Engenharia de Software

- Engenharia de Requisitos

- identificação dos interessados
- reconhecimento dos diferentes pontos de vista
- coleta colaborativa de requisitos e especificação
- revisão dos requisitos
- gestão dos requisitos

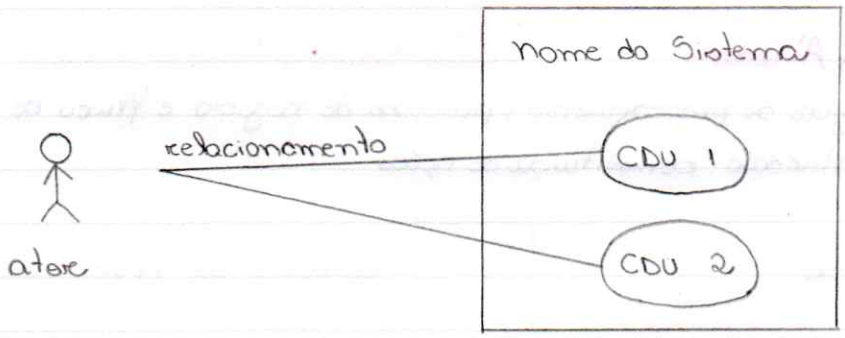
- Casos de Uso (CDUs)

- diz O QUE o sistema deve fazer, não COMO fazer
- verbo no infinitivo
- não pode juntar 3 funcionalidades, CDU é apenas 1 por vez

1) Atores

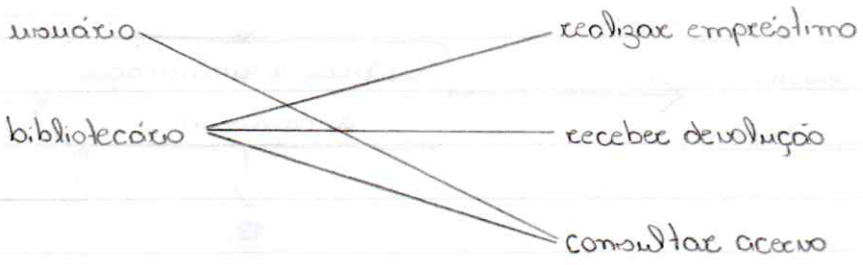
- pessoas, outros sistemas, máquinas...

→ Diagrama de CDUs



- a fronteira do sistema é representada pelo retângulo
- o CDU é representado por uma elipse
- o ator é representado por um boneco

Exercício: CDUs e atores para biblioteca



- pensar pela visão da organização responsável pelo sistema

- Especificação de CDUs:
  - determinar para cada CDU:
    - atores principais
    - atores secundários
    - pré condições
    - pós condições
    - fluxo básico
    - fluxos alternativos (detalhar os tópicos do fluxo básico)
    - requisitos especiais
    - variações tecnológicas e de dados
    - frequência de ocorrência
    - problemas em aberto

exemplo no livro: "Utilizando UML e padrões" pág 94 à 98

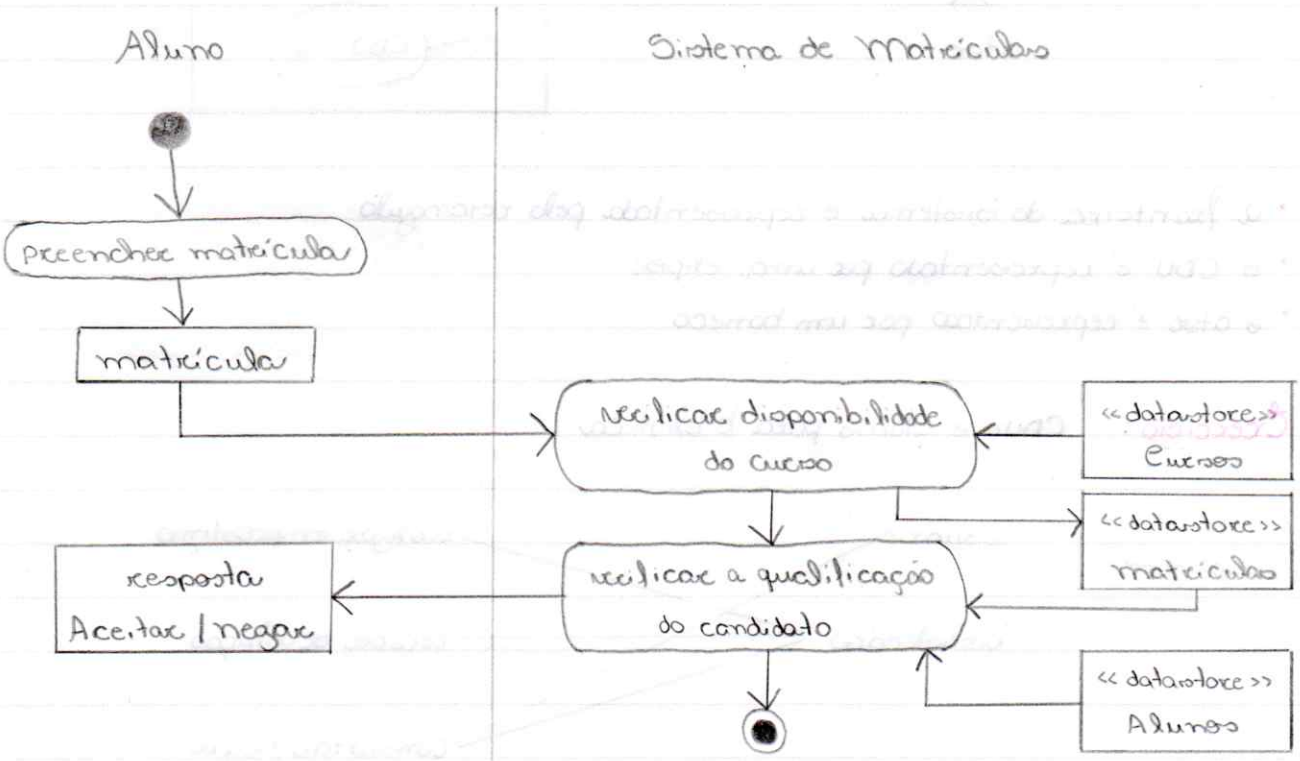
obs: pensar em cada detalhe para a construção dos fluxos, levando em consideração as possíveis ocorrências que fogem da rotina, sempre especificando bem as possíveis resoluções.

- Modelagem de Análise

exemplos: livro do Larman pág 484 à 489

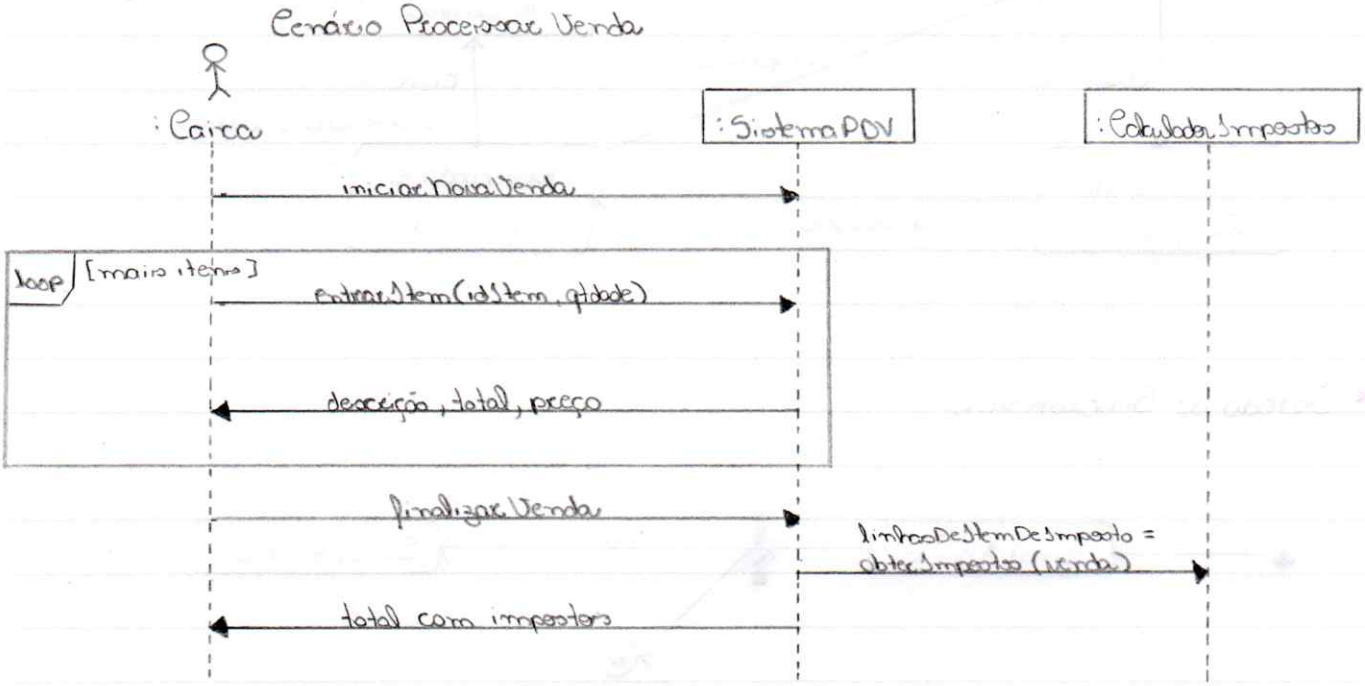
→ Diagrama de Atividades

- descreve a lógica de procedimento, processo de negócio e fluxo de trabalho
- mostra uma atividade, constituída de ações



→ Diagrama de Sequência

- Ilustração do comportamento do sistema
- DSS = diagrama de sequência do sistema
- baseados nos CDUs

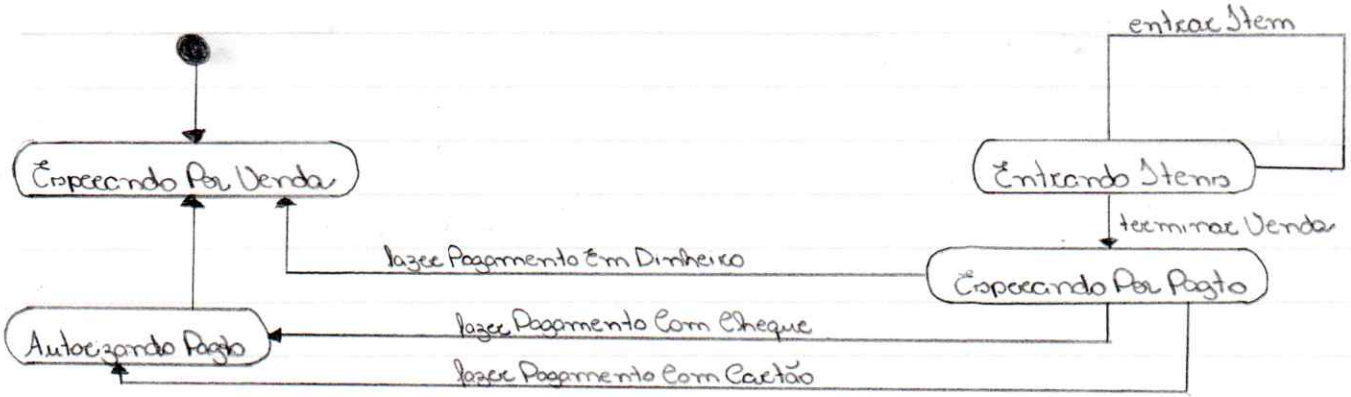


continuação de acordo com a especificação de CDUs

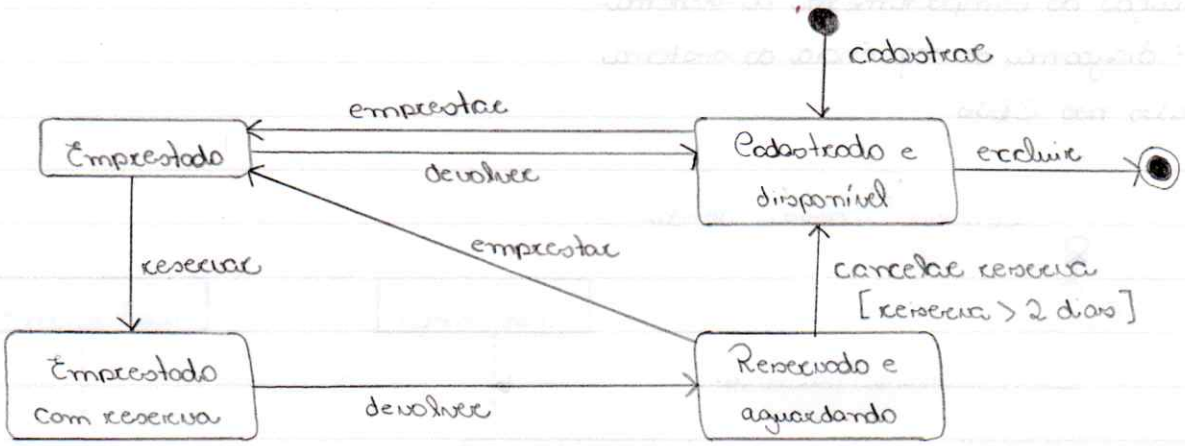
→ Diagrama de Estados

- acompanhar os estados por que passam uma ou mais instâncias de uma determinada classe (exemplo: situação de um documento, como um cheque)
- representar os CDUs
- modelagem de sistemas em tempo real

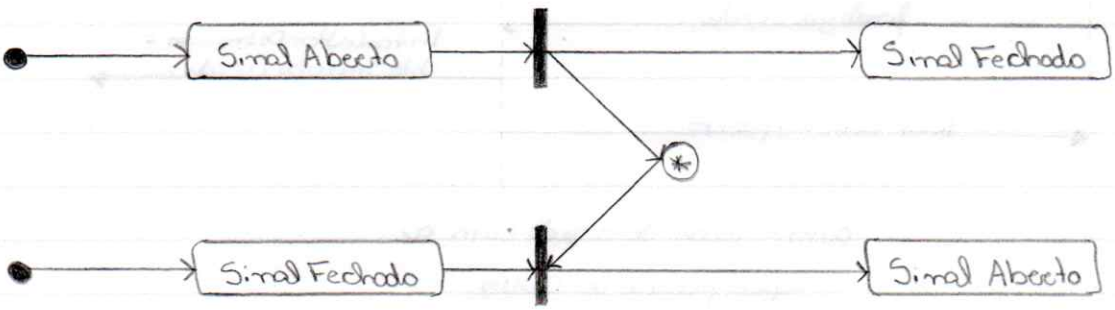
* Aplicação em CDUs



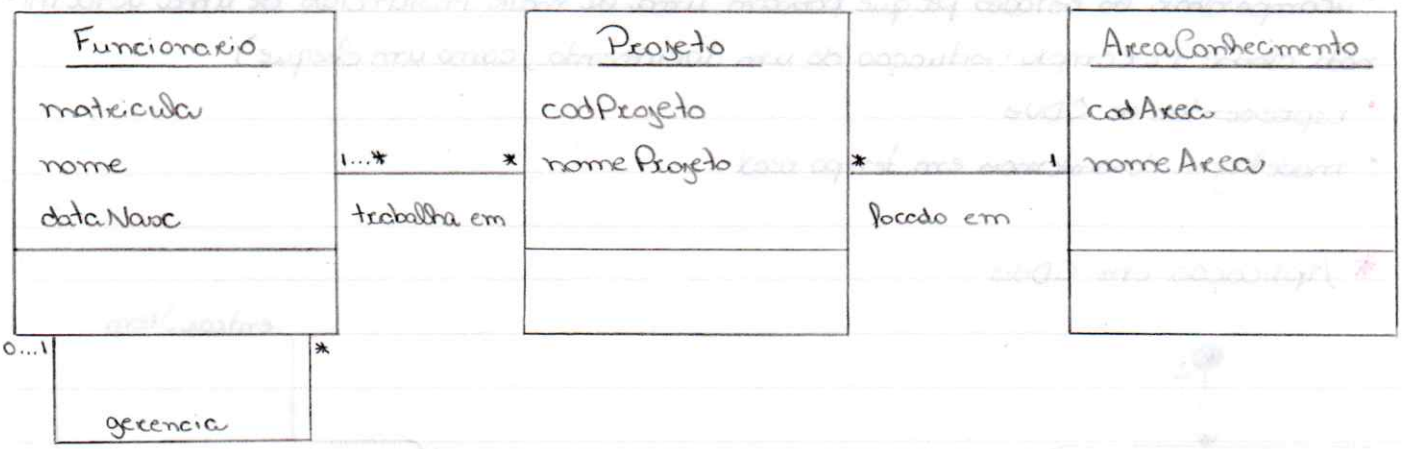
* Aplicação em classe/objetos



* Estado de Sincronismo



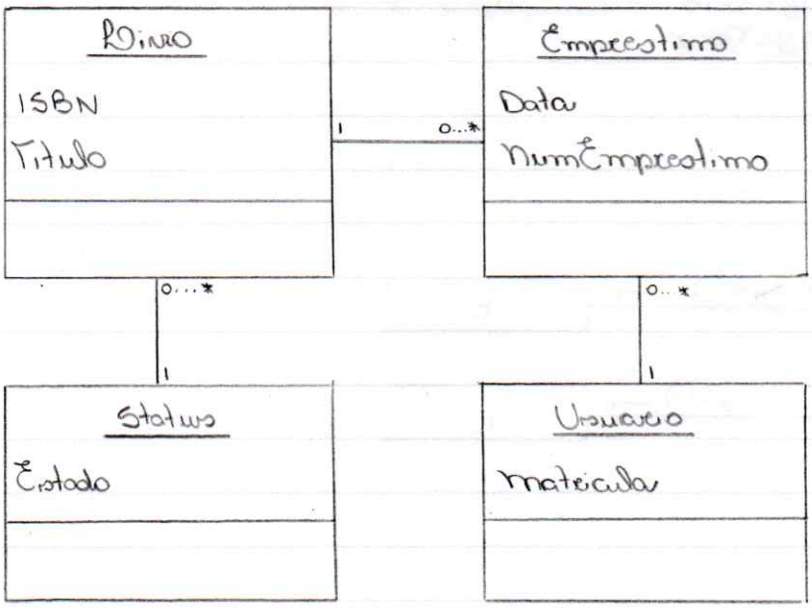
→ Diagrama de Classes Conceitual



* Exemplo Código para Diagrama de Classes

```
Public Class Livro {
        Private integer ISBN;
        Private String Titulo;
        Private Status EstadoLivro;
}

Public Class Status {
        Private String Estado;
}

Public Class Empréstimo {
        Private Date Data;
        Private Integer NumEmpréstimo;
        Private Usuario QuemEmprestou;
        Private Livro LivroEmp;
}

Public Class Usuario {
        Private integer matricula;
}
```

* Diagrama do código acima



obs: a classe Livro tem ligação com a classe Status

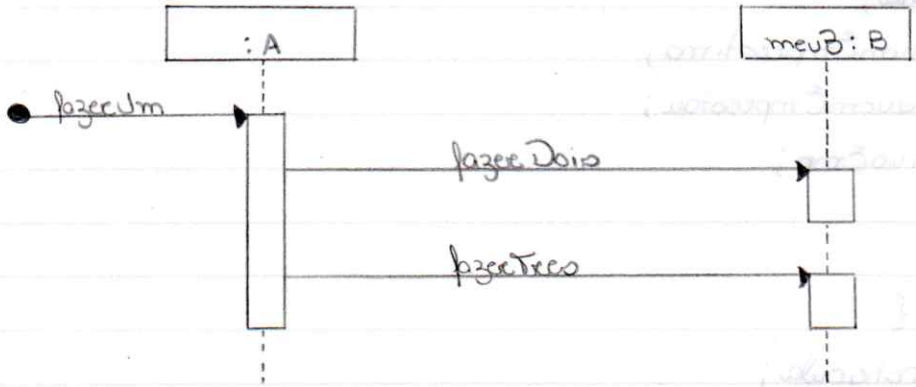
a classe Empréstimo tem ligação com as classes Usuario e Livro

- Projeto e Arquitetura de Software

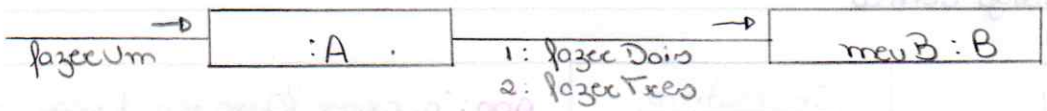
→ Diagramas de Interação

```
public class A {
    private B meuB = new B();
    public void fazerUm() {
        meuB.fazerDois();
        meuB.fazerTres();
    }
}
```

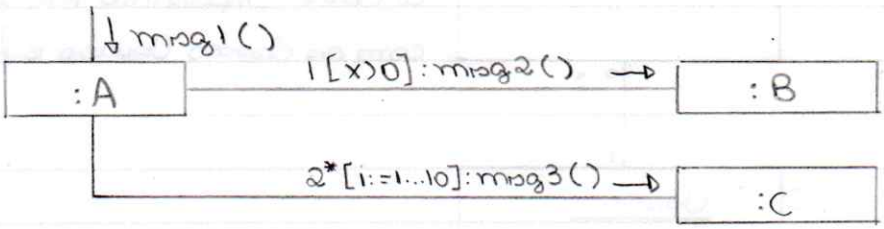
• usando Diagrama de Sequência:



• usando Diagrama de Comunicação:



- Diagrama de Comunicação

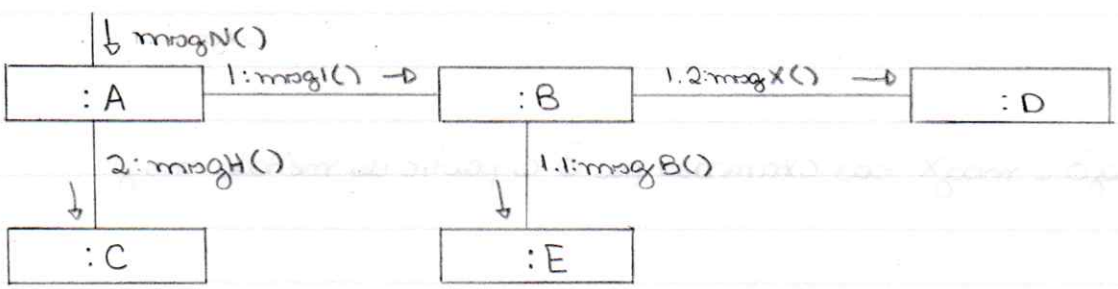


* Código do diagrama anterior

```
public class A {
    public void msg1() {
        if(x>0){
            B.msg2();
        }
        for(i=1; i<=10; i++){
            C.msg3();
        }
    }
}

public class B {
    public void msg2(){

    }
}

public class C {
    public void msg3(){

    }
}
```

obs: msg2 é um método de B
msg3 é um método de C
msg1 é um método de A que, em ordem, envia uma mensagem para B e depois para C

* Exemplo 2: Diagrama de Comunicação

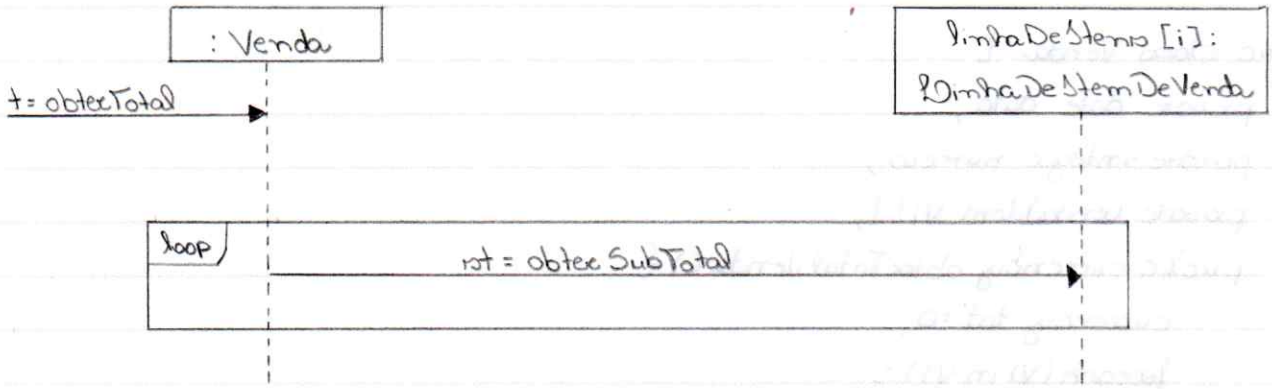


* Código do diagrama anterior:

```
public class A {
    public void msgN() {
        B.msg1();
        C.msgH();
    }
}

public class B {
    public void msg1() {
        E.msgB();
        D.msgX();
    }
}

public class C {
    public void msgH() {

    }
}

public class D {
    public void msgX() {

    }
}

public class E {
    public void msgB() {

    }
}
```

obs: msgB e msgX são chamadas por B a partir do método msg1

Diagrama de Sequência



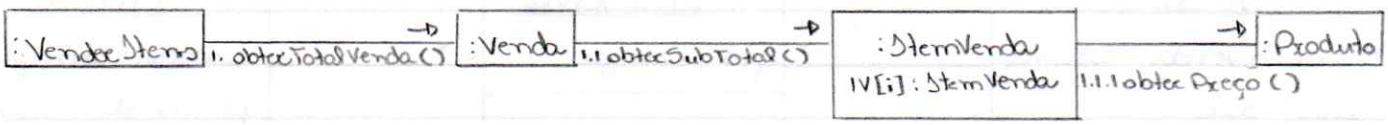
Código do diagrama acima

```
public class Venda {
    private LinhaDeItemDeVenda linhaDeItens[];
    public float obtemTotal() {
        float tot = 0;
        foreach Liv in linhaDeItens {
            tot = tot + Liv.obtemSubTotal();
        }
        return tot;
    }
}

public class LinhaDeItemDeVenda {
    public float obtemSubTotal() {

    }
}
```

Padrões GRASP

Expert (especialista)

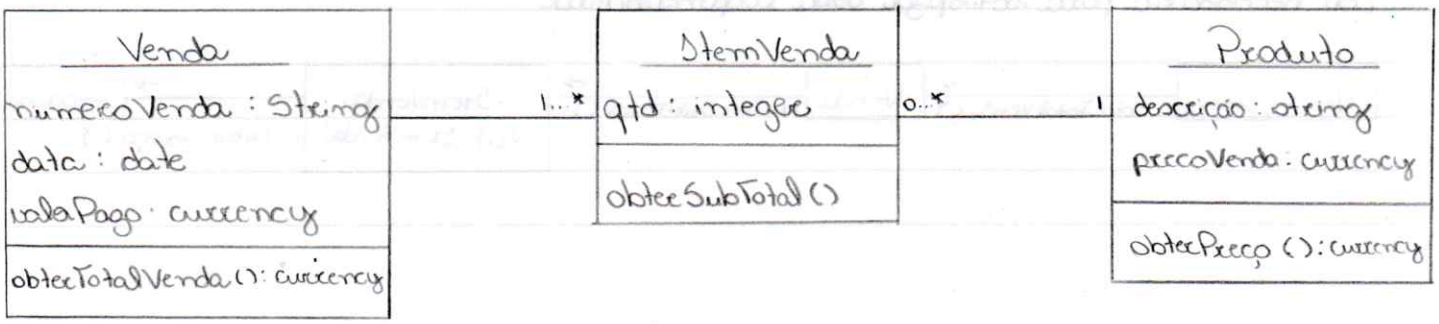
• quem é responsável por obter determinada informação? A classe que tem a informação necessária para satisfazer essa responsabilidade



* Código do diagrama anterior

```
public class Venda {
    private date data;
    private integer numero;
    private vendaItem VI[];
    public currency obterTotalVenda() {
        currency tot = 0;
        foreach (vi in VI) {
            tot = tot + vi.obterSubTotal();
        }
        return tot;
    }
}

public class ItemVenda {
    private integer qtd;
    private Produto prod;
    public currency obterSubTotal() {
        return qtd * prod.obterPreco();
    }
}

public class Produto {
    private string descricao
    private currency precoVenda;
    public obterPreco() {
        return precoVenda;
    }
}
```

* Diagrama de Classes de Projeto



→ Creator (criador)

• quem é responsável pela criação de uma nova instância de uma classe? Uma classe que agregue a outra classe, que usa de maneira muito próxima os objetos da classe criada ou que tenha os dados de inicialização necessários para a criação da nova instância

→ Controller (controlador)

• quem é responsável por tratar eventos do sistema? Uma classe controladora.

obs: criar uma classe controladora para cada CDU

→ Coesão Alta

• as responsabilidades devem ser atribuídas de forma que a coesão seja alta

• classes com coesão baixa são:
  • difíceis de compreender
  • difícil de reutilizar
  • difíceis de manter

baixa coesão, alto acoplamento

obs: pensar no padrão controlador

→ Acoplamento Fraco

• as responsabilidades devem ser atribuídas de forma que o acoplamento seja fraco

• formas de acoplamento:
  • uma classe X tem um atributo que referencia uma instância da classe Y
  • um objeto da classe X chama os serviços de um objeto da classe Y
  • a classe X tem um método que referencia uma instância da classe Y
  • a classe X é uma subclasse da classe Y

- Testes de Software

→ Teste de Unidade

• verificação da menor unidade (módulo, classe, função)

• técnica de caixa branca

• pode ser realizado em paralelo para multiplos módulos

* Teste Caixa Branca

• olhar com detalhe os caminhos lógicos

• problema: o número de caminhos lógicos pode ser muito grande

• solução: testar com um número limitado de opções (os mais impactantes)

• Complexidade Ciclomática
- métrica que proporciona uma medida quantitativa da complexidade lógica
- define o número de caminhos independentes e oferece um limite máximo para o número de testes que deve ser feito para garantir que todas as instruções sejam executadas pelo menos uma vez

• Teste de Condição
- garantir que não tem erros em cada condição do programa:
  - operadores booleanos incorretos / faltando / extras
  - erros de parênteses
  - erros nas variáveis booleanas

• Teste de Fluxo de Dados
- avaliar a inicialização das variáveis e o fluxo das mesmas no programa

• pensar nas condições - limite
  - ideal testar o primeiro e o último
• pensar nas manipulações de erros → fáceis de entender
  - ter mensagens de erros inteligíveis (explicar o erro na mensagem)
  - testar erros em cálculos (ex: divisão por zero) e haver mensagem antes do sistema intervir
• testar estruturas de dados locais
  - testar se as estruturas foram bem carregadas, definidas e organizadas

→ Testes de Integração → deixa de se preocupar com estruturas de código
• descobrir erros associados a interfaces
• entradas e saídas entre módulos devem ser compatibilizar
• testes de caixa preta (não se preocupa com estrutura interna de cada módulo)

* Testes de Caixa Preta
• particionamento de equivalência
  • entradas e saídas são particionadas em "conjuntos de equivalência"

ex: entrada de 5 dígitos entre 10.000 e 99.999, as partições serão:
números < 10.000 ; 10.000 < números < 99.999 ; números > 99.999

- testes baseados em grafos
- testes da matriz ortogonal
  - casos com poucos parâmetros de entrada (poucos valores possíveis)
  - "espalhar" casos de teste uniformemente pelo domínio do teste
  - boa cobertura de testes sem fazer testes exaustivos

- Teste de integração top-down
  - começa com os componentes de alto nível em direção aos de baixo nível

- Teste de integração bottom-up
  - ao contrário do teste top-down

→ Teste de Validação

- foco no nível dos requisitos → fácil usar?
  - funcionais, características comportamentais, desempenho, usabilidade, documentação, compatibilidade, recuperação de erros
    → rede cai, o que acontece?
- testes alfa:
  - usuário participa do teste alfa ("olha sobre o ombro do usuário")

- testes beta
  - usuário usa o programa e reporta os erros para serem consertados

→ Teste de Sistema

- pôr completamente à prova o sistema
  é robusto? aguenta o tranco? suporta determinadas situações?
  volume grande de acesso o sistema cai (não teve teste de sistema)

- teste de recuperação → aparecimento de rede no meio de transação (o q acontece?)
  - força o sistema a falhar e verifica se a recuperação é bem executada
- teste de segurança
  - verifica se a proteção do sistema protege de acessos indevidos
- teste de estresse
  - executa o sistema exigindo quantidades
  - analista tenta destruir o sistema

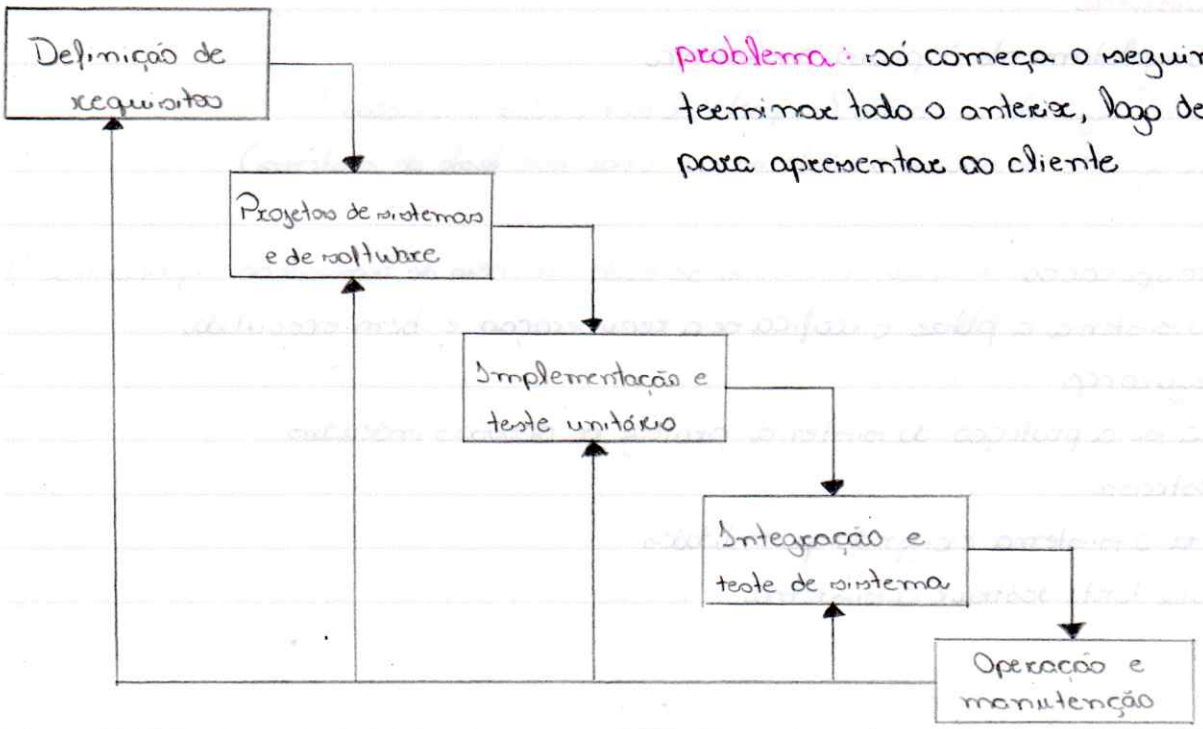
- teste de desempenho
  - testa o desempenho de "runtime" do sistema
    - tempo de execução

Processos de Desenvolvimento de Software

- Arcabouço de processo:
  - comunicação
  - planejamento
  - modelagem (análise e projeto)
  - construção
  - implantação

- Atividades guarda-chuva
  - acompanhamento e controle do projeto
  - gestão de risco
  - garantia de qualidade
  - revisões técnicas formais
  - medição
  - gestão de configuração
  - gestão de reusabilidade
  - preparação e produção do produto do trabalho

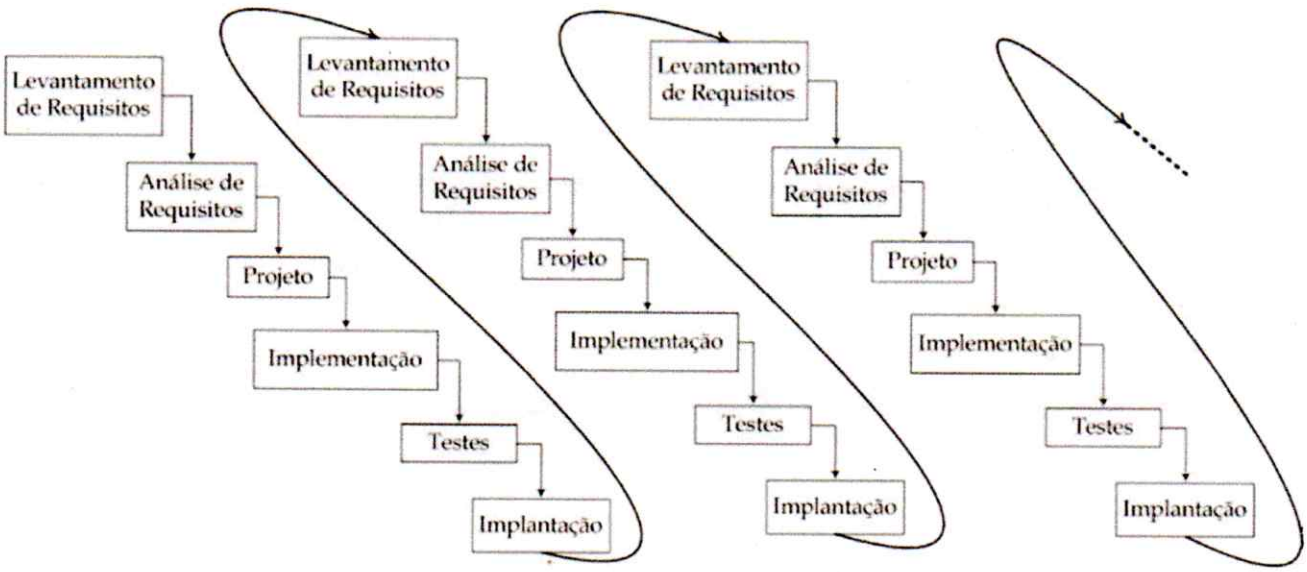
Modelo em Cascata



problema: só começa o seguinte quando termina todo o anterior, logo demora muito para apresentar ao cliente

- é recomendável quando:
  - requisitos estáveis (não sofrem muita alteração)
  - quando o cliente conhece muito bem os requisitos
  - documentar muito bem cada etapa

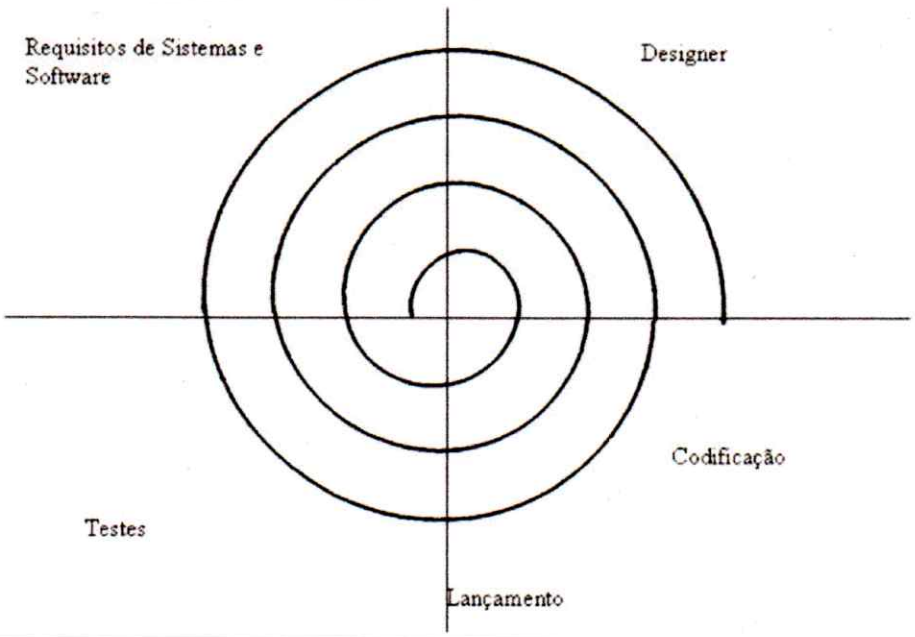
→ Modelo iterativo e incremental



- várias iterações incrementando o resultado na próxima iteração

→ Modelo espiral

- também iterativo e incremental



→ Processo Unificado (PU)

- concepção
- elaboração
- construção
- transição

- feedback mais rápido
- mais focado em redução de riscos
- cliente vê os resultados mais cedo (maior confiança)

* Concepção: → exploração, estudo de viabilidade e planejamento
- visão aproximada, casos de negócio, escopo e estimativas vagas

* Elaboração:
- visão refinada, implementação iterativa da arquitetura central, resolução de altos riscos, identificação da maioria dos requisitos e do escopo e estimativas mais realistas

* Construção:
- implementação iterativa dos elementos restantes de menor risco e mais fáceis e preparação para a implantação

* Transição:
- testes beta e implantação

- Desenvolvimento Ágil
- ênfase na entrega rápida (2 semanas até 2 meses)
- processo se molda à necessidade das pessoas
- software funcionando = medida do progresso (sistema binário: 0 ou 1)
- comunicação com o cliente e aceitação de mudanças
- adaptabilidade
- coesão do time e maturidade

→ Extreme Programming (XP)
- planejamento
  - histórias do usuário
  - valor para histórias

- atribuir custo / história
- custo < 3 semanas
- escolher valor mais alto e/ou maior risco
- após 1ª versão, rever velocidade do projeto

- projeto
  - cartões CRC
  - possível protótipo
  - refabricação

- codificação
  - desenvolve antes testes unitários exercitando histórias
  - programa depois (aos pares)
  - integração do código ao restante

- teste
  - unitários
  - integração ("mede o progresso")
  - aceitação do cliente

- Métricas

- Medida
  - tomada de um único valor
- Medição
  - coleção de uma ou mais medidas
- Métrica (de software)
  - relação entre algumas medidas

→ LOC (linhas de código)

- problemas:
  - comparar as medidas em linguagens diferentes
  - não pode garantir que a quantidade de linhas de código é de maior qualidade

- orientada a tamanho
- vantagem: fácil de contar

→ Pontos por Função

• quantifica a funcionalidade

* Produtividade com PFs

• medida em termo de número de pontos de função entregues por uma pessoa ou grupo

* Cálculo dos PFs

• calcular números de entradas, saídas, consultas, arquivos e interfaces

• multiplicar cada número por um peso, de acordo com a complexidade do sistema e somar o resultado

• problemas:

  • medida indireta, parcialmente subjetiva

  • dificuldade na automação da medida

  • dados não podem ser interpretados fisicamente

• vantagem: independência de linguagem de programação, dados determinados no início

- Qualidade

→ CMMI e MPS.BR

• níveis de maturidade

• qualidade de processos

• CMMI = internacional

• MPS.BR = nacional