação padrão para a captação de informação acerca de um negócio, de forma a identificar formas de o melhorar.
- Cada view, oferece uma perspetiva diferente do negócio para suportar os interesses dos Stakeholders.



TOGAF – The Open Group Architecture Framework

- É uma framework que oferece uma arquitetura de alto nível para o desenvolvimento de software empresarial.
- O seu propósito é ajudar a organizar o processo de desenvolvimento através de uma abordagem sistemática direcionada a reduzir erros, respeitar prazos, não exceder o orçamento, e alinhar as Tecnologias de Informação com as unidades de negócio para produzir resultados de qualidade.
- É modelada em quatro domínios:
    1. Negócios
    2. Aplicação
    3. Dados
    4. Tecnologia



Zackman's framework

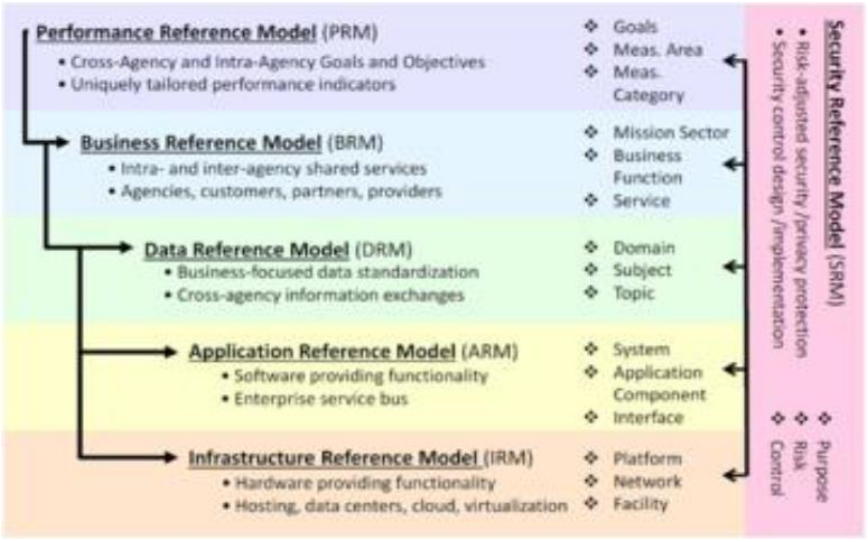
- É uma ontologia empresarial, e uma estrutura para arquitetura empresarial, que oferece uma maneira estrutural e formal de visualizar e definir uma empresa.
- Consiste num esquema bidimensional de classificação que reflete a interseção entre O quê, Como, Quando, Onde e Porquê, com Identificação, Representação, Especificação, Configuração e Instanciação.
- Não é uma metodologia, e não especifica nenhum método ou processo para recolher gerir ou usar a informação que descreve.
- O seu principal objetivo é organizar e analisar dados.

	Why	How	What	Who	Where	When
Contextual	Goal List	Process List	Material List	Organisational Unit & Role List	Geographical Locations List	Event List
Conceptual	Goal Relationship	Process Model	Entity Relationship Model	Organisational Unit & Role Relationship Model	Locations Model	Event Model
Logical	Rules Diagram	Process Diagram	Data Model Diagram	Role Relationship Diagram	Locations Diagram	Event Diagram
Physical	Rules Specification	Process Function Specification	Data Entity Specification	Role Specification	Location Specification	Event Specification
Detailed	Rules Details	Process Details	Data Details	Role Details	Location Details	Event Details

	DATA *What*	FUNCTION *How*	NETWORK *Where*	PEOPLE *Who*	TIME *When*	MOTIVATION *Why*
Objective/Scope (contextual) *Role: Planner*	List of things important in the business	List of Business Processes	List of Business Locations	List of important Organizations	List of Events	List of Business Goal & Strategies
Enterprise Model (conceptual) *Role: Owner*	Conceptual Data/ Object Model	Business Process Model	Business Logistics System	Work Flow Model	Master Schedule	Business Plan
System Model (logical) *Role:Designer*	Logical Data Model	System Architecture Model	Distributed Systems Architecture	Human Interface Architecture	Processing Structure	Business Rule Model
Technology Model (physical) *Role:Builder*	Physical Data/Class Model	Technology Design Model	Technology Architecture	Presentation Architecture	Control Structure	Rule Design
Detailed Reprentation (out of context) *Role: Programmer*	Data Definition	Program	Network Architecture	Security Architecture	Timing Definition	Rule Speculation
Functioning Enterprise *Role: User*	Usable Data	Working Function	Usable Network	Functioning Organization	Implemented Schedule	Working Strategy

Federal Enterprise Architecture

- É a framework de referência de arquitetura empresarial do governo federal nos Estados Unidos.
- Oferece uma abordagem comum para a integração de estratégia, negócio e gestão de tecnologia como parte do desenho e melhoria de performance de uma organização.
- Segue uma metodologia de planeamento colaborativo, que é um processo simples e repetitivo que consiste de uma análise integrada e multi-disciplinar que resulta em recomendações formadas em colaboração entre os elementos da liderança, planeamento, implementação e os Stakeholders.



Anti padrões de Software

- São padrões de projeto de software que são ineficientes e contra/produtivos para o desenvolvimento do projeto.
- Não são apenas simples erros, podem ser más práticas que pode fazer com que o desenvolvimento do projeto no futuro possa ser impraticável e de baixa manutenibilidade, mesmo estando a funcionar no presente.
- São de evitar pois são considerados de padrões escuros ou armadilhas ocorrendo em várias categorias.

Tipos de anti padrões de software

- **Anti padrões organizacionais**
  - Padrões que prejudicam a organização de um projeto.
  - Bleeding Edge – Usar tecnologias recentes instáveis ou não testadas que prejudicam o orçamento e muitas vezes a performance do projeto.
  - Paralisia de análise – Quando se dedica um esforço desproporcional à frase de análise do projeto.
- **Anti padrões de gerência de projeto**
  - Padrões que prejudicam o desenvolvimento do projeto ao nível da sua gestão.
  - Marcha da morte – Quando os desenvolvedores não fornecem feedback da má situação do projeto e não são comunicadas ao Responsável do projeto.
  - Modelo em cascata – Usar metodologias de desenvolvimento inadequadas ao tipo de projeto em questão.
- **Anti padrões de análise**
  - Padrões que acontecem quando é feita uma má análise de um projeto.
  - Apatia do espetador – Quando uma decisão ou análise é feita de forma incorreta e quem se apercebe do erro não o reporta.
- **Anti padrões gerais de design**
  - Padrões que afetam a estrutura do projeto
  - Sistema em chaminé – Sistema montado com grande dificuldade de ser mantido e com componentes mal relacionados.
  - Inflação de interface – Interface tão complexa que é impossível de ser realizada.
- **Anti padrões de programação**
  - Padrões que afetam o código que os programadores realizam.
  - Fé cega – Não verificação de uma funcionalidade na sua totalidade com esperança que funcione à primeira.
  - DRYF (Don’t repeat yourself) – Escrever código que se repete ao longo do projeto.
- **Anti padrões metodológicos**
  - Padrões que afetam a metodologia do projeto.
  - Reinventar a roda – Fazer de raiz uma funcionalidade já feita anteriormente.
  - Ctrl + C, Ctrl + V – Utilizador código existente ao invés de utilizar soluções genéricas.