Modelação e Análise de Sistemas

Resumos
2013/2014

Bárbara Jael | 73241

MAS

2º ANO
2º SEMESTRE

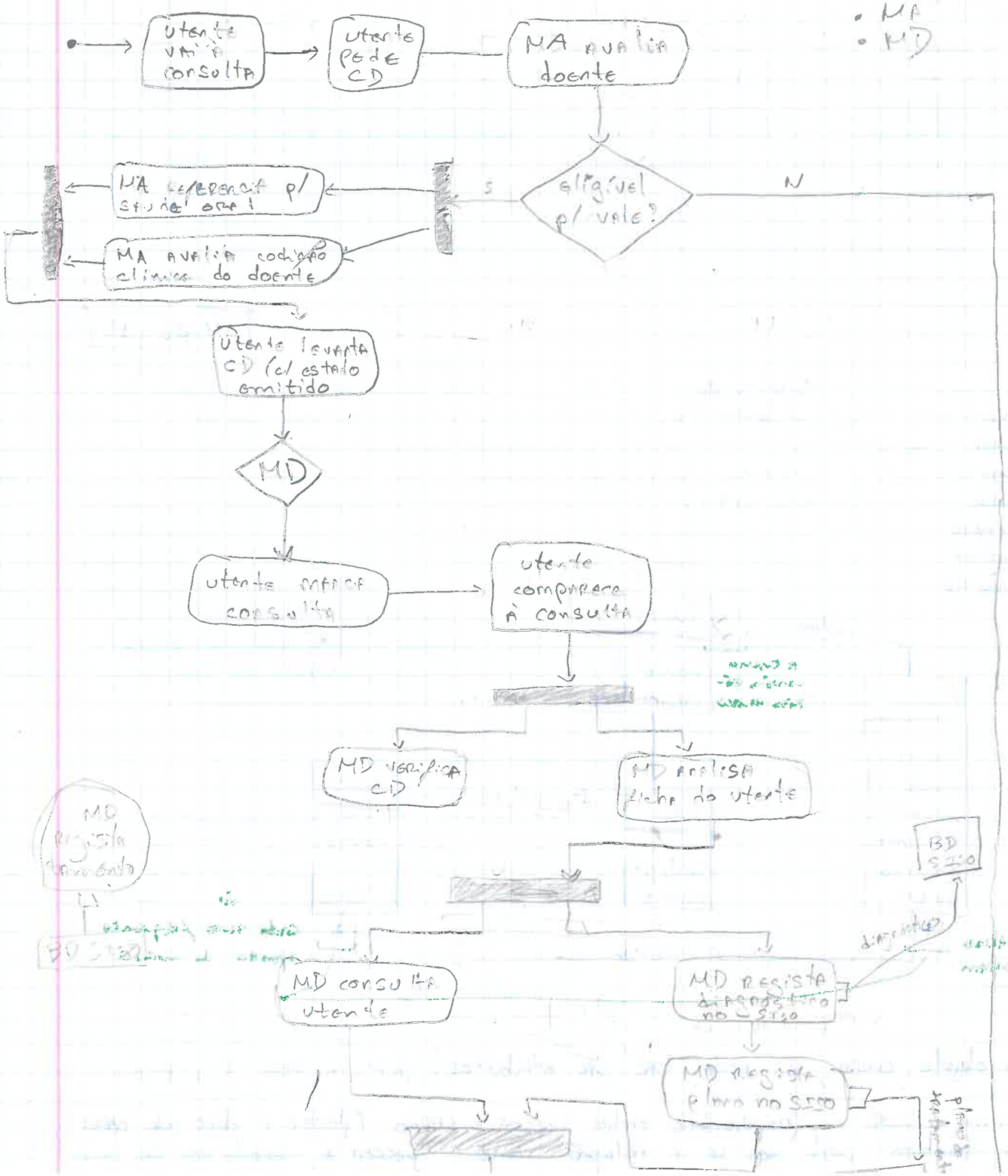
P

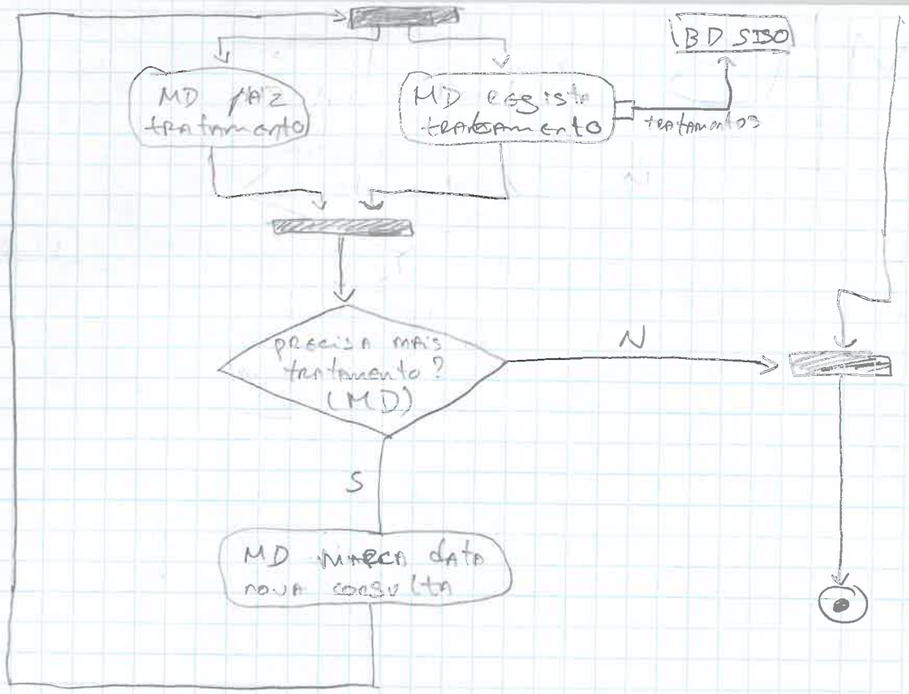
19.02.15

P.1.1

b)

3 participantes
- utente
- MA
- MD



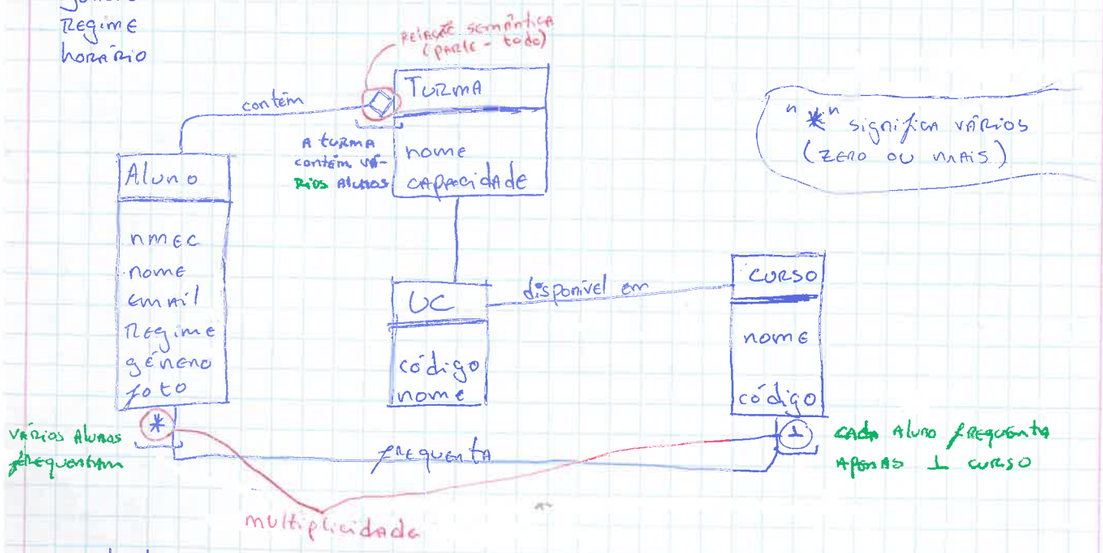


24.02.15

email
nmec
nome
curso
género
regime
horário

Instituição
cadeira (UC)

turma
ano letivo



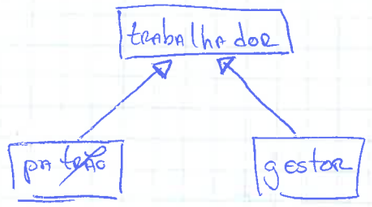
num objeto CARRO, as RODAS não são atributos! mas a cor é, p.e.

há uma relação de propriedade entre PESSOA e CARRO (PESSOA é dono do CARRO), mas também pode ver-se a relação como "PESSOA é condutor do CARRO"

Algumas ações do carro são: ligar, desligar, acelerar

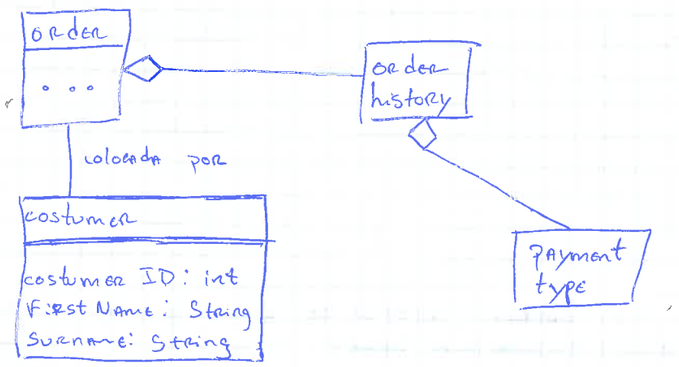
um ligeiro é uma especificação de um CARRO e um aluno é uma especificação de PESSOA

especificações herdam as características da super classe, mas não acontece no sentido oposto



patrão e gestor são especificações de trabalhador

a relação é apenas de A p/ B, de B p/ A não me interessa

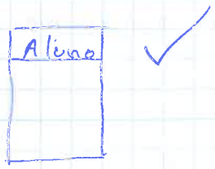


chave de toda modelo de tratamentos utente

} diagrama de classes

03.03.15

Classes



✓

✗ → uma categoria é sempre singular

✗ → uma categoria é sempre uma ~~entidade~~ (isto é um verbo, uma ação, nunca seria uma entidade)

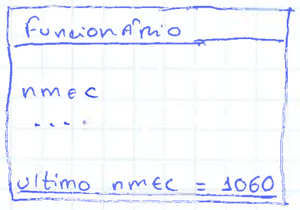
✗ → (isto é uma consequência de uma ~~ação~~, ~~não~~ é uma ~~entidade~~)

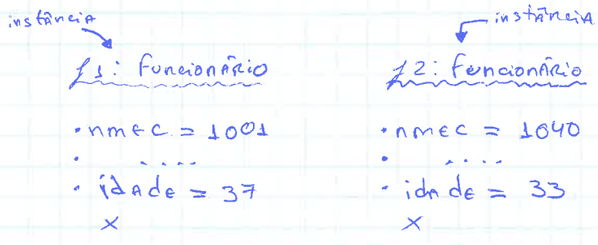
✓

/atributo = atributo derivado

↳ ~~não~~ se escreve a barra; marca-se como um atributo derivado e o programa insere a barra

↳ estes atributos ~~não~~ aparecem na base de dados



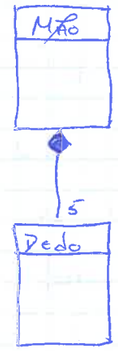


atributo = atributo de classe ← neste caso é o "último nmec"

↳ é relativo à classe, mas ~~não~~ entre p/ cada instância da classe

Agregação

Composição



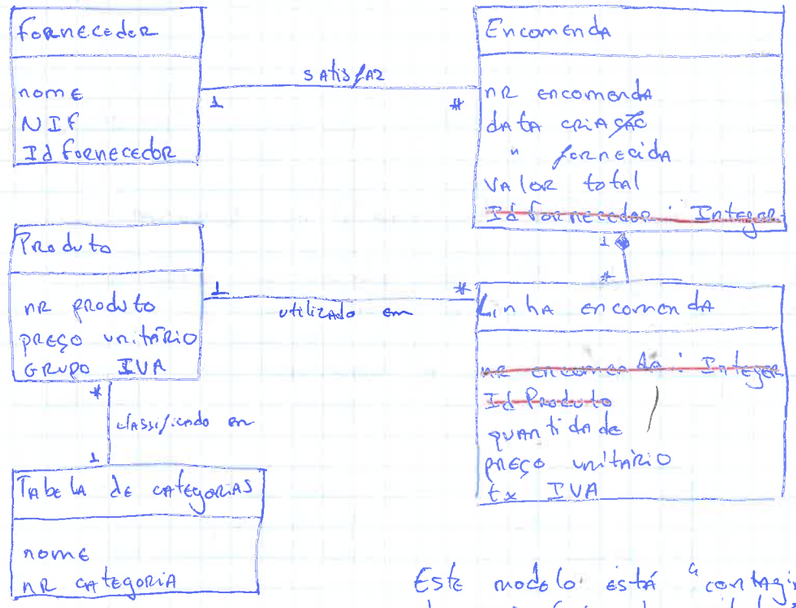
Agregação → instâncias partilháveis

(ex) • um Aluno pode pertencer a mais que uma turma
• se a turma for apagada, o aluno continua a existir

Composição → instâncias ~~não~~ partilháveis

(ex) • um ficheiro (em si e ~~não~~ cópias) pertence apenas a uma pasta
• se a pasta for apagada, os seus ficheiros ~~são~~ também apagados

(ex) cada ~~mão~~ tem 5 dedos mas ~~não~~ se pode andar a trocar os dedos de cada ~~mão~~ p/ a outra



Este modelo está "contagiado" pelo raciocínio de modelo ~~não~~ de Base de Dados — é necessário retirar coisas

10.03.15

Moodle — CAU

Aluno (motivações p/ consultar moodle)

- entregar trabalho
- consultar material das aulas
- ler fórum
- ver se as notas já saíram ...

Professor (motivações também)

- lançar notas
- disponibilizar material pedagógico ...

CAU → perspetiva do utilizador
→ uma motivação p/ usar o sistema
→ verbo
→ caixa-fechada

17.03.15

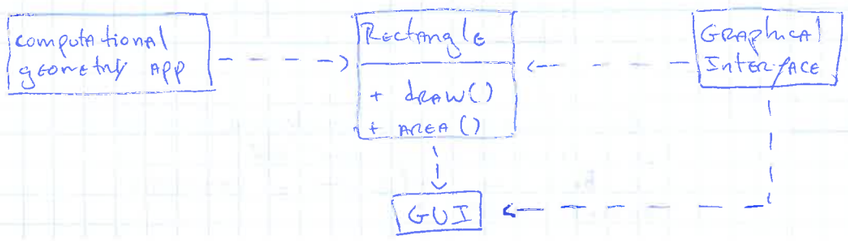
Descrição / propósito
Fluxo típico
Sequências alternativas
} em templates de casos de estudo

07.04.15

coupling → quando uma classe precisa de outra(s) p/ funcionar

se uma classe depende de 20 outras classes p/ funcionar, o coupling é muito alto (o que não é nada bom!)

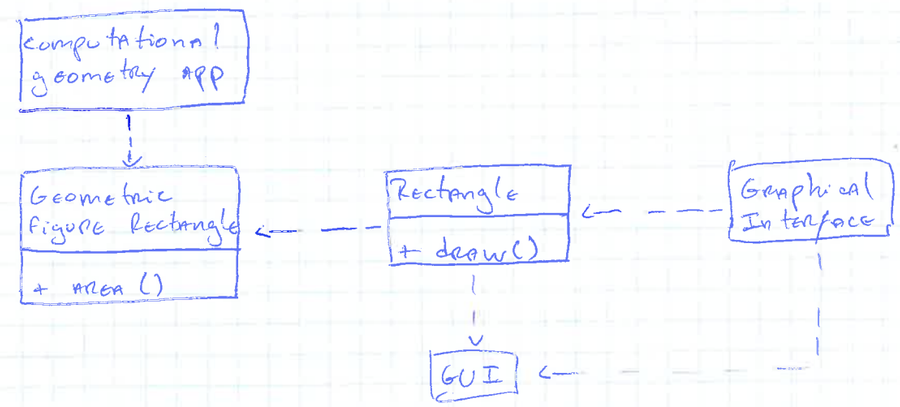
o coupling mais forte de todos é a hierarquia



não há coesão já as duas responsabilidades do retângulo (desenhar-se e calcular a área) não funcionam em os sistemas em que está aplicado

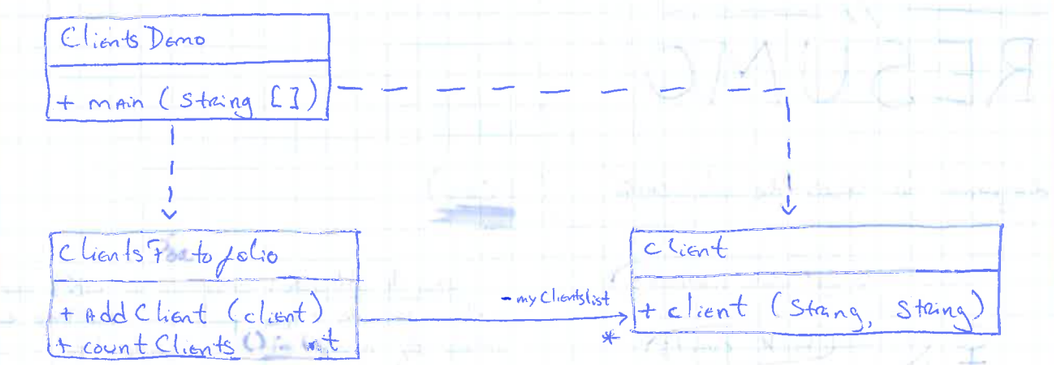
(ex) numa aplicação de computador sem qualquer ambiente gráfico, o retângulo nunca vai conseguir desenhar-se

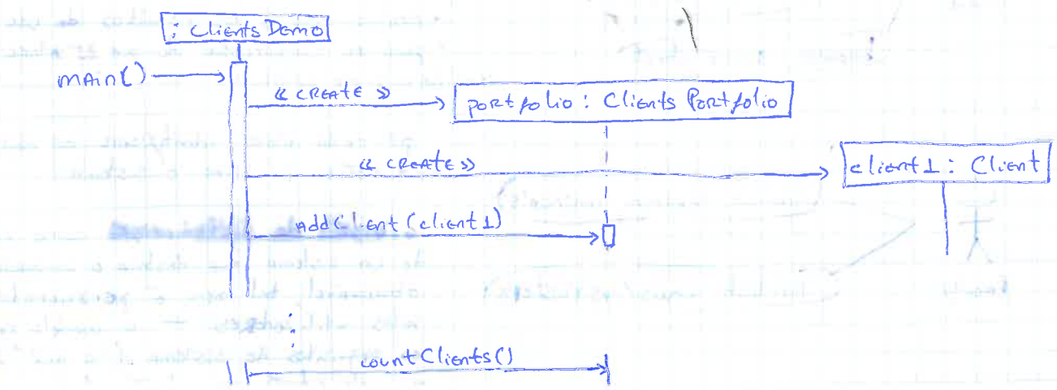
Resolução do problema:



passou a ser coeso (apesar de o coupling ter aumentado)

conversão do código do slide 21 p/ UML:





14.04.15

ificação vs validação

(Ex) um cientista quer um meio de transporte não poluente → constrói-se uma bicicleta

verificação: a implementação está bem feita, ou seja, os pedais estão bem postos, a bicicleta trava quando é preciso, ...

validação: não fizemos levantamento de requisitos suficiente afinal o cientista quer um meio de transporte que seja subaquático

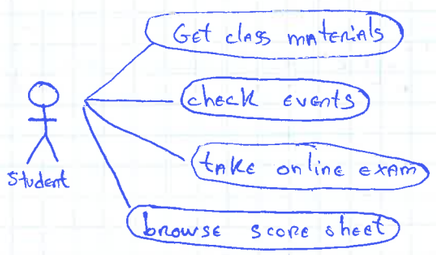
21.04.15

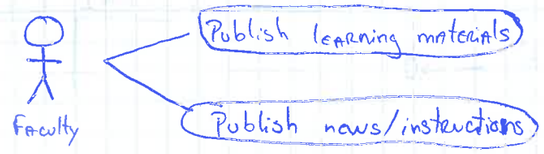
requisitos { funcionais
não funcionais (qualidades, atributos)

15.5 → 17

RESUMO

diagrama de casos de utilização (DCaU)





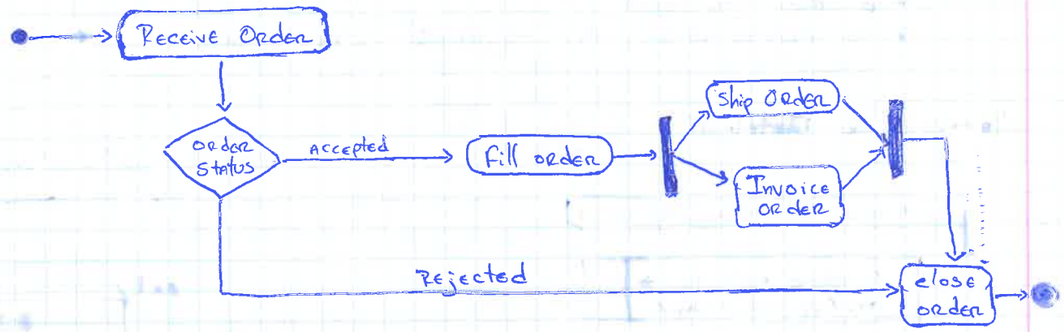
caso de utilização: sequência de ações que um sistema executa e que produzem resultado c/ valor p/ algum ator em particular

implica na análise de requisitos:
- foco no user e nos episódios de uso
- foco na compreensão do que os atores consideram de valor

p/ cada ator, identificar os objetivos/motivações p/ usar o sistema

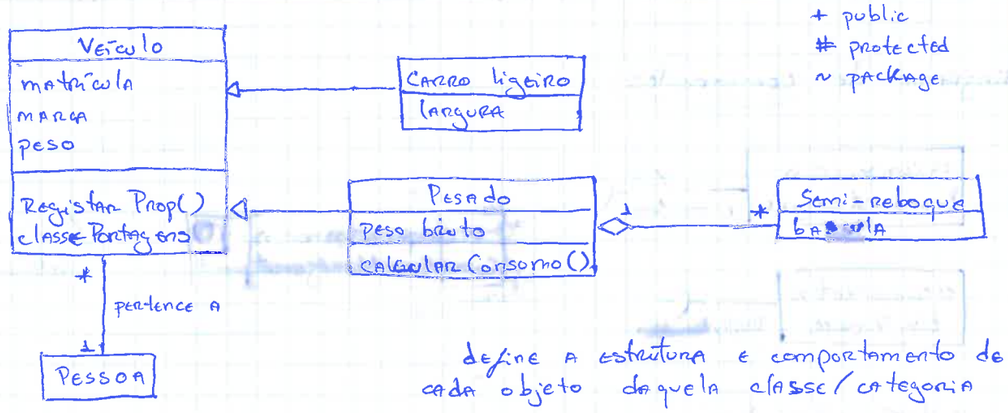
modelo do CaU: como vista de um sistema que destaca o comportamento observável, tal como é percebido pelos utilizadores — o modelo capta os requisitos do sistema ("o quê"), não a implementação da solução ("o como")

diagrama de atividades (DAtiv.)



Quando aplicar?
- modelar fluxos de trabalho / processos de negócio
- descrever algoritmo complexo
- descrever a sequência de interações entre atores e o sistema sob especificação, num caso de utilização
- descrever processos organizacionais existentes/novos
  - neutro em relação à programação
  - bom a captar papéis
  - pode captar fluxo de dados

diagramas de classes (DC)



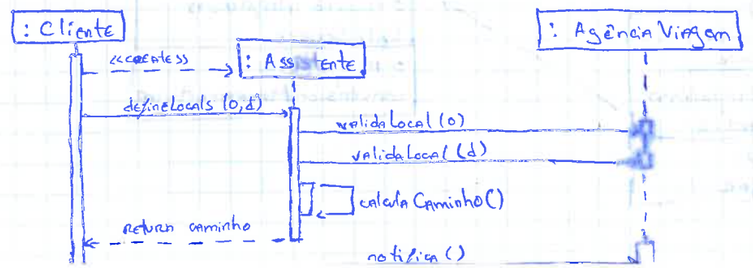
- privado
+ público
# protegido
~ package

define a estrutura e comportamento de cada objeto daquela classe/categoria

funciona como um molde p/ criar objetos

um objeto tem/um estado (interno) — atributos
- comportamento (funcionalidades) + funções/métodos

diagramas de sequência (DSeq)



Ao desenhar um diagrama de interação, estamos a atribuir responsabilidades

diagramas de interação:
- diagramas de sequência (formato achatado)
- diagrama de comunicação (formato grafo)

mostram, p/ um cenário de um CAU:
- os eventos que os atores externos geram
- a ordem temporal
- as necessidades de integração entre sistemas

diagrama de instalação (D Inst.)

mostrar o setup p/ produção

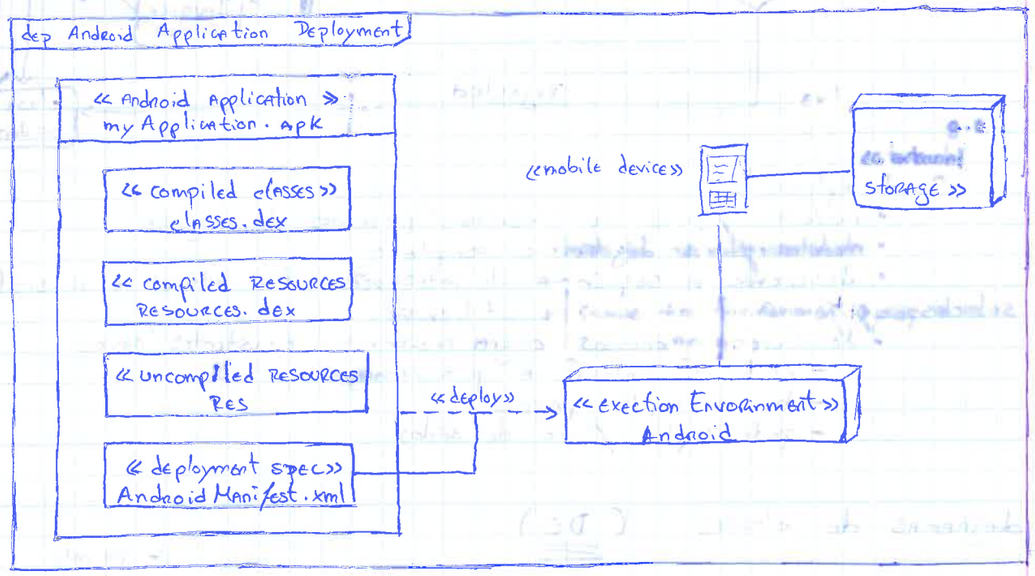
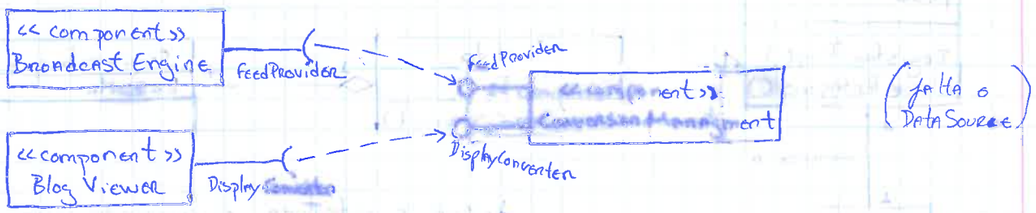
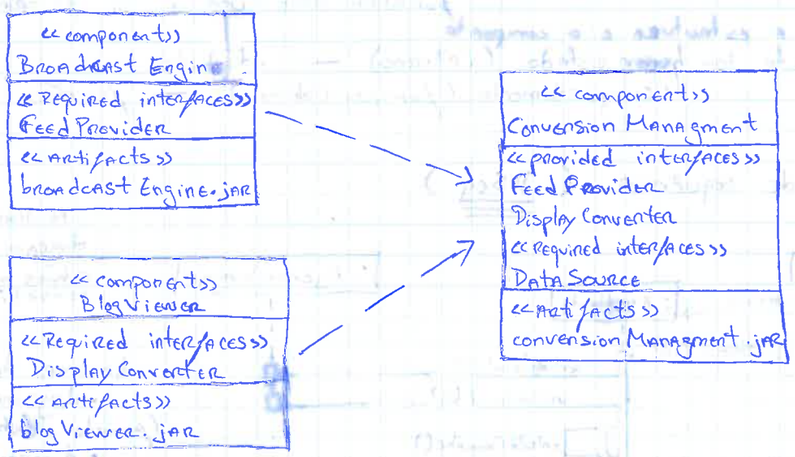


diagrama de componentes



(falta o DataSource)

notação alternativa:



divide sistemas complexos em subsistemas mais geríveis

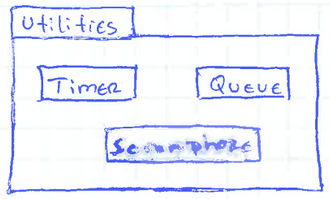
componentes peça tangível da solução (ex ficheiro, arquivo)

↖ peça substituível, reusável de um sistema maior (detalhes de implementação são abstraídos)

A funcionalidade de um componente é descrita por um conjunto de interfaces fornecidos
para além de implementar, o componente pode requerer funcionalidades de outros

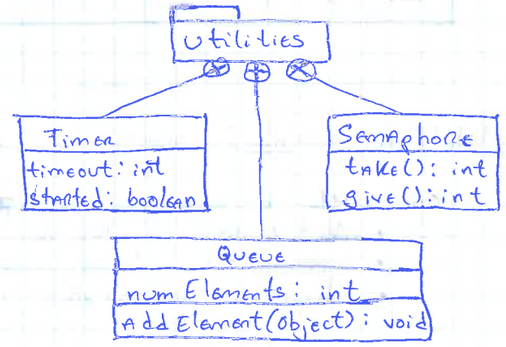
Arquiteturas de baixo "coupling"

diagrama de pacotes



exemplifica blocos e dependências (de uso)

ou



Diagramas de

- classes
- componentes
- pacotes
- instalação
  - (estrutura composta
  - objeto)

} diagramas de estrutura

- Atividade
- caso de uso
- interação — sequência
  - comunicação
  - temporal
  - visão geral)
- máquina de estados

} diagramas de comportamento

diagrama de estados = máquina de estados

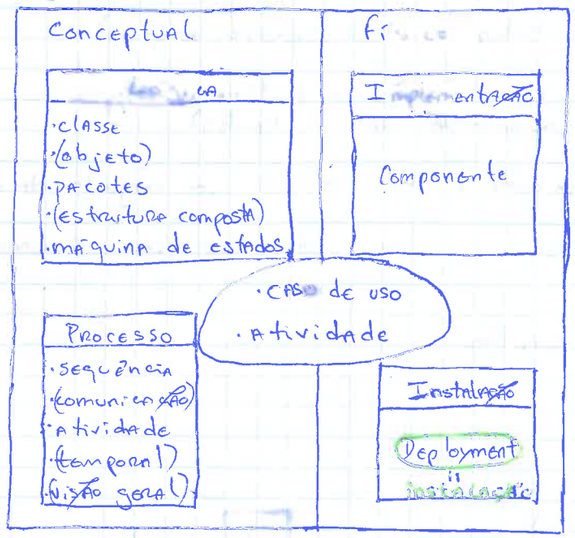
Agregação:

agregação "fraca", pois as partes existem sem o todo

Composição:

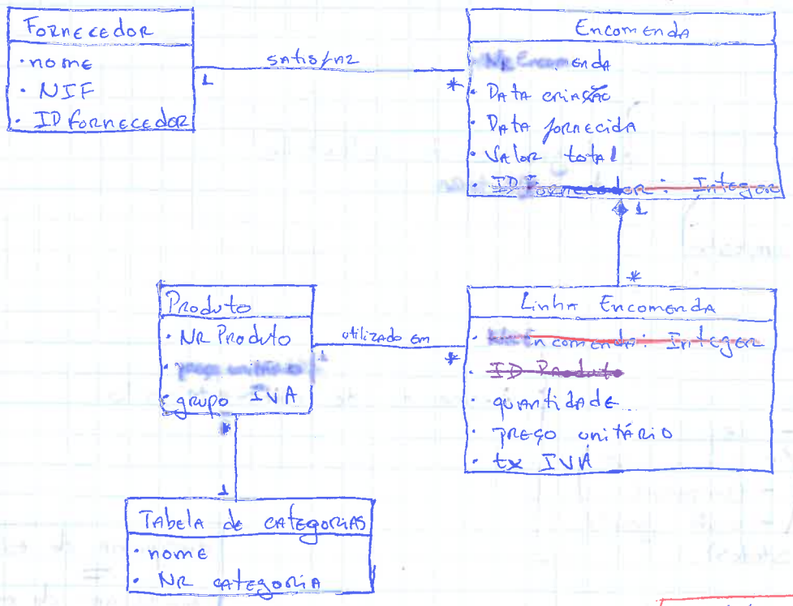
agregação "forte", pois as partes não existem sem o todo

Modelo "4+1" (visões)



Modelo do domínio: mostra os conceitos de um problema

Em UML é representado c/ diagrama de classes
- identificar objetos do problema
- representar classes num DC
- adicionar associações e atributos



modelo contaminado por raciocínio de modelação de BD

Requisitos funcionais

- captam o comportamento pretendido do sistema (serviços, funções ou tarefas que o sistema deve realizar)
  - podem ser captados nos CaUs
  - " descritos c/ diagramas de comportamento: atividades, sequência

Requisitos não funcionais

- Restrições globais num sistema de software (ex robustez, portabilidade ...)
  - por regra, não afetam apenas um módulo/CaU
  - também designados como atributos de qualidade

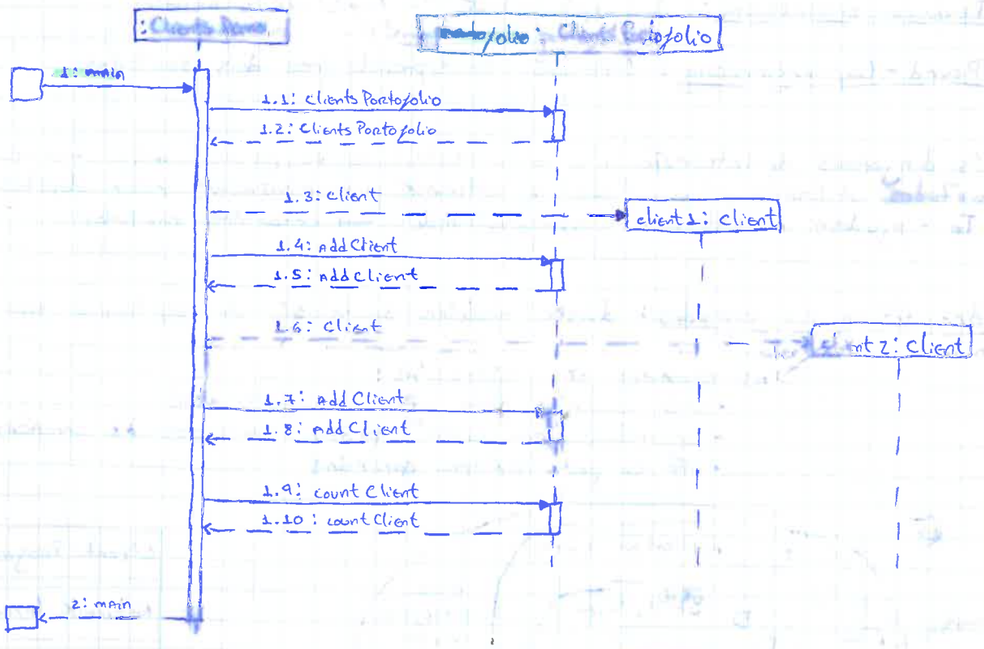
do código p/ o diagrama

①
```
public class ClientsDemo {
    public static void main (String[] args) {
        Clients Portofolio portofolio = new Clients Portofolio();

        Client client1 = new Client ("C101", "Logistica Tal");
        portofolio.AddClient (client1);

        Client client2 = new Client ("C104", "Jose & Maria Lda");
        portofolio.AddClient (client2);

        System.out.println ("Clients count: " + portofolio.countClients());
    }
}
```

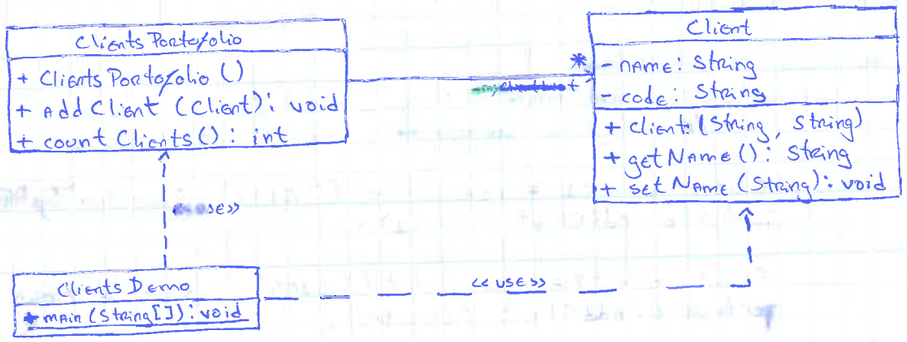


② public class ClientsPortofolio {
    private ArrayList<Client> myClientsList;

    public ClientsPortofolio() {
        myClientsList = new ArrayList<>();
    }

    public void addClient (Client newClient) {
        this.myClientsList.add (newClient);
    }

    public int countClients () {
        Return this.myClientsList.size();
    }
}



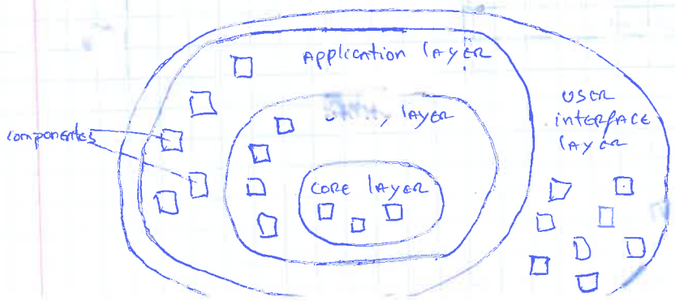
Foward engineering: do modelo p/ a implementação (código)

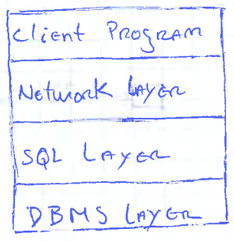
Reverse engineering: da implementação (código) p/ o modelo

Round-trip engineering: transição transparente nos dois sentidos

Os diagramas de interação
- são utilizados p/ visualizar a interação via mensagens entre objetos
- ajudam a distribuir responsabilidades → encontrar métodos

Arquitetura por camadas: divisão modular da solução de software em camadas/níveis → as camadas são sobrepostas:
- cada camada tem uma especialização
- camadas "em cima" pedem serviços às camadas "de baixo"
- não se pode saltar camadas





Arquitetura lógica: organização geral da solução em blocos (packages)
↳ os packages podem representar agrupamentos muito diferentes
(ex)
- packages num programa em JAVA
- " " modelo UML
- subsistemas / divisões do sistema sob especificação

(diagramas de pacotes)

Arquitetura física: explicita a configuração concreta do sistema no ambiente de produção pretendido

Responde às questões:
- que servidores são necessários?
- onde se instala cada módulo?

Processo de eng. de software: conjunto de atividades e resultados associados que produzem uma peça de software

modelo de processo de ES: descrição abstrata de um processo de ES segundo uma interpretação

propõe:
- atividades a realizar
- produtos " "
- papéis na equipa

(ex)
- modelo em cascata (Waterfall)
- " espiral (spiral)
- eXtreme Programming
- Unified Process

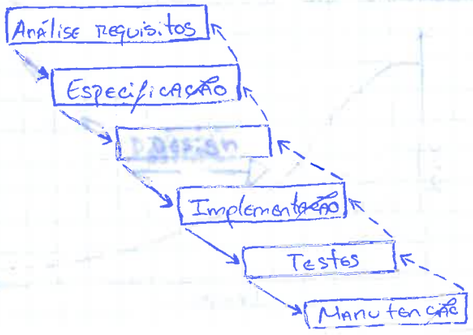
Analista:
- identifica os stakeholders
- define a visão
- detalha os requisitos
- cria e valida a arquitetura
- especifica requisitos suplementares
- responsável pelo glossário e pelo modelo de CAU

Arquiteto: descrever e refinar a arquitetura

stakeholder — (mais papéis à frente)

Waterfall

- o resultado de cada fase é um conjunto de artefactos que deve ser aprovado
- a fase seguinte começa quando a anterior termina
- em caso de erros, é necessário repetir passos anteriores
- encaixa na lógica de outros processos de eng. A. (ex Hw)



vantagens:
- abordagem simples e disciplinada
- milestones são fáceis de marcar (no fim de cada fase)
- progresso linear (de um passo p/ o outro)
- sabe-se o que se segue
- pode ajudar a transferência de conhecimentos (documentos objetivos claros)

problemas:
- confirmação tardia de que os riscos estão controlados
- integração e testes tardios
- "black out" ostensivo
- pressupõe-se que os requisitos são estáveis
- custo de mudança aumenta quase exponencialmente c/ o tempo

Desenvolvimento Ágil

objetivo: desenvolver software rapidamente p/ responder às alterações rápidas dos requisitos

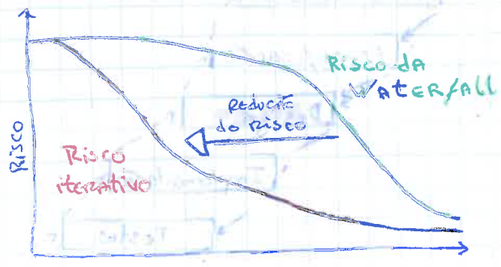
é necessário:
- práticas que equilibrem a disciplina e o feedback
  - ciclos curtos e entrega de valor frequente
  - integração em contínuo
  - desenvolvimento orientado por testes
- aplicar princípios de desenho que favorecem a construção de software flexível e evolutivo

abordagens:
- incremental
- iterativa
- incremental e iterativa

abordagem iterativa → foca a entrega de valor orientada por ciclos curtos

em cada iteração é produzido algum resultado executável (uma parte do produto); depois, o resultado é revisto e melhorado na iteração seguinte; isto repete-se até o produto estar acabado

entregas frequentes e integração de forma contínua reduzem significativamente o risco do projeto, pois os erros são encontrados e tratados conforme vão aparecendo (e não tudo no fim)



Fases do OpenUP

- conceção (inception) [sem requisitos detalhados]

objetivo: conseguir concordância entre todos os stakeholders sobre os objetivos do projeto e se é p/ avançar ou não

milestone de objetivos → neste ponto, examinam-se custos vs. benefícios do projeto e decide-se se é p/ avançar ou cancelar o projeto

- elaboração [sem design detalhado]

objetivo: atenuar riscos técnicos e não-técnicos

↳ riscos técnicos são habitualmente tratados por estabelecer a base de uma arquitetura executável do sistema e proporcionar uma base estável p/ a maioria do esforço de desenvolvimento da fase seguinte

milestone da arquitetura → é
- combinada a base de requisitos
- examinado o detalhe dos objetivos e o âmbito do projeto, a escolha da arquitetura e a resolução dos maiores riscos

↳ este milestone só estará concluído quando a arquitetura for validada

concorda-se c/ a arquitetura executável p/ ser usada no desenvolvimento da aplicação e considera-se que o valor conseguido até este momento e os riscos são aceitáveis?

- construção

objetivo: desenvolver de forma rentável uma versão operacional do sistema que possa ser implatada na comunidade de utilizadores

milestone da capacidade operacional inicial → a este ponto, o produto está pronto p/ ser entregue à equipa de transição. Todas as funcionalidades foram desenvolvidas e os testes iniciais foram completados. Para além do software, foi desenvolvido um manual do utilizador e existe uma descrição da versão atual. O produto está pronto p/ testes como versão beta.

considera-se que a aplicação já está suficientemente perto de ser lançada e que se devia focar o foco principal p/ a equipa que vai equipar, "polir" e assegurar-se do desenvolvimento c/ sucesso

- transição

objetivo: certificar que o software está pronto p/ ser lançado p/ os utilizadores

milestone do lançamento do produto → decidir se os objetivos foram atingidos e se é necessário outro ciclo de desenvolvimento. Este milestone é resultado das reviews e aceitação das entregas do projeto por parte do cliente.

A aplicação está pronta p/ ser lançada?

— (mais papéis aqui)

gerente do projeto:
- planear o projeto
- gerir e planear as iterações
- Responsável pela lista de itens de trabalho e pela lista de riscos
- avaliar resultados

"developer":
- desenhar e implementar solução
- desenvolver um "plano B"
- integrar e criar a construção

"tester":
- criar casos de teste
- implementar e executar testes

—

→ OpenUP é um exemplo de um método ágil, pq

→ é um processo iterativo c/ as iterações distribuídas por 4 fases: concepção, elaboração, construção e transição.

→ cada fase pode ter quantas iterações forem necessárias

→ as iterações podem ter duração variável, dependendo das características do projeto

→ desenvolvimento orientado por casos de utilização

→ os casos de utilização são usados p/:
- identificar o que é p/ fazer
- planear o desenvolvimento (os casos de utilização são priorizados e os prioritários serão os primeiros a ser construídos)
- dar um cenário/contexto p/ guiar o programador
- dar cenários p/ os planos de teste

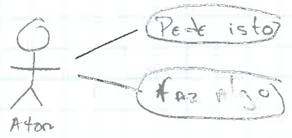
→ os CaU são um resultado central, que liga as várias atividades do desenvolvimento

→ desenvolvimento incremental → o produto é projetado, implementado e testado de forma incremental (um pouco mais é adicionado de cada vez) até que o produto esteja concluído. Isto envolve desenvolvimento e manutenção. Considera-se que o produto está terminado quando satisfaz todos os requisitos.

entrega de valor frequente → uma parte essencial dos métodos iterativos é o contacto frequente c/ o cliente e a demonstração dos incrementos que têm relevância p/ o cliente. Por isso mesmo, o ritmo de entrega/demonstração de incrementos (que geram valor p/ o negócio) é mais intenso que nos métodos sequenciais.

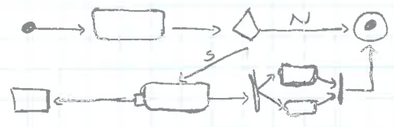
EXERCÍCIOS

diagrama de casos de utilização [comportamento]



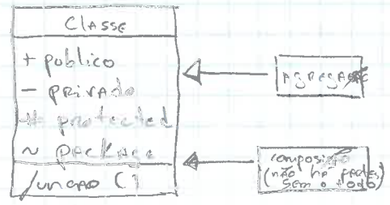
foca-se nas ações de cada Ator na perceção que cada ator tem do sistema

diagrama de atividades [comportamento]



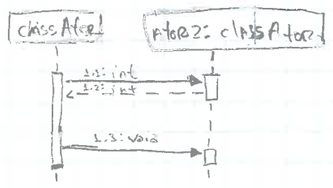
descreve processos operacionais
mostra fluxo do sistema
descreve sequência de interações entre sistema e atores
usado p/ transmitir ideia de código complexo

diagrama de classes [estrutura]



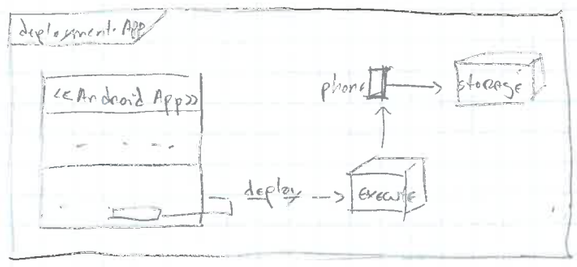
mostra estado (interno)/atributos e comportamento/funções de cada objeto
identifica classes envolvidas no sistema e a ligação/relação entre elas

diagrama de sequência [comportamento]



Atribui responsabilidades → encontrar responsabilidades
/ fluxo temporal da sequência de interações via mensagens

diagrama de instalação [estrutura]

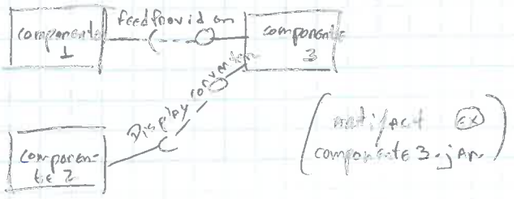


- mostra como é feito o setup da aplicação

diagrama de componentes [estrutura]

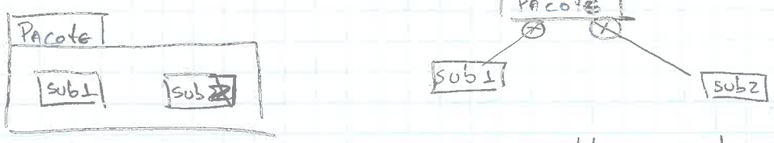
(ex) ficheiro, arquivo

peça substituível e reusável de um sistema maior
implementa uns e pode depender de funcionalidades de outros



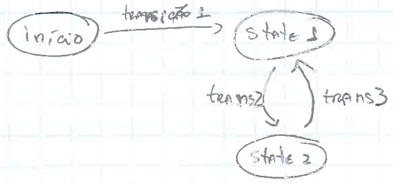
torna sistemas complexos em sub-sistemas mais gerúveis

diagrama de pacotes [estrutura]



exemplifica blocos e dependencias de uso

diagrama de estado = máquina de estados [comportamento



dá informações sobre:
- estado inicial
- transições possíveis
- novos estados que resultam de algum input

Modelo "4+1"

modelo do domínio → mostra os conceitos do problema

feito c/ diagramas de classes (expressa)
- identifica os objetos do problema
- adiciona atributos e associações

Requisitos funcionais vs. não funcionais, (transversais)

captam comportamento pretendido do sistema (ex serviços, funções, tarefas)
Podem ser captados dos CaUs
" " representados c/ diagramas

Restrições globais num sistema de software (ex robustez, portabilidade)
normalmente não afetam apenas um modelo/CaU