PREFÁCIO

Para ensinarmos aos estudantes essa cadeira de Engenharia de Software recorremos a um número importante de livros com várias espicificidades. Na qualidade de docente do módulo Engenharia de Software desde 12 anos respectivamente no curso de Matemática-Informática da Universidade Pedagógica Nacional (Kinshasa, RDC) e no curso de Engenharia Informática do Instituto Politécnico da Universidade Kimpa Vita, reunir os elementos mais pertinentes à compreensão e ao domínio dessa matéria pelos formandos.
Pelo seu carácter profissional, essa cadeira será de um grande apoio a cada pessoa motivada que deseja acrescentar seus conhecimentos no ramo da Engenharia de Software, pela sua qualidade didáctico-pedagógica e a sua clareza.

Esse material de apoio tem o mérito de apresentar claramente e concisão:

- O nascimento e a importância da Engenharia de Software ;
- O contexto objecto no qual nasceu a UML ;
- A modelagem por diagramas UML.

Por fim, cada capítulo é vestido de exercícios ilustrativos corrigidos, que já constituiram para certos controlos de conhecimento aos antigos estudantes.
No sentido de dar ao estudante a ocasião de testar ses conhecomentos adquiridos e verificar sue nível de compreensão, a este material é acrescentado um suporte de exercícios sobre a Concepção Orientada a Objecto e Linguagem de modelagem unificada-UML.
Por fim, propomos neste fascículo uma série de exercícios aos estudantes e leitores desejando confirmar os conhecimentos adquiridos nas aulas teóricas e teórico-práticas e exercer-se em Engenharia de *software*.

As perguntas sobre a teoria são todas aplicadas aos conceitos de Concepção **Orientada a objecto** (C.O.O.) e **Linguagem de modelagem unificada**-**UML**, obviamente objecto, classe, associação, *Agregação e Composição,* herança, modelagem UML.

As perguntas e exercícios propostos são agrupados em função das subdivisões da matéria.

Os exercícios sobre a modelagem UML são ligados aos sistemas de informação a serem modelados por meios de diagramas UML. De modo geral, os exercícios são projectos em vários ramos que podem ser realizados, respeitando o ciclo de vida do *software*. Por ser uma ferramenta pedagógica, para dinamizar o processo de aprendizagem, os estudantes encontrarão neste fascículo propostas de soluções sobre a modelagem de sistemas de informação de alguns pequenos projectos.

ÍNDICE

INTRODUÇÃO GERAL

A Engenharia de Software é uma ciência recente cuja orígem releva aos anos 1970. Naquela época, o aumento da potência material permitiu de realizar *softwares* mais complexos, mas sofrendo novos defeitos : prazos não respeitados, custos de produção e de manutenção elevados, falta de fiabilidade e de performaces. Esta tendência ainda continua nos dias de hoje.

O surgimento da Engenharia de Software é uma resposta aos desfios postos pela complexificação dos softwares e da actividade que visa a produzir-los.

O objectivo da Engenharia de Software é de compreender o processo de desenvolvimento do software, em particular as phases de análise e de concepção orientada a objecto empregando a UML (*Unified Modeling Linguagem*). Este material introduz o ramo da Engenharia de Software para estudantes informáticos novatos nesse ramo. Trata-se de um documento dirigido aos estudantes do curso de licenciatura de Engenharia Informática do ano curricular em que a cadeira está inserido, e pode ser útil a qualquer estudante (Licenciatura, Master ou Estudos de engenheiro) durante a sua formação ou no momento da realização de seu projecto de fim de cíclo, assim como a todo profissional no ramo da concepção e do desenvolvimento informático.

Este material de apoio é organizado em três (3) capítulos chaves, cujo o contéudo se resume da seguinte maneira:

Capítulo 1 : Introdução à Engenharia de Software. Neste capítulo são apresentadas várias razões do surgimento da Engenharia de Software. O capítulo estuda todas as actividades que conduzem de uma necessidade à entrega do software, e cobre principais cíclos de vida de um software.

Capítulo 2 : Introdução a Concepção Orientada Objecto. O objectivo desse capítulo é de se familiarizar com a abordagem orientada objecto no que concerne a concepção. Mais específicamente, ao término do capítulo, o estudante será capaz de entender os conceitos desta abordagem.

Capítulo 3 : Linguagem de modelagem unificada a objectos- UML (*Unified Modeling Language*). Neste capítulo o estudante descubrirá a linguagem UML, que permite modelar de forma simples e pronta a estrutura e o comportamento de um sistema orientado a objecto. UML é apresentado com detalhes necessários possibilitando ao estudante de entender sua utilidade e de poder utiliza-lá para realizar análises ou concepções orientadas objecto.

CAPÍTULO 1: INTRODUÇÃO À ENGENHARIA DE SOFTWARE

1. Introdução

O objectivo principal deste capítulo é de fazer entender ao estudante as causas do surgimento da engenharia de *software* e os principais ciclos de vida de um *software*.

A palavra engenharia de *software* foi introduzido no fim dos anos 60, durante uma conferência realizada para discutir da " crise do *software*".

1.1. Crise do software

Os sintomas mais característicos desta crise são:

- os *softwares* realizados não correspondiam muitas vezes as necessidades dos utilizadores;
- os *softwares* contêm muito mais erros (qualidade do *software* insuficiente);
- os custos de desenvolvimento são raramente previsíveis e são geralmente exagerados;
- a manutenção dos *softwares* é uma tarefa complexa e custosa;
- os prazos de realização são geralmente ultrapassados;
- os *softwares* são raramente portáteis.

São problemas que conduziram à emergência de uma disciplina designada "**a engenharia de software**".

1.2. Definição de Engenharia de Software

A Engenharia de software (*Software Engineering*, em Inglês) é um campo das ciências do engenheiro cuja finalidade é a **concepção**, a **realização** (a **fábrica**) e a **manutenção** dos sistemas informáticos complexos, seguros e de boa qualidade.

Assim, a Engenharia de software define um conjunto de **métodos**, **técnicas** e **ferramentas** que contribuem à produção de um ***software*** de **qualidade** com domínio dos **custos** e **prazos**.

1.3. Objectivo da Engenharia de software

O objectivo da Engenharia de software é de desenvolver nos prazos fixados *softwares* de qualidade. A Engenharia de *software* se preocupa de procedimentos de produção de *softwares* de modo a assegurar que o produto fabricado:

- Mantem-se nos limites financeiros previstos no início: **Custo**;
- Responde às necessidades dos utilizadores: **Funcionalidades**;
- Corresponde ao contrato de serviço inicial: **Qualidade**
- Mantem-se nos limites de tempo previstos no início: **Prazo**

Resumindo, surge a regra do **CFQP**.

1.4. Qualidade exigida de um software

Se a Engenharia de software é a arte de produzir *softwares* seguros, por conseguinte é necessário de afixar os critérios de qualidade de um *software*.

1) **Validade**: é a capacidade do *software* a cumprir suas funções exactamente como definido pelo caderno dos encargos e as especificações.

2) **Fiabilidade ou robustez**: capacidade do *software* para operar em condições anormais.

3) **Extensibilidade** (manutenção): é a facilidade com que o *software* é adequado para a manutenção, ou seja, a uma alteração ou ampliação das funções que são necessários ; facilidade de adaptação do software as mudanças de especificações.

4) **Reutilização** (ou reusabilidade): capacidade de um *software* a ser reciclado, em tudo ou em parte, nas novas aplicações.

5) **Compatibilidade**: é a facilidade com que o *software* pode ser combinado com outros programas.

6) **Eficácia**: utilização óptima dos recursos de *hardware*.

7) **Portabilidade**: é a facilidade com que o *software* pode ser transferido para diferentes ambientes de *hardware* e de *software*.

8) **Verificabilidade**: facilidade de preparação dos procedimentos de ensaio (ou teste).

9) **Integridade**: capacidade do *software* para proteger seu código e os dados contra acesso não autorizado.

10) **Facilidade de uso**: facilidade de aprendizagem, uso, preparação de dados, interpretação de erros e correcção em caso de erro do utilizador.

1.5. Principais actividades do processo de desenvolvimento de um software

Independentemente da abordagem adoptada para desenvolver um *software*, existe um certo número de actividades de base.

1.5.1. Análise das necessidades

O objecto dessa actividade é de evitar desenvolver um *software* não adequado. Logo, vamos estudar o ramo de aplicação, assim como o estado actual do ambiente do futuro sistema a fim de determinar os limites /fronteiras, o papel, os recursos disponíveis e requeridos, os constrangimentos de utilização e de performance, etc.

O resultado dessa actividade é um conjunto de documentos descrevendo os aspectos pertinentes do ambiente do futuro sistema, seu papel e sua futura utilização. As vezes um livro de utilização preliminar é produzido.

1.5.2. A especificação global

Actividade que tem como objecto de estabelecer uma primeira descrição do futuro sistema. Seus dados são os resultados da análise das necessidades assim como das considerações de técnica e de faseamento informática. Seu resultado é uma descrição de aquilo que deve fazer o *software* evitando decisões prematuras de realização (diz-se **O quê**, mas não **O como**). Essa descrição servirá de ponto de partida para o desenvolvimento. Essa actividade é fortemente ligada a análise das necessidades com a qual importantes trocas são necessárias.

1.5.3. Concepção arquitectural e detalhada

Actividade de concepção consiste em enriquecer a descrição do *software* de detalhes de implementação a fim de atingir uma descrição muito próximo de um programa. Desenrola se muitas vezes em duas etapas:

- A concepção arquitectural;
- A concepção detalhada.

A etapa de concepção arquitectural tem como objecto de **decompor** o *software* em **componentes mais simples**.
Especifica-se as interfaces e as funções de cada componente. Com a essa etapa, obtêm-se uma descrição da arquitectura do *software* e um conjunto de especificações de seus diversos componentes.

A etapa de concepção detalhada **fornece para cada componente uma descrição da maneira cujas funções do componente são realizadas**: algoritmos, representação dos dados.

1.5.4. Programação ou Codificação

Esta actividade consiste a passar do resultado da concepção detalhada a um conjunto de programas ou de componentes de programas.

1.5.5. Gestão de configuração e integração

A gestão de configuração tem por objecto permitir a gestão dos componentes do *software*, de dominar a evolução e as actualizações ao longo do processo de desenvolvimento. A integração consiste a reunir todo ou parte dos componentes de um *software* para obter um sistema executável. Essa actividade utiliza a gestão de configuração para unir versões coerentes de cada componente.

1.5.6. Validação e verificação

O problema da adequação dos resultados da análise de necessidades e da especificação geral é delicado.

- A pergunta feita é: foi descrito o "bom" sistema, isto é um sistema que atenda às expectativas dos utilizadores e às restrições de seu ambiente?

  A actividade que se destina a garantir que a resposta a esta pergunta é satisfatória é chamada de **validação**.
- Em contraste, a actividade que consiste em garantir que as descrições sucessivas do *software*, e o próprio *software* atenda à especificação global é chamada de **verificação**.
  A pergunta respondida no caso de verificação é: o desenvolvimento está correto em relação à especificação inicial?

A validação consiste essencialmente em revisões e inspecções de especificações ou manuais e prototipagem rápida.

A verificação também inclui inspecções de especificações ou programas, bem como da prova e do teste.

Uma prova refere-se a uma especificação detalhada ou a um programa e permite provar que esta ou aquela satisfaz bem a especificação inicial.
O teste consiste em procurar erros em uma especificação ou um programa.

1.5.7. Papel de modelagem

Quando as necessidades não são certas, a actividade de validação fica difícil. Para o efeito, adoptamos a solução de maquete (prototipagem rápida), trata-se de desenvolver um programa que é um rascunho do futuro *software* (não tem o desempenho ou todas as funcionalidades de um produto acabado).
Este programa é então submetido a cenários em conjunto com futuros utilizadores para especificar suas necessidades. Isso é chamado de modelo exploratório. A maquete também pode ocorrer durante uma fase de concepção: diferentes escolhas podem ser experimentadas e comparadas por meio de maquetes (neste caso, falamos de maquete experimental).

2. Ciclo de vida do software

O ciclo de vida de um *software* (em Inglês, *software lifecycle*) consiste na sequência das diversas actividades necessárias ao seu desenvolvimento.
O ciclo de vida permite detectar os erros o mais cedo possível e assim controlar a qualidade do produto, os prazos para a sua realização e os custos associados.
Existem vários modelos de ciclo de vida de um *software*.

2.1. Modelo em " Cascata "

O ciclo de vida dito da « cascata » data de 1970, é a obra de Royce.

Princípio do modelo muito simples: convem concordar em ter um certo número de etapas (ou fases). Uma etapa deve terminar em uma determinada data com a produção de determinados documentos ou *softwares*.

Os resultados da etapa são sujeitos a uma revisão aprofundada e a próxima etapa só é realizada quando são considerados satisfatórios.

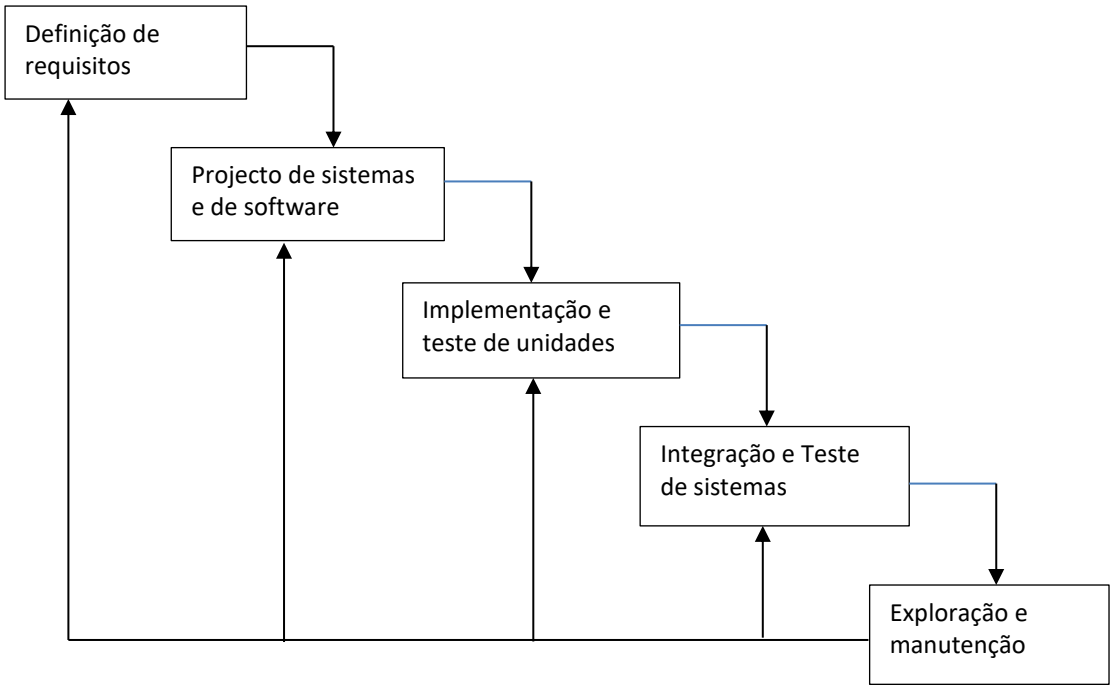


Figura 1- Modelo em cascata

O modelo original não tinha opção de reversão. Isso foi adicionado posteriormente com base no facto de que uma etapa apenas põe em causa a etapa anterior, que, na prática, é insuficiente.
As versões actuais deste modelo mostram validação - verificação em cada etapa. Portanto, encontramos sucessivamente:

- Análise e definição de requisitos (especificação de requisitos)

Nesta fase, define-se as funções, as restrições e os objectivos do sistema. São estabelecidos por meio da consulta aos utilizadores do sistema.
Em seguida, são definidos em detalhes e servem como uma especificação do sistema.

- Projecto de sistemas e de *software*

Fase que agrupa os requisitos em sistemas de *hardware* ou de *software* para estabelecer uma arquitectura do sistema geral.

- Implementação e teste de unidades

Durante essa fase, o projecto de *software* é compreendido como um conjunto de programas ou de unidades de programa.
O teste de unidades envolve verificar que cada unidade atenda a sua especificação.

- Integração e teste de sistemas

As unidades de programa ou programas individuais são integrados e testados como um sistema completo a fim de garantir que os requisitos de *software* foram atendidos. Depois dos testes, o sistema de *software* é entregue ao cliente.

- Exploração e Manutenção

Fase mais longa do ciclo de vida em que o sistema é instalado e colocado em operação. A manutenção envolve corrigir erros que não foram descobertos nas etapas anteriores do ciclo de vida ou aumentar as funções desse sistema na medida em que surgem novos requisitos.

- *Vantagens*

- Fácil de implementar,
- A documentação é produzida em cada etapa.

- *Desvantagens*

- Dificuldade em ter todas as especificações do cliente;
- Prova tardia de operação adequada;
- Não é transparente para o cliente durante o desenvolvimento.

Este modelo é mais adequado para pequenos projectos ou aqueles com especificações bem conhecidas e fixas.

2.2. Modelo em " V "

Derivado do modelo em cascata, o modelo V do ciclo de desenvolvimento mostra não apenas a sequência de fases sucessivas, mas também as relações lógicas entre fases mais distantes.

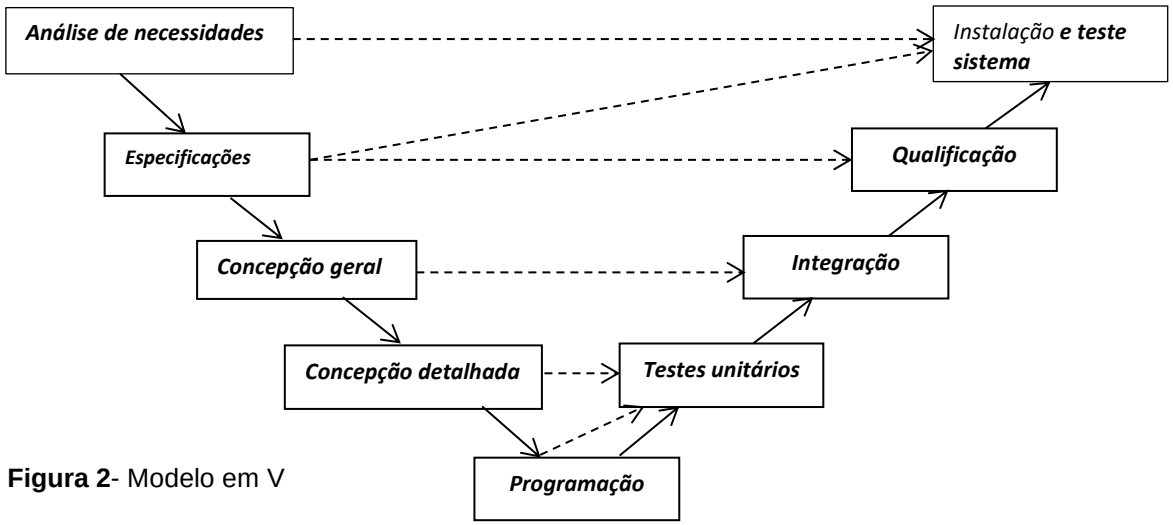


Figura 2- Modelo em V

Além das setas contínuas que reflectem o encadeamento sequencial de etapas no modelo em cascata, notamos a adição de setas descontínuas que representam o facto de que parte dos resultados da etapa inicial é usada directamente pela etapa de chegada, por exemplo, após a conclusão do projecto arquitectural, o protocolo de integração e os jogos de teste de integração devem ser totalmente descritos. O ciclo V é o ciclo que foi padronizado, é amplamente utilizado.

- *Vantagens:*

- Os planos de teste são melhores,
- Quaisquer erros podem ser detectados mais cedo.

- *Desvantagens:*

-Os planos de teste e seus resultados requerem reflexão e *feedback* sobre a descrição actual,
-A parte certa pode ser melhor preparada e planeada.

Este modelo é ideal quando as necessidades são bem conhecidas, quando a análise e a concepção são claras, este modelo é adequado para projectos de média dimensão e complexidade.

2.3. Modelo de prototipagem

A prototipagem também é considerada um modelo de desenvolvimento de *software*.
Trata-se de escrever uma primeira especificação e fazer um subconjunto do produto de *software* final. Este subconjunto é então cada vez mais refinado e avaliado até que o produto final seja obtido.

A prototipagem é uma técnica que consiste em desenvolver rapidamente um "**rascunho**" do que seria o sistema de informação quando ser finalizado. Esse rascunho é chamado de **protótipo**.

Tipos de prototipagem

1) Gastável: esqueleto do *software* que foi criado para um objectivo e em uma fase particular de desenvolvimento

2) Evolutivo: conservado ao longo do ciclo de desenvolvimento. É melhorado e completado para obter o *software* final.

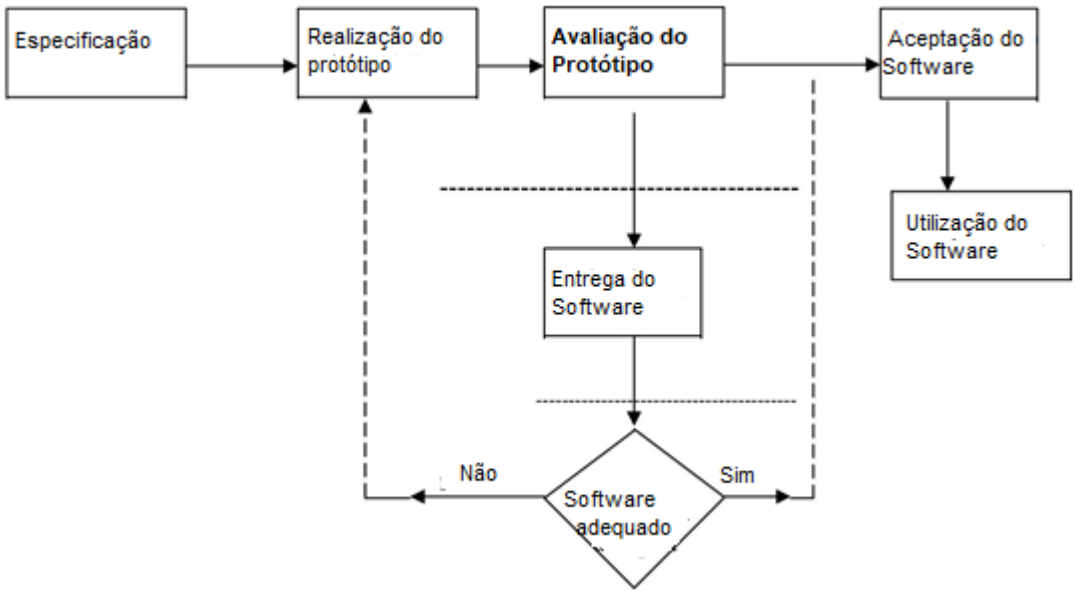


Figura 3- Modelo por prototipagem

- *Vantagens:*

– Validação muito cedo no processo,

– Visibilidade (compreensão) do produto final,

– Antecipação de mudanças: manutenção inclusive ao longo do projecto.

- *Desvantagens:*

– Risco de destruturação do código,

– Dificuldade de planificar e de gerir o desenvolvimento.

Esse modelo é mais adaptado para os projectos inovadores.

2.4. Modelo incremental

O desenvolvimento incremental é uma abordagem mais simplista para a identificação dos requisitos, buscando obter um produto para entrega o mais rápido possível.

- A cada entrega, os requisitos são refinados para que haja a expansão das funcionalidades.
- Modelo que combina elementos dos fluxos de processos lineares e paralelos.

- Aplica sequências lineares, de forma escalonada, à medida que o tempo avança.
- O primeiro ciclo foca nos requisitos essenciais, mínimos para o funcionamento do *software*.

- Cada entrega gera um produto operacional, efectivamente.

Nesse tipo de modelo, apenas um subconjunto dos componentes de um sistema é desenvolvido de uma vez: o *software* núcleo é desenvolvido primeiro e, em seguida, os incrementos são desenvolvidos e integrados sucessivamente.

O processo de desenvolvimento de cada incremento é um dos processos clássicos.

- *Vantagens:*

- Cada desenvolvimento é menos complexo,
- As integrações são progressivas,
- Pode haver entregas e comissionamento após cada integração de incremento.

- *Desvantagens:*

- Envolva o núcleo ou incrementos anteriores,
- Não sendo capaz de integrar novos incrementos.

Este modelo é utilizado em grandes projectos que exigem entregas rápidas.

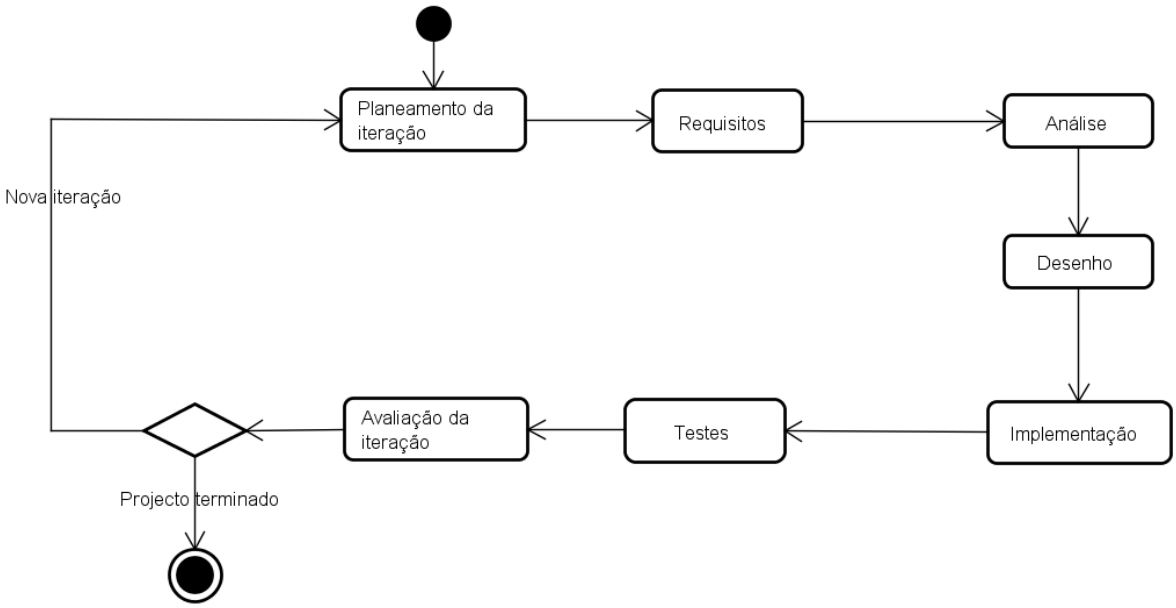


Figura 4- Modelo incremental

2.5. O modelo em espiral

Para corrigir as deficiências da abordagem linear, surgiram os chamados modelos espirais, propostos por B. Boehm em 1988, onde os riscos, quaisquer que sejam, são constantemente tratados através de *loops* sucessivos:

Cada ciclo da espiral ocorre em quatro fases:

1. Determinações dos objectivos do ciclo, alternativas para os atingir, constrangimentos, a partir dos resultados dos ciclos anteriores ou, caso não existam, a partir de uma análise preliminar das necessidades;
2. Análise de risco, avaliação de alternativas, possivelmente prototipagem;
3. Desenvolvimento e verificação da solução escolhida;
4. Revisão dos resultados e planeamento do próximo ciclo.

- *Vantagens:*

- Validade dos requisitos,
- Inclui análise de risco e prototipagem.

- *Desvantagens:*

- Frequentemente, cronograma e orçamento irrealistas,
- Dificuldade em antecipar os componentes necessários nos ciclos subsequentes,
- A sua implementação requer grande competência e deve limitar-se a projectos inovadores pela importância que atribui à análise de risco.

Este modelo é usado para projectos inovadores, em riscos, e cujos os desafios são importantes.

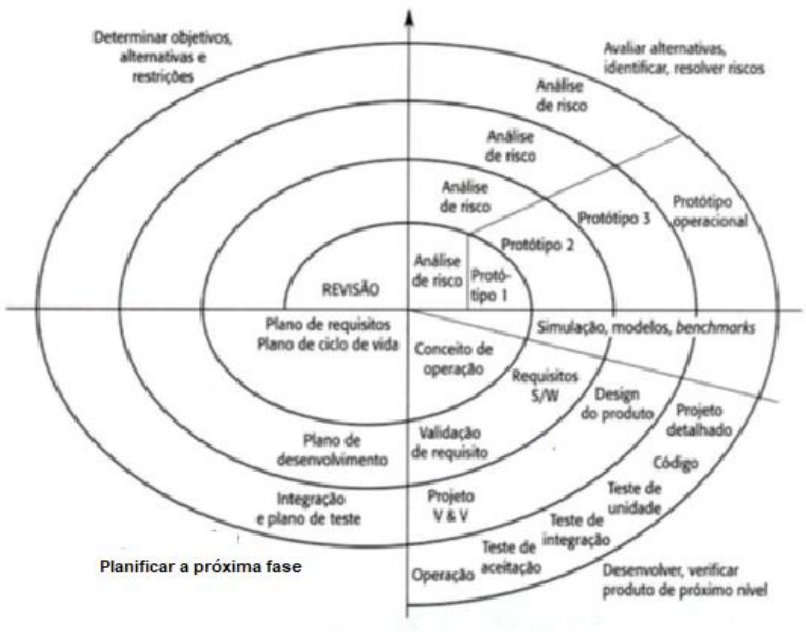


Figura 5- Modelo em espiral

3. Conclusão parcial

A escolha de um modelo depende do campo de aplicação. Quando este é crítico, os modelos adequados são o modelo espiral e, em menor grau, a prototipagem. Para aplicações clássicas (gerenciamento), o modelo V ou o modelo em cascata podem ser apropriados. Para áreas que requerem entregas rápidas, mas não críticas, o modelo incremental poderá ser empregado.

Exercícios

Exercício 1:
Comparar a prototipagem descartável em relação a prototipagem não descartável (evolutivo).

Exercício 2:
Apresentar num quadro comparativo, as vantagens, as desvantagens e também o tipo de projecto correspondente a cada modelo de ciclo de vida.

4. Método Rational Unified Process

4.1. Conceito e objectivo

Rational Unified Process (**RUP**) é um método genérico baseado no paradigma orientado a objectos. Chama-se também de **processo unificado da Rational.**
RUP é um processo de Engenharia de Software criado para apoiar o desenvolvimento orientado a objectos, fornecendo uma forma sistemática para se obter vantagens no uso da UML (*Unified Modeling Linguagem*).

Principal objectivo do RUP:

Atender as necessidades dos utilizadores garantindo uma produção de *software* de alta qualidade. Assim, o RUP mostra como o sistema será construído na fase de implementação, gerando o modelo do projecto e, opcionalmente, o modelo de análise que é utilizado para garantir a robustez. O RUP define perfeitamente quem é responsável, como as coisas deverão ser feitas e quando devem ser realizadas, descrevendo todas as metas de desenvolvimento especificamente para que sejam alcançadas.

4.2. Fases do Rational Unified Process

O RUP organiza o desenvolvimento de *software* em quatro (4) fases, onde são tratadas questões sobre planeamento, levantamento de requisitos, análise, implementação, teste e implantação do *software*.

1) Fase de Concepção /Iniciação: fase que abrange as tarefas de comunicação com o cliente e planeamento. Um plano de projecto é feito avaliando os possíveis riscos, as estimativas de custo e prazos, estabelecendo as prioridades, levantamento dos requisitos do sistema e preliminarmente analisá-lo. Assim, haverá uma anuência das partes interessadas na definição do escopo do projecto, onde são examinados os objectivos para se decidir sobre a continuidade do desenvolvimento.

2) Fase de Elaboração: abrange a modelagem do modelo genérico do processo. O objectivo é de analisar mais detalhadamente o domínio do problema, revisando os riscos que o projecto pode sofrer e a arquitectura do projecto começa a ter sua forma básica. Nesta etapa são esclarecidas indagações como "O plano do projecto é confiável? Os custos são admissíveis?".

3) Fase de Construção: envolve os componentes de *software*. O principal objectivo é a construção do sistema de *software*, com foco no desenvolvimento de componentes e outros recursos do sistema. É na fase de construção que a maior parte de codificação ocorre.

4) Fase de Transição: consiste na entrega do *software* ao utilizador e na realização de testes. O objectivo é de disponibilizar o sistema para o utilizador final. As actividades desta fase incluem o treinamento dos utilizadores finais e também a realização de testes da versão beta (β) do sistema visando garantir que o mesmo possua o nível adequado de qualidade.

Referências básicas

PRESMANN, R. (2011). **Engenharia de Software**: uma abordagem profissional. 7. ed., Rio de Janeiro: Mc Graw Hill. Cap. 2.

SOMMERVILLE, I. (2007). **Engenharia de Software**. 8. ed., Rio de Janeiro: Pearson. Cap. 4.

5. Metodologias ágeis de desenvolvimento de software

5.1. Introdução

As metodologias ágeis de desenvolvimento de *software* são indicadas como sendo uma opção às abordagens tradicionais para desenvolver *softwares*.
Comparadas a outras metodologias, produzem pouca documentação. É recomendado documentar o que realmente será útil.

São recomendadas para projectos para os quais:
- existem muitas mudanças;
- os requisitos são passíveis de alterações;
- a recodificação do programa não acarreta alto custo;
- a equipe é pequena;
- as datas de entrega são curtas e acarretam alto custo;
- o desenvolvimento rápido é fundamental.
- Em essência, as Metodologias Ágeis foram desenvolvidas com o objectivo de vencer as fraquezas percebidas e reais da Engenharia de Software (Pressman, 2010).

5.2. Manifesto Ágil

Trata-se de uma declaração de princípios que fundamentam o desenvolvimento ágil de software. Em 2001, Kent Beck e mais 16 desenvolvedores, produtores e consultores de software, que formavam a Aliança Ágil, assinaram o Manifesto de Desenvolvimento Ágil de Software.

5.3. Os 12 princípios do Manifesto Ágil

1) Garantia da satisfação do consumidor com entrega rápida e contínua de *softwares* funcionais.
2) Mudanças de requisitos, mesmo no fim do desenvolvimento, ainda são bem-vindas.
3) Frequentemente são entregues *softwares* funcionais (semanas, ao invés de meses).
4) Desenvolvedores e pessoas relacionadas aos negócios devem trabalhar, em conjunto, até o fim do projecto.

5) Construir projectos com indivíduos motivados, dar-lhes ambiente e suporte necessários e confiar que farão seu trabalho.
6) Uma conversa face a face é o método mais eficiente e efectivo de transmitir informações para e dentro de uma equipe de desenvolvimento.
7) Software em funcionamento é a principal medida de progresso.
8) Desenvolvimento sustentável, de modo a manter um ritmo constante indefinidamente.
9) Atenção contínua para com a excelência técnica e para com bons projectos aumenta a agilidade.
10) Simplicidade – a arte de maximizar a quantidade de trabalho não efectuado – é essencial.
11) As melhores arquitecturas, requisitos e projectos emergem de equipes auto-organizáveis.
12) Em intervalos regulares, a equipe deve refletir sobre como se tornar mais eficiente.

5.4. Outras metodologias

Pressman (2010) apresenta as seguintes metodologias:

- Extreme Programming (XP)
- Desenvolvimento Adaptativo de Software (DAS)
- Dynamic Software Development Method (DSDM)
- Scrum
- Crystal
- Feature Driven Development (FDD)
- Modelagem Ágil (AM)
- Processo Unificado Ágil (AUP)

5.4.1. Extreme Programming

Extreme Programming utiliza como paradigma de desenvolvimento o Orientado Objecto. Inclui um conjunto de regras e práticas com base nas seguintes actividades:

-Planeamento;
-Projecto;
-Codificação;
-Teste.

1) Planeamento

-Criação de um conjunto de "histórias de utilizadores" descrevendo as características e funcionalidades requeridas pelo *software* que será construído;
-As histórias (semelhantes aos casos de uso) **são escritas pelos clientes** e colocadas em cartões de indexação;

-O cliente atribui uma prioridade à cada história;

-Os desenvolvedores analisam cada história e atribuem um custo a cada uma delas, com base em número de semanas necessárias para o seu desenvolvimento;

-Se a história precisar de mais de três (3) semanas para desenvolvimento, é solicitado ao cliente que ela seja dividida em histórias menores;

-Desenvolvidas em três modos:

i) Todas as histórias serão implementadas imediatamente (dentro de poucas semanas).
ii) As histórias com valor mais alto serão antecipadas no cronograma e implementadas primeiro.
iii) As histórias de maior risco serão antecipadas no cronograma e implementadas primeiro.

-Com o avanço do projecto, o cliente pode adicionar novas histórias, mudar a sua prioridade, subdividi-la e eliminá-las.

2) Projecto

-Segue rigorosamente o KIS (*keep it simple*).

-Estimula o uso de cartões CRC (Classe, Responsabilidade e Colaboração) para a identificação e organização das classes Orientadas-Objecto relevantes para o incremento do *software*.

-Cartões CRC permitem a descrição dos conceitos identificados na metáfora na forma de classes.

-Responsabilidades são identificadas para cada classe.

-As colaborações determinam as interacções entre classes.

Os cartões permitem que o todo o *time* possa colaborar com o *design*.

-Os cartões CRC são o único produto de trabalho do projecto;

-Caso seja identificado um problema difícil na história, recomenda-se a criação imediata de um protótipo operacional daquela parte do projecto. Denominado Solução de Ponta.
-Encoraja a **refatoração**: técnica que altera a estrutura do sistema sem modificar o comportamento externo.

3) Codificação

Depois que as histórias forem desenvolvidas e o início do projecto for feito, recomenda-se não iniciar a programação. A codificação é um elemento chave do XP.
É recomendado realizar testes unitários sobre cada uma das histórias que serão incluídas na versão actual.
Depois de os testes unitários terem sido criados, o desenvolvedor está focado no que deve ser implementado.

Programação em pares:

-duas pessoas trabalhando junto na mesma máquina;
-cada pessoa fica encarregada de uma actividade;
-quando o trabalho dos programadores é completado, é feita uma integração com o trabalho de outros;
-existe uma equipe responsável pela integração.

4) Teste

São aplicados os testes unitários. Os testes de aceitação (ou teste de cliente) são especificados sob a óptica do cliente e abrangem as características e as funcionalidades do sistema global visíveis e passíveis de revisão.
“Resolver pequenos problemas a cada intervalo de umas poucas horas leva menos tempo do que resolver grandes problemas perto da data de entrega”, Wells (1999) *apud* Pressman (2010).

5.4.2 Desenvolvimento Adaptativo de Software

O Desenvolvimento Adaptativo de Software ou *Adaptative Software Development* (ASD) é proposto para auxiliar no desenvolvimento de sistemas e *softwares* complexos.

Concentra-se na colaboração humana e na auto-organização da equipe.
É uma propriedade de sistemas adaptativos e incorpora três fases:

- Especulação;
- Declaração da missão do projecto;
- Identificação das restrições do projecto.

-Realiza o levantamento dos requisitos básicos;
-Colaboração;
-Filosofia de que pessoas motivadas trabalhando juntas multiplicam seus talentos e resultados;
-Aprendizado;
-Clientes/utilizadores informam feedback;
-Revisão dos componentes de *software* desenvolvidos;
-Avaliação do desempenho da equipe DAS.

5.4.3 Dynamic Systems Development Method

Dynamic Systems Development Method (DSDM) é a metodologia que fornece um arcabouço para construir e manter sistemas que satisfazem às restrições de prazo apertadas por uso de prototipagem incremental em um ambiente controlado de projecto (CS3 *Consulting Services*, *apud* Pressman, 2010).

DSDM Consortium é um grupo mundial de empresas que definiu um **modelo ágil** de processo chamado ″ciclo de vida DSDM″, com as seguintes actividades:

-Estudo de viabilidade;
-Estudo do negócio;
-Iteração do modelo funcional;
-Iteração de projecto e construção e;
-Implementação.

1) Estudo de viabilidade

Define-se requisitos básicos e restrições do negócio, avalia se é viável desenvolver a aplicação usando DSDM.

2) Estudo do negócio

- Identifica os requisitos funcionais;
- Define a arquitectura básica da aplicação;
- Identifica os requisitos de manutenibilidade da aplicação.

3) Iteração do modelo funcional

- Constrói um conjunto de protótipos incrementais.
- Objectiva adquirir requisitos adicionais com feedbacks dos s à medida que usam o protótipo.

4) Iteração de projecto e construção

- Verifica os protótipos construídos garantindo que cada um tenha passado por engenharia.

5) Implementação

Coloca o último incremento do software no ambiente de produção, apesar de ele poder ainda não estar completo e novas modificações serem solicitadas.

5.4.4 Scrum

Desenvolvido na década de 90 por Jeff Sutherland, o Scrum apresenta princípios coerentes com os do Manifesto Ágil. Enfatiza o uso de um conjunto de "padrões de processo de *software*" ideais para projectos com prazos apertados, requisitos mutantes e negócios críticos. É baseado em **ciclos de 30 dias** (chamados de ***Sprints***), o Scrum trabalha para alcançar objectivos bem definidos.
Os objectivos são representados no *Product Backlog*, que é uma lista de actividades a serem realizadas.

5.4.4.1 Papéis

-Time

-Product owner

-Scrum master

1) Time

- grupo pequeno, normalmente, entre 5 e 9 pessoas;
- deve ser comprometido com o trabalho a fim de atingir a meta de um Sprint;
- desenvolve e produz com qualidade;
- deve ser cada vez mais auto gerável e multidisciplinar.

2) Product Owner

- faz o intermédio entre o cliente e o fornecedor do produto
- boa noção do produto e das necessidades do cliente;
- responsável por actualizar o *Product Backlog*.

3) Scrum Master

- um líder, mediador e facilitador;
- remove impedimentos da equipe assegurando que as práticas Scrum estão sendo executadas com eficiência.

5.4.4.2 Fluxo de processo

1) Definição do Backlog

-funcionalidades ou mudanças no produto são feitas pelo *Product Owner* no *Product Backlog*;

-a lista é priorizada para reflectir a necessidade dos clientes ou demandas do mercado.

2) Sprints

-unidades de trabalho necessárias para atender os itens do Product Backlog, normalmente, são 30 dias.

3) Reuniões

-diárias, aproximadamente 15 minutos;

-todos os membros do time respondem às perguntas:

P1. O que você fez ontem?

P2. O que pretende fazer hoje?

P3. Que impedimentos estão lhe atrapalhando?

4) Revisões

-o time apresenta os resultados ao *Product Owner* e demais interessados;

-os itens do *backlog* são considerados prontos e inicia-se um novo *sprint*.

Figura 8- Metodologia SCRUM

5.4.5 Família Crystal

Criada por Cockburn e Highsmith, contempla um conjunto de metodologias, cada qual com elementos centrais que são comuns a todas, papéis, padrões de processos, produtos de trabalho e práticas específicas de cada uma (Pressman, 2010).

Objectivo: permitir as equipes ágeis de seleccionar o membro da família mais apropriado para o seu projecto e ambiente.

Cada método *Crystal* é caracterizado por uma cor.
Quatro (4) parâmetros determinam o método de desenvolvimento:

- Tamanho da equipe.
- Localização geográfica.
- Criticalidade /Segurança
- Recursos.

5.4.5.1 Crystal Clear

É uma metodologia leve, para equipes de 1 a 8 pessoas, podendo chegar até 12 casos especiais.

- Yellow: 10 a 20 membros.
- Orange: 20 a 50 membros.
- Red: 50 a 100 membros.

Cada um dos métodos com graus de gerenciamento e de comunicação ajustados de acordo com o tamanho da equipe.
Especificação e projecto são feitos informalmente usando quadros publicamente visíveis.
A metodologia é propositalmente pouco definida. Para permitir que cada projecto implemente as actividades que lhes pareçam mais adequadas. Fornecendo um mínimo de suporte útil a documentação e comunicação.

5.4.6 Feature Driven Development

Define característica como sendo "*uma função valorizada pelo cliente que pode ser implementada em duas semanas ou menos*", Coad (1999) *apud* Pressman (2010).

Os benefícios em se utilizar a filosofia de características são:

- os s podem descrevê-las mais facilmente;
- podem ser organizadas de forma hierárquica;
- desenvolvimento a cada duas semanas;
- facilidade em analisar projecto e código;
- projecto e cronograma são guiados pela hierarquia de características, ao invés de um conjunto de tarefas de engenharia de software adoptado arbitrariamente.

Exemplos:

-Gerar o relatório de venda para um determinado período;

-Exibir as especificações técnicas de um produto;

-Adicionar o produto a um carrinho de compras.

5.4.7 Test Driven Development

Técnica de desenvolvimento de *software* que se baseia em um ciclo curto de repetições:

- O desenvolvedor escreve um caso de teste automatizado que define uma melhoria desejada ou uma nova funcionalidade.

- É produzido código que possa ser validado pelo teste para posteriormente o código ser refatorado para um código sob padrões aceitáveis.

Kent Beck, considerado o criador da técnica, declarou em 2003 que *Test Driven Development* encoraja *designs* de código simples e inspira confiança.

Através desta técnica, os programadores podem aplicar o conceito de melhorar e depurar código legado desenvolvido a partir de técnicas antigas.

-Escreva um teste, antes mesmo de escrever o código que este teste consome.
-Faça o teste funcionar, escrevendo o código do qual o teste depende, mesmo que seja um código ruim.
-Refactore, eliminando duplicações de código, tanto nos testes quanto nas implementações.

CAPÍTULO 2: INTRODUÇÃO A CONCEPÇÃO ORIENTADA OBJECTO

2.1. Introdução

O principal objectivo deste capítulo é fazer com que o aluno compreenda os conceitos orientados a objectos.

A concepção propõe uma solução para o problema especificado durante a análise Existem duas grandes abordagens de concepção: a abordagem orientada a funções (abordagem funcional) e a abordagem orientada a objectos[1].

2.1.1. Abordagem funcional

- Raciocínio em termos de funções do sistema;
- Separação de dados e código de processamento;
- Decomposição funcional descendente.

Os limites da abordagem funcional

- Um programa é concebido como um conjunto de módulos funcionais (procedimentos ou funções) que manipulam dados;
- Comunicação entre funções por passagem de parâmetros ou por variáveis globais;
- Acesso gratuito aos dados por qualquer função;
- Dificuldade em reutilizar código já escrito e testado.

2.1.2 Abordagem orientada a objecto

- Agrupamento dados - processamentos;
- Redução a diferença entre o mundo real e sua representação informática;
- Decomposição por identificação relações entre objectos.

2.2. Conceitos da abordagem orientada a objecto

2.2.1. Objecto

Um objecto é uma entidade representada por um estado e um conjunto de operações que manipulam este estado.

[1] A abordagem de concepção orientada a objectos é baseada na noção de objecto que representa as entidades do mundo real.

Um **objecto** é caracterizado por:

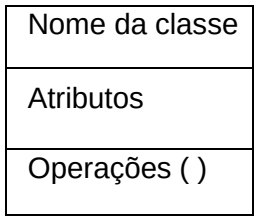
- um **identificador** (nome do objecto) ;
- um **estado** (sob forma de um conjunto de atributos) ;
- um **comportamento** (um conjunto de operações).

2.2.2. Classe

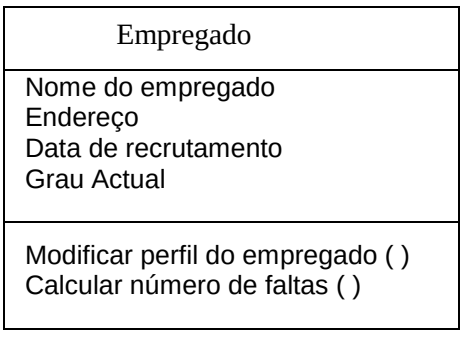
Uma classe corresponde à descrição de uma família de objectos com a mesma estrutura e o mesmo comportamento.
Uma classe define, uma parte estática e uma parte dinâmica:

- A parte estática é representada por um conjunto de **atributos** que podem ter um valor (esses atributos caracterizam o estado dos objectos durante suas execuções).
- A parte dinâmica é representada pelo conjunto de **operações** que definem o comportamento comum aos objectos da mesma classe.

Notação gráfica[2]

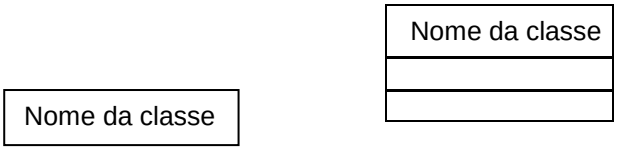


Exemplo de uma classe:



[2] As notações gráficas utilizadas nesta disciplina são inspiradas de UML (*Unified Modeling Language*).

Nota: Uma classe pode ser citada apenas com seu nome, sem especificar os detalhes.



Instância de uma classe: Uma instância é um objecto criado a partir de uma classe. A criação de um objecto a partir de uma classe chama-se a instanciação.

2.2.2.1. Associação entre classes

-Uma **associação** representa um relacionamento entre várias classes. Corresponde à abstracção[3] das ligações que existem entre objectos no mundo real. Uma associação pode ser identificada por seu nome. É possível expressar multiplicidades (cardinalidades) na ligação de associação.

A **multiplicidade** indica um domínio de valores para especificar o número de instâncias de uma classe frente a outra classe para uma determinada associação. Define o número de instâncias da associação para uma instância de uma classe.

2.2.2.2. Notação de multiplicidades de associação

É possível especificar o papel de uma classe dentro de uma associação. A função é colocada em uma extremidade da ligação de associação, portanto, é diferenciada do nome da associação localizado no centro da ligação.

O quadro seguinte apresenta as multiplicidades de associação.

1	um e um só
0..1	zero ou um
*	de zero a vários
0..*	de zeros a vários
1..*	de 1 a vários
N	Exactamente N
M..N	de M a N (inteiros naturais)

[3] Abstracção consiste em identificar as características interessantes de uma entidade em vista a um uso preciso.

Uma associação pode ser **binária** ou ***n-ária***:

1) **Associação binária:** Uma associação binária é representada por uma ligação simples.

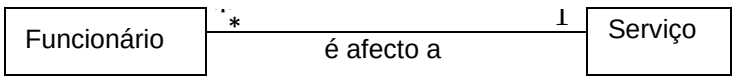


Figura 11- Exemplo de associação binária com multiplicidades

Neste exemplo, representa-se o facto que um funcionário é afecto a um só serviço mas este último pode acolher vários funcionários.

É possível especificar o papel de uma classe dentro de uma associação. O papel (ou função) é colocado em uma extremidade da ligação de associação, portanto, é diferenciada do nome da associação localizado no centro da ligação.

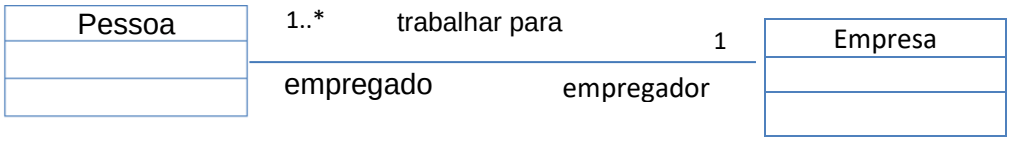


Figura 12 -Modelagem de classes com nomes dos papéis e sua multiplicidade

Neste exemplo, uma pessoa trabalha para uma e apenas uma empresa. A empresa emprega pelo menos uma pessoa. A empresa é a empregadora das pessoas que nela trabalham e uma pessoa tem estatuto de empregado na empresa.

Observação *:* Pode-se encontrar várias associações entre duas classes.

Exemplo*:*

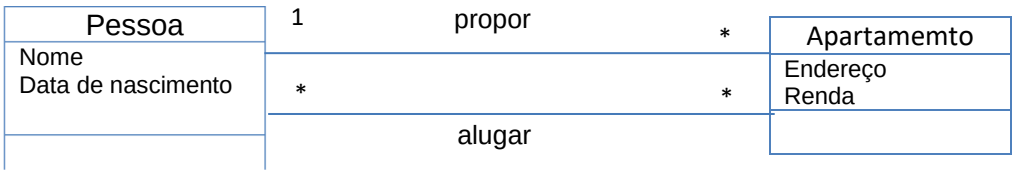
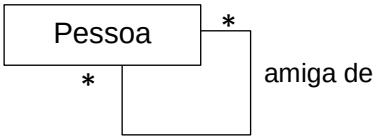


Figura 13- Modelagem de classes com nomes dos papéis e sua multiplicidade

Repara-se que quando a ligação existe entre objectos da mesma classe, fala-se de associação reflexiva.

Exemplo*:*

Duas pessoas podem ser amigos (as) (considera-se que a amizade é recíproca).



2) Associação n-ária: Uma associação n-ária liga mais de duas classes. É representado graficamente por um losângulo com um caminho que leva a cada classe participante. O nome da associação, se aplicável, aparece perto do losângulo.

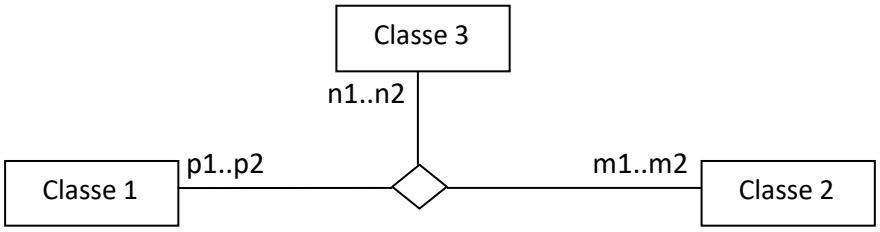


Figura 14- Formalismo de associação ternária

Para uma associação ternária, as cardinalidades são lidas como o seguinte: Para um par (casal) de instâncias de classe 1 e classe 2, há pelo menos n1 instâncias de classe 3 e no máximo n2.

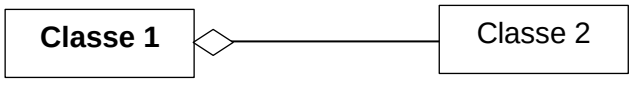
2.2.3. Agregação e Composição

Existe ligações de associação particulares, tais que a agregação e a composição.

2.2.3.1 Agregação

Uma agregação é um tipo de associação que descreve uma relação de inclusão entre uma parte e um tudo (agregado). Agregação se representa por um pequeno losângulo branco do lado do agregado.

Notação gráfica:



Exemplo de uma agregação

No quadro de uma formação o currículo desta é uma agregação de módulos a ensinar.

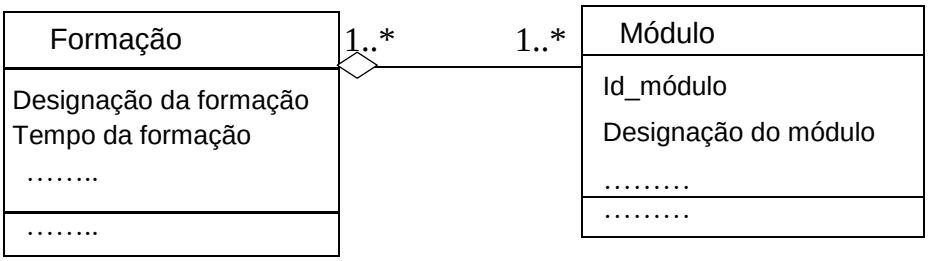


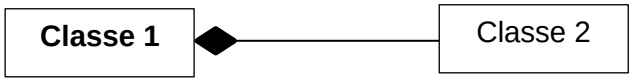
Figura 15- Exemplo de uma agregação

Repara-se neste exemplo, que a supressão de uma formação não conduz automaticamente a supressão dos módulos sendo que os últimos podem muito bem ser ensinados em outras formações.

2.2.3.2 Composição

Uma composição é uma forma forte de agregação. Isto é, a supressão do objecto agregado implica a supressão dos objectos agregandos. A cardinalidade do lado do composto (*composite*) não deve ser maior que 1 (1 ou 0..1). A composição se representa por um pequeno losângulo de cor preta.

Notação gráfica:



Exemplo de uma composição:

Uma encomenda é composta de um conjunto de linhas de encomenda que descreve os produtos encomendados. A supressão de uma encomenda levará necessariamente à supressão de todas as suas linhas.

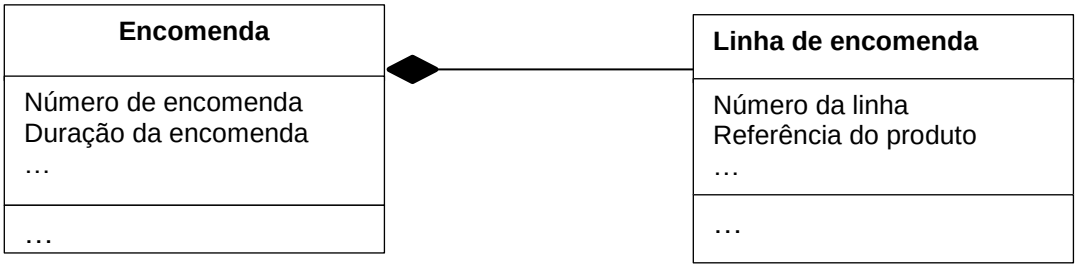
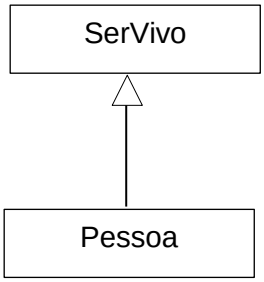


Figura 16- Exemplo de uma composição

2.2.4. Herança

A herança permite um compartilhamento hierárquico de propriedade (atributos e operações). A relação de herança entre duas classes é representada por uma seta na ponta com a forma de um triângulo branco.

Exemplo: Aqui a classe Pessoa que herda da classe SerVivo.

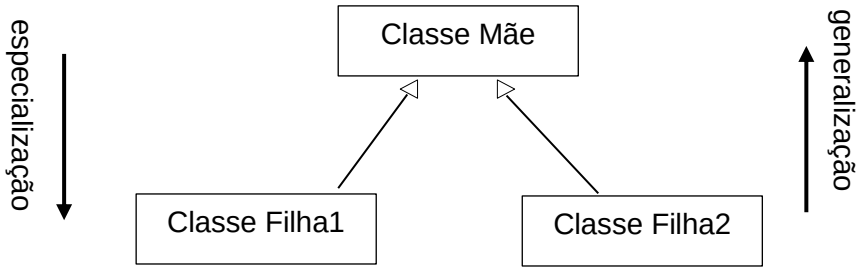


2.2.4.1. Implementação

A herança é implementada graças a duas propriedades que são: **generalização** e **especialização**.

- A generalização descreve o facto de poder reagrupar um conjunto de classes compartilhando elementos em comum em uma única superclasse (ou classe mãe)
- A especialização representa o fenómeno inverso, ou seja, poder derivar de uma classe ou superclasse subclasses (ou classes-filhas) possuindo propriedades específicas que os distinguem umas das outras.

Formalismo:



2.2.4.2. Tipos de herança

1) Herança simples

A herança é simples quando uma classe herda de uma superclasse.

Exemplo:

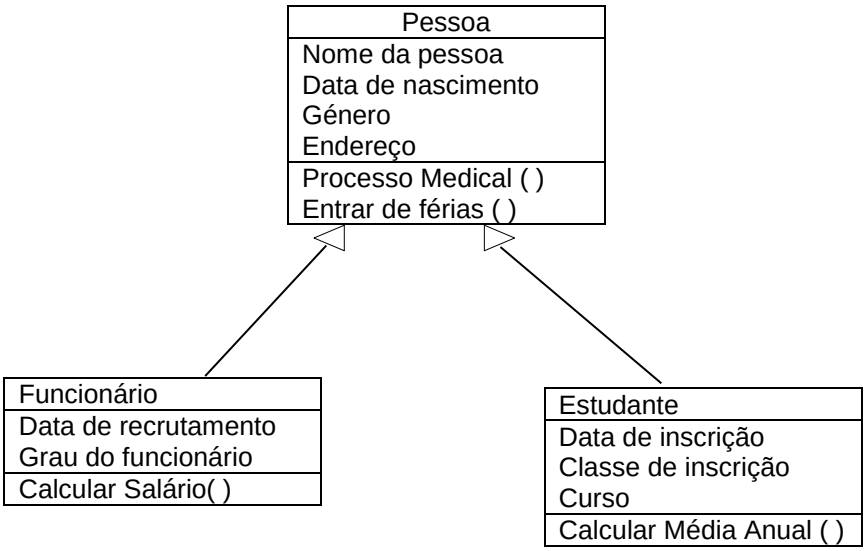


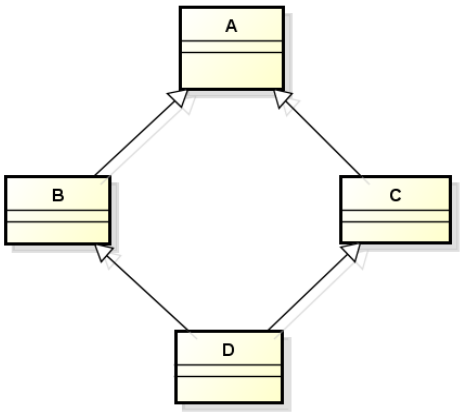
Figura 17- Exemplo de uma herança simples

As classes “Funcionário” e “Estudante” são derivadas da classe “Pessoa” e herdam as propriedades desta. No entanto, "Funcionário" se distingue por outros atributos, como data de recrutamento e salário, enquanto "Estudante" tem data de inscrição, classe de inscrição, curso e média anual.

2) Herança múltipla

A herança é múltipla quando há duas ou mais superclasses para a mesma subclasse.

Formalismo de herança múltipla:



2.2.5. Polimorfismo

O polimorfismo consiste, mantendo o mesmo nome para um método herdado, em associar um código específico que passa a substituir o do método herdado.

Exemplo:

Se tomarmos o exemplo de herança simples, a operação «Entrar de férias( )» herdada pelas classes "Funcionário" e "Estudante" poderia ter implementações diferentes para essas duas subclasses. Portanto, seria preferível implementar esta operação por dois métodos diferentes: um para a subclasse "Funcionário" e outro para a subclasse «Estudante».

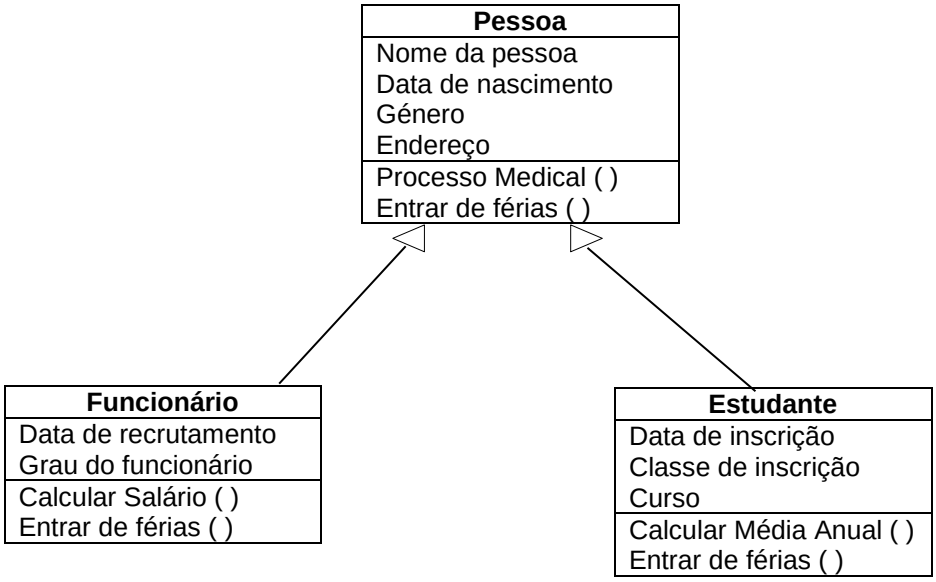


Figura 18- Exemplo de uma herança simples

2.2.6. Encapsulamento

O encapsulamento é um mecanismo que consiste de reunir dados e métodos dentro de uma estrutura enquanto oculta a implementação do objecto. O encapsulamento permite definir níveis de visibilidade dos elementos da classe.

Existem **três níveis diferentes de visibilidade**: público, protegido e privado. O encapsulamento é representado por um sinal de mais "**+**" no caso de púbico, um sinal de menos "**–**" no caso de privado e um "**#**" agudo no caso de protegido.

A tabela a seguir detalha o significado desses sinais:

público	**+**	elemento não encapsulado visível para todos
protegido	#	elemento encapsulado visível em subclasses da classe
privado	**-**	elemento encapsulado visível apenas na classe

Notação gráfica e Exemplo:

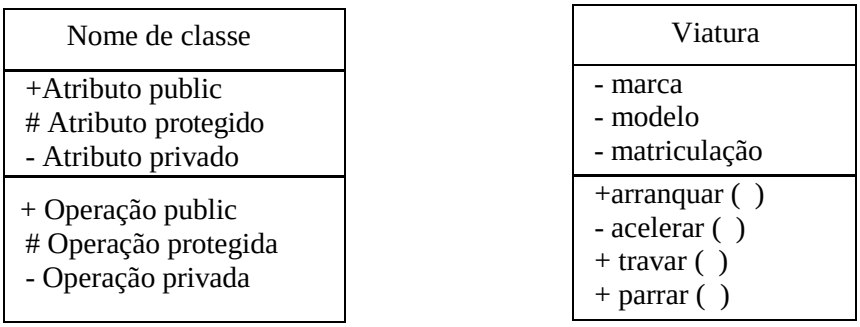


Figura 19- Exemplo de níveis de visibilidade

2.3. Conclusão

Este capítulo apresentou o pensamento orientado a objectos. Em suma:

- A abordagem orientada a objectos permite modelar sua aplicação na forma de interacções entre objectos.
- Os objectos têm propriedades e podem realizar acções.
- Os objectos mascaram a complexidade de uma implementação por meio do encapsulamento.
- A noção de herança corresponde ao reaproveitamento (ou reutilização) de um conceito para a especificação de um novo conceito, seja pela especialização de um conceito mais geral em vários conceitos mais específicos, seja pela generalização de vários conceitos em um conceito mais geral.

2.4. Exercícios

Vide caderno de exercícios_COO.

CAPITULO 3: LINGUAGEM DE MODELAGEM UNIFICADA A OBJECTOS-UML (*Unified Modeling Language*)

3.1. Introdução

O principal objectivo deste capítulo é permitir ao estudante descobrir a UML (*Unified Modeling Language*). UML é uma linguagem de modelagem para especificação, construção, visualização e documentação de software.

É uma linguagem gráfica para modelar sistemas e processos, baseada na abordagem orientada a objectos, aquela que primeiro levou à criação de linguagens de programação como Java, C ++, C # ou *Smalltalk*.

3.1.1. Uma breve historial

As principais etapas da disseminação da UML podem ser resumidas da seguinte forma:

- 1994 - 1996: fusão de 3 métodos OMT[4], BOOCH[5] e OOSE[6] e nascimento da primeira versão da UML.
  - **OMT** de James Rumbaugh (*General Electric*) fornece uma representação gráfica dos aspectos estático, dinâmico e funcional de um sistema;
  - **BOOCH (OOD)** de Grady Booch, define para o *Department of Defense*, introduz o conceito de pacote (package);
  - **OOSE** de Ivar Jacobson (Ericsson) fonda a análise sobre a descrição das necessidades dos utilizadores (**casos de uso** ou ***use cases***).
- 23 de novembro de 1997: versão 1.1 da UML adoptada pela OMG[7] que teve como objectivo definir um padrão notação utilizável nos desenvolvimentos informáticos baseados sobre o objecto;
- 1998 - 1999: lançamento das versões UML 1.2 a 1.3.
- 2000 - 2001: lançamento das seguintes versões mais recentes 1.x.
- 2002 - 2003: preparação da versão 2.
- 10 de outubro de 2004: lançamento da versão 2.1.
- 5 de fevereiro de 2007: lançamento da versão 2.1.1.

[4] OMT : Object Modeling Technique
[5] BOOCH : nome do conceptor Grady Booch
[6] OOSE : Object Oriented Software Engineering
[7] OMG: Object Management Group

Os trabalhos de melhoria da UML 2 continuam ... UML é articulada em torno de vários tipos de diagramas.
A UML é unificada porque vem de várias notações que a precederam. Hoje, a UML é promovida pelo *Object Management Group* (OMG), um consórcio de mais de 800 empresas e universidades que atuam na área das tecnologias de objecto.
A UML agora se tornou uma linguagem de modelagem amplamente usada, em particular graças à sua riqueza semântica que a torna abstrata de muitos aspectos técnicos.

3.1.2. Abordagem

Duas abordagens relacionadas à UML:

(1) O Processo Unificado, processo de desenvolvimento e evolução de software;

(2) A arquitectura MDA (*Model-Driven Architecture*) destinada à realização de sistemas desconsiderando a plataforma física e seus aspectos tecnológicos.

3.1.3. Arquitetura baseada em modelo

Model-Driven Architecture (MDA) é uma proposta da OMG cujo objectivo é o projecto de sistemas baseado na modelagem única do domínio, desconsiderando os aspectos tecnológicos. A partir dessa modelagem, o MDA se propõe a obter por transformação os elementos técnicos capazes de funcionar dentro de uma plataforma de software como Java ou .NET.

No MDA, o modelo de objecto de domínio é denominado *Platform-Independent Model* (PIM). O PIM é composto por um conjunto de elementos que devem ser projetados independentemente de qualquer linguagem de programação ou tecnologia. Esse modelo é então transformado manual ou automaticamente em um modelo específico para uma plataforma e uma linguagem de programação. Esse modelo específico é denominado *Platform-Specific Model* (PSM).

O *link* com a UML reside no nível do PIM. Na verdade, a UML é uma óptima candidata como linguagem nesse nível. A UML tem a vantagem de descrever objectos com precisão, enquanto permanece independente de tecnologias.

3.1.4. Abstracção

A abstracção é um princípio muito importante na modelagem. Consiste em reter apenas as propriedades relevantes de um objecto para um problema específico. Os objectos usados em UML são abstracções de objectos do mundo real. Esta abstracção é suportada por diagramas.

3.1.5. Diferentes tipos de diagramas UML

Existem dois tipos de visualizações do sistema, cada uma com seus próprios diagramas:

- **Vistas estáticas:**

  - diagramas de classes
  - diagramas de objectos
  - diagramas de casos de uso
  - diagramas de componentes
  - diagramas de implantação

- **Vistas dinâmicas:**

  - diagramas de sequência
  - diagramas de colaboração (ou comunicação)
  - diagramas de estado-transições
  - diagramas de actividades.

3.2. Diagrama de Classes

Este diagrama representa a descrição estática do sistema integrando em cada classe a parte dedicada aos dados e a parte dedicada ao processamento. É o diagrama chave (pivô) de toda a modelagem dum sistema.

O diagrama de classes expressa a estrutura estática do sistema em termos de classes e relacionamentos ou associações entre classes.

3.2.1 Características dos atributos

A sintaxe dum atributo é o seguinte :

[Visibilidade] nome [: tipo] [multiplicidade] [= valorPadrão]

Onde:

- Acolchetes indicam cláusulas opcionais: apenas o nome é necessário.
- *Nome de atributo* : nome dado ao atributo.
- *Tipo*: tipo do atributo.
- *Visibilidade*: consultar as explicações fornecidas no capítulo 2.
- *Multipilicidade*: indica o número de valores possíveis do atributo para um objecto.
- *valorPadrão*: valor por defeito associado ao atributo de um objecto se não houver valor especificado na sua criação.

3.2.2 Características das operações

A sintaxe de uma operação é o seguinte:

[Visibilidade] nome ([lista de parâmetros]) [: tipo]

Onde:

- *Visibilidade* : consultar as explicações dadas no capítulo 2.
- *Nome* : nome dado à operação.
- *lista de parâmètros* : definição de um ou vários parâmètros.

A ausência de parâmetro é indicado por ( ).

- *Tipo* : tipo de valor reenviado pela operaçãon.

Uma operação que não reenvia valor é indicado por exemplo pela palavra reservado « void » na linguagem C++ ou Java.

3.2.3 Associação entre classes

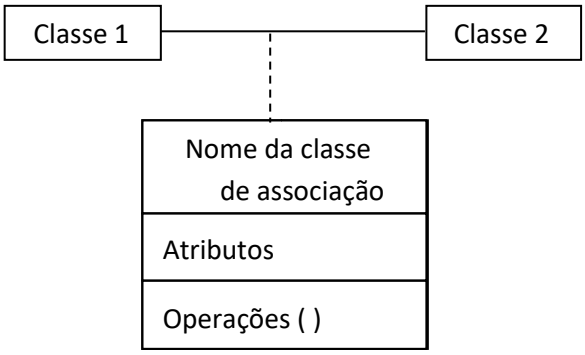
Uma associação entre classes representa os relacionamentos que existem entre as instâncias dessas classes. Uma associação pode ser identificada pelo seu nome. É possível exprimir as multiplicidades (cardinalidades) sobre o *link* de associação.

A multiplicidade indica um domínio de valores para especificar o número de instâncias de uma classe perante uma outra classe para uma determinada associação.

3.2.4. Classe de associação

Uma classe de associação é necessária quando uma associação deve possuir suas próprias propriedades.

Notação gráfica:



Exemplo :

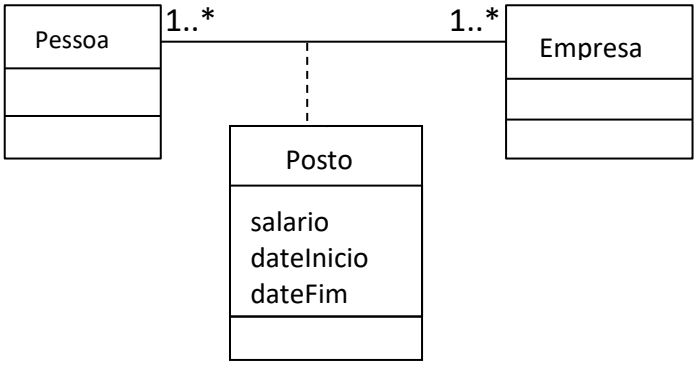


Figura 20- Exemplo de classe de associação

Observação*:*

As notações gráficas e os exemplos apresentados no Capítulo 2 “Introdução a concepção orientado a objectos” fazem parte da notação do diagrama de classes.

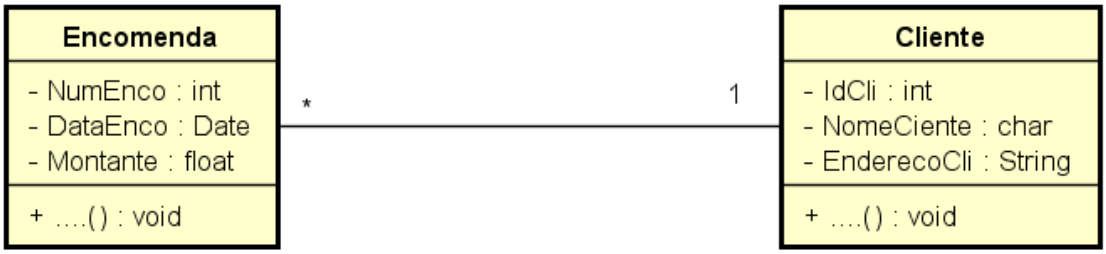
3.3. Diagrama de objectos

O diagrama de objectos permite a representação de instâncias das classes e das associações entre instâncias.

O diagrama de objectos modela **factos** e o diagrama de classes modela as **regras**.

Exemplo*:*

No diagrama de classes a seguir, representamos uma associação entre duas classes **Encomenda** e **Cliente**.



Se estivermos interessados em um cliente específico para ver as encomendas que ele fez, representamos isso por um diagrama de objectos dado pela figura a seguir:

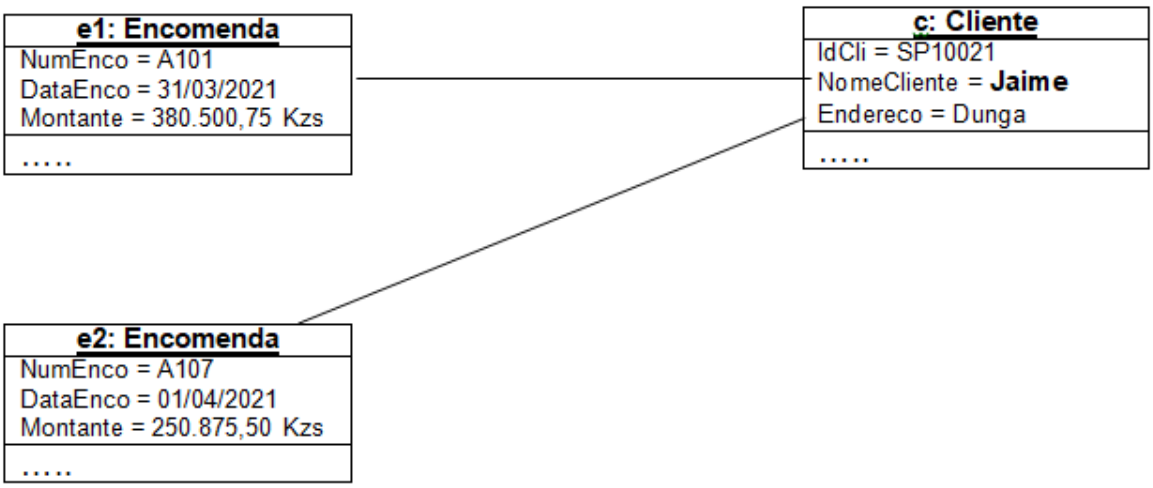


Figura 21- Ilustração de um diagrama de objectos

3.4. Diagrama de casos de uso

Este diagrama é destinado para representar as necessidades dos utilizadores em relação ao sistema.

3.4.1. Conceitos de base

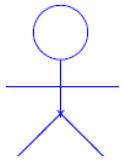
A representação de um diagrama de caso de uso envolve três conceitos: o actor, o caso de uso e a interação entre o actor e o caso de uso.

3.4.1.1 Actor

Um actor representa uma função desempenhada por uma entidade externa (utilizador humano, dispositivo de hardware ou outro sistema) que interage diretamente com o sistema estudado.

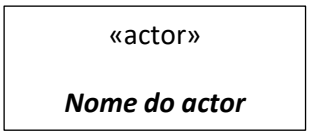
Representação gráfica

Um actor é representado por um *stickman* com seu nome inscrito de baixo.



Nome do actor

Um actor pode também ser representado sob forma de um arquivo estereotipado «actor».

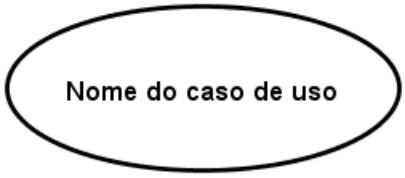


3.4.1.2 Caso de uso

Um caso de uso (*use case* – UC em inglês) é uma série de acções que o sistema terá que executar em resposta à necessidade de um actor.
Um caso de uso corresponde a um objectivo do sistema, motivado por uma necessidade de um ou mais actores.

Representação gráfica

Um caso de uso é representado por um elipso contendo o nome do caso (um verbo ao infinitivo ou o serviço).



3.4.1.3 Relação

A relação expressa a interacção existente entre um actor e um caso de uso.

Representação gráfica

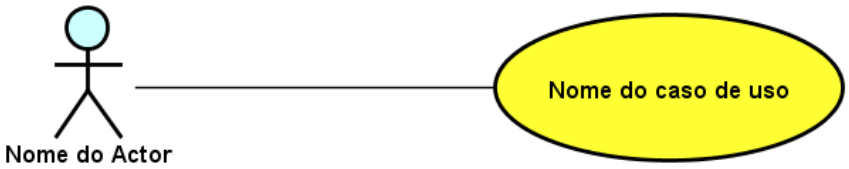


Figura 22- interacção entre actor e caso de uso

Observação:

- Os casos de uso podem estar contidos em uma estrutura que representa os limites do sistema. O nome do sistema está dentro da moldura na parte superior. Os actores estão, então, necessariamente fora da estrutura, uma vez que não fazem parte do sistema.

- Diz-se que o actor é o **principal** para um caso de uso quando este (caso) é útil para esse actor. Um actor **secundário** é solicitado para informações adicionais. Por convenção e na medida do possível, os actores principais estão localizados à esquerda do caso de uso. Os actores secundários estão localizados à direita do caso de uso.

3.4.2. Relações entre casos de uso

Três (3) tipos relações podem ser descritos entre os casos de uso: relação de generalização, relação de e relação inclusão de extensão.

3.4.2.1 Relação de generalização

Em uma relação de generalização entre dois casos de uso, o caso de uso filho é uma especialização do caso de uso pai.

Representação gráfica e exemplo:

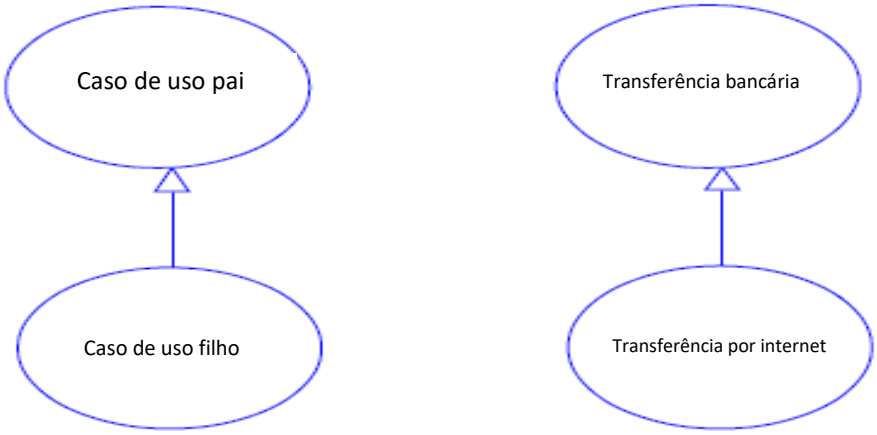


Figura 23- Formalismo da generalização

Um actor também pode participar de relações de generalização com os outros actores. Os actores "filhos" poderão então se comunicar com os mesmos casos de uso do que os actores "pais".

Exemplo: Relação de generalização entre 2 actores

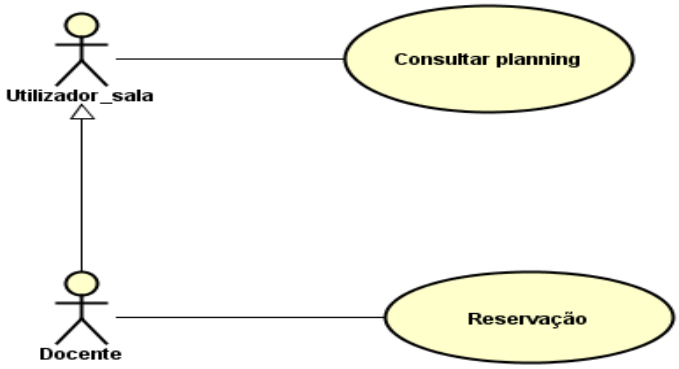
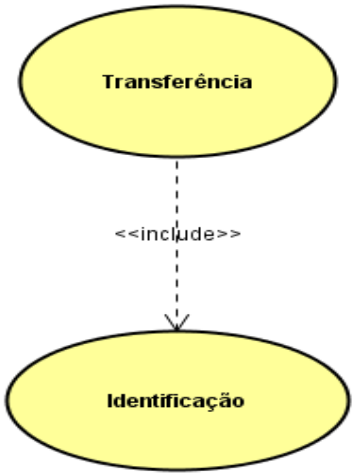


Figura 24- Exemplo da generalização

3.4.2.2 Relação de inclusão

Indica que o caso de uso de origem **contém também o comportamento descrito** no caso de uso de destino. A inclusão é obrigatória, a fonte que especifica para qual coloque o caso de uso de destino deve ser incluído. Esse relacionamento ajuda a quebrar comportamentos e definir comportamentos compartilháveis entre vários casos usar.

Representação gráfica e exemplo:



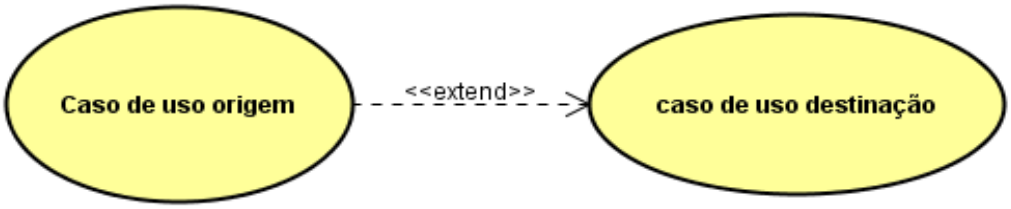
Para atingir o objectivo "transferência", é necessário usar "identificação"

Figura 25- Exemplo de relação de inclusão

3.4.2.3 Relação de extensão

Indica que o caso de uso de origem adiciona seu comportamento ao caso de uso destino. A extensão pode estar sujeita a condição. O comportamento adicionado é inserido em nível de um ponto de extensão definido no caso de uso de destino. Essa relação permite modelar as variantes comportamentais de um caso de uso (de acordo com interacções entre as partes interessadas e o ambiente do sistema).

Representação gráfica:



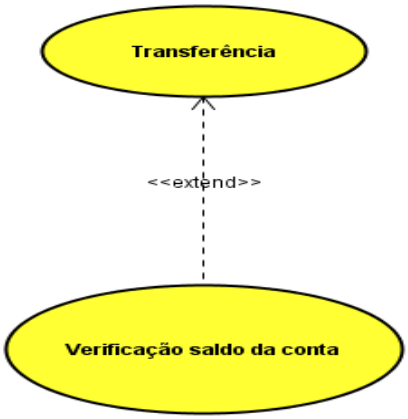


Figura 26- Exemplo de relação de extensão

3.4.3. Identificação de casos de uso

Para elaborar os casos de uso, é preciso se basear sobre as entrevistas com os utilizadores.

Um caso de uso deve, em primeiro lugar, ser simples, inteligível e descrito de maneira clara e concisa. O número de actores que interagem com o caso de uso geralmente é limitado. Geralmente, não há um caso de uso para um actor.

Ao desenvolver um caso de uso, deve-se perguntar:

- Quais são as tarefas do actor?
- Que informações o actor precisa para criar, salvar, modificar ou ler?
- O actor deve informar o sistema de mudanças externas?
- O sistema deve informar o actor das condições internas?
- Quais são as condições para iniciar e parar o caso de uso?

Os casos de uso podem ser apresentados através de **múltiplas visões**: um actor com todos os seus casos de uso, um caso de uso com todos os seus actores.

Um caso de uso é uma abstracção: **descreve de maneira abstracta um conjunto de cenários**. Portanto, não perceba muitos casos de uso porque seria uma falta de abstracção. Em qualquer sistema, há relativamente poucos casos de uso, mas muitos cenários. Um grande número de casos de uso significa, portanto, que a essência do sistema não é compreendida.

3.4.4. Descrição de casos de uso

Para descrever um caso de uso, toma-se em conta os seguintes elementos: Designação do caso de uso, descrição (acções), actor(es).

Formalismo 1: Para um determinado sistema em modelação, a tabela abaixo mostra-nos uma descrição adequada de casos de uso identificados.

Nº	Designação do caso de uso	Descrição	Actor
1.	Caso de uso 1	Descrever as acções do caso de uso. É o cenário do UC	▪ Actor(es) envolvidos
2.	Caso de uso 2	Descrever as acções do caso de uso. É o cenário do UC	▪ Actor(es) envolvidos
…	…	……..	…
n.	Caso de uso n	Descrever as acções do caso de uso. É o cenário do UC	▪ Actor(es) envolvidos

Formalismo 2

1) Para a autenticação, utiliza-se a seguinte descrição :

Autenticação do utilizador	
Actores	Geralmente o Administrador do sistema
Pré-condição	Nenhuma
Pós-condição	Um utilizador válido é conectado, e sua sessão é registada no sistema.
Fluxo de eventos principal	1-O utilizador informa o *login* e senha. 2-O sistema verifica se o *login* e a senha são válidos (verifica-se se o *login* e senha pertencem a uma conta). 3-O sistema regista o início da sessão de utilizador.
Fluxos secundários	No passo 2, se o *login* ou a senha forem inválidos, o sistema exibe uma mensagem e volta ao passo 1.

2) Para os casos de uso principais :

Actores	**Actor(es) principal (ais)**
Operação	Corresponde a uma funcionalidade do sistema
Cenário principal	Descrição do funcionamento do caso sob a forma de uma sequência de mensagens trocadas entre os actores e o sistema. Contem **sempre uma sequência nominal** que corresponde ao funcionamento nominal do caso.
Pré-Condições	Indicam em que estado o sistema se encontra antes que se desenrola a sequência.
Pós-Condições	Indicam em que estado o sistema se depois do desenrolamento da sequência nominal. E confirma-se que a operação é realizada com sucesso.

3.5. Diagrama de sequência

O objectivo do diagrama de sequência é de representar as interacções entre objectos indicando a cronologia das trocas.
Os diagramas de sequência têm intrinsecamente uma dimensão temporal, mas não representam explicitamente as ligações entre os objectos.

Assim, eles favorecem a representação temporal para a representação espacial e são mais aptos de modelar os aspectos dinâmicos do sistema.

3.5.1. Linha da vida dos objectos

A linha de vida dos objectos é representada por uma linha vertical em linhas tracejadas, colocada sob o símbolo do objecto em questão. Essa linha especifica a existência do objecto concernido durante um certo período de tempo.

Em geral, uma linha de vida é representada em toda a altura do diagrama de sequência. No entanto, ele pode iniciar e parar dentro do diagrama.

Representação gráfica

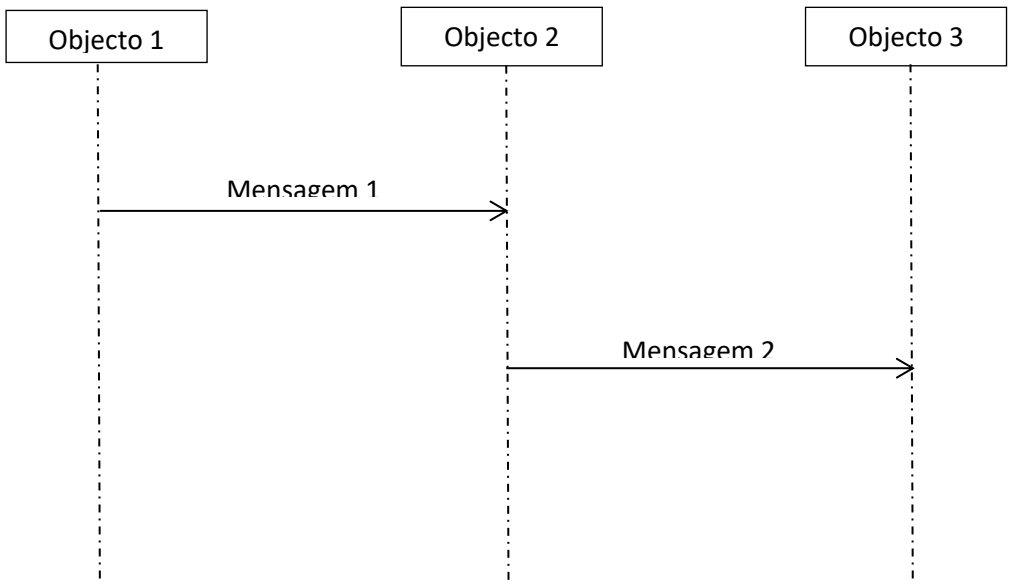


Figura 27- Formalismo do diagrama de sequência

O diagrama de sequência é usado para exibir as mensagens por uma leitura de cima para baixo. O eixo vertical representa o tempo, o eixo horizontal os objectos que colaboram. Uma linha pontilhada vertical é anexada a cada objecto e representa sua duração de vida.

3.5.2. Exemplo de diagrama de sequência

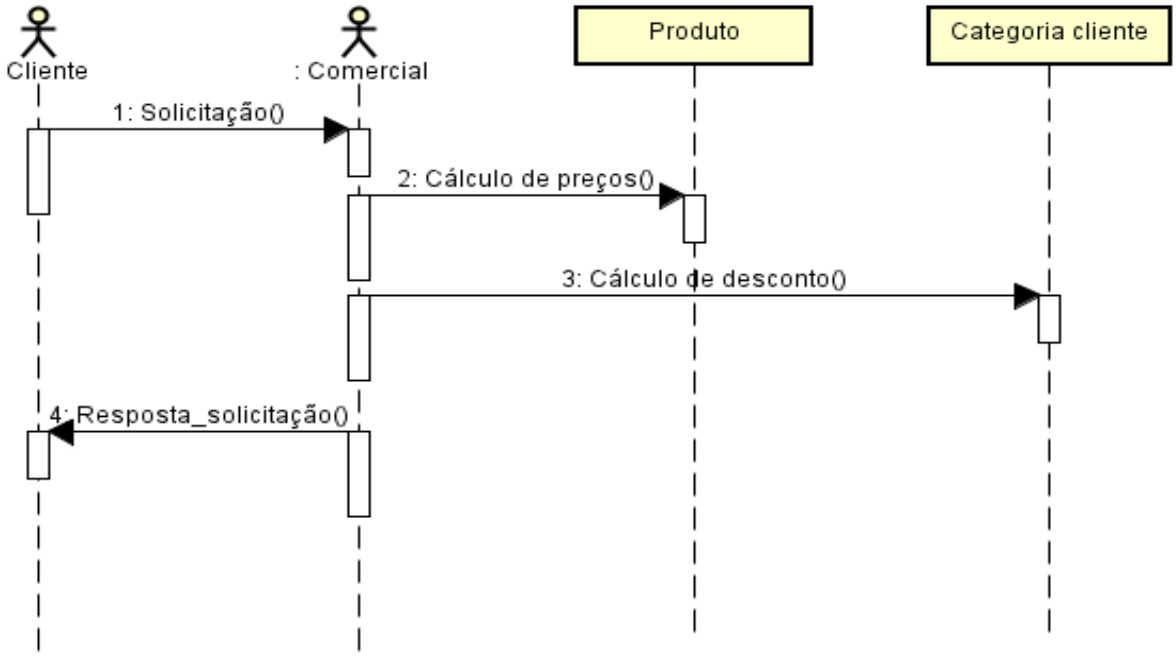


Figura 28- Exemplo do diagrama de sequência

3.5.3. Barra de activação

Os diagramas de sequência permitem de representar os períodos de actividade dos objectos.

Um período de actividade é o tempo durante o qual um objecto realiza uma acção, directamente ou por meio de outro objecto que serve como subcontratado.

Os períodos de actividade são representados por faixas rectangulares colocadas na linha de vida dos objectos.

Representação gráfica:

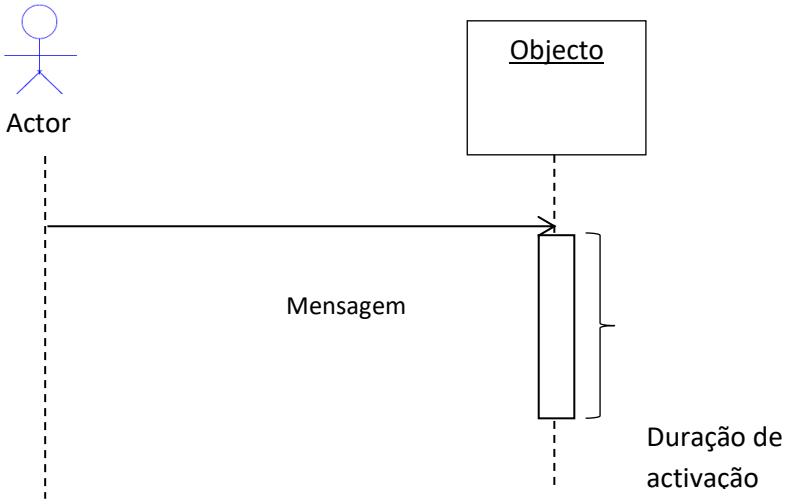
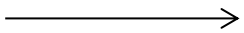


Figura 29- Barra de activação um diagrama de sequência

3.5.4. Mensagem síncrona e Mensagem assíncrona

- **Mensagem síncrona** : o emissor fica em espera da resposta a sua mensagem antes de perseguir suas acções. A mensagem de retorno **pode não ser** representada pois é incluso no fim de execução da operação do objecto destinatário da mensagem.
- **Mensagem assíncrona :** o emissor não espera a resposta a sua mensagem, ele continua a execução de suas operações.

Mensagens	**Notação gráfica**
Síncrona	
Assíncrona	
Retorno	

3.5.5. Mensagem reflexiva

Um objecto pode enviar uma mensagem a si mesmo.

Representação gráfica

Esta situação é representada por uma seta que retorna em um *loop* na linha de vida do objecto.

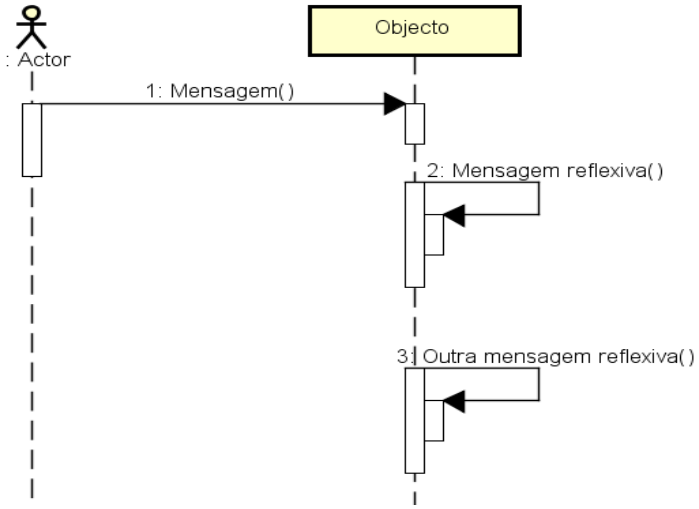


Figura 30- Mensagem reflexiva no diagrama de sequência

Observação: Num diagrama de sequência, pode-se também ser adicionadas anotações temporais que se referem a mensagens.

Exemplo :

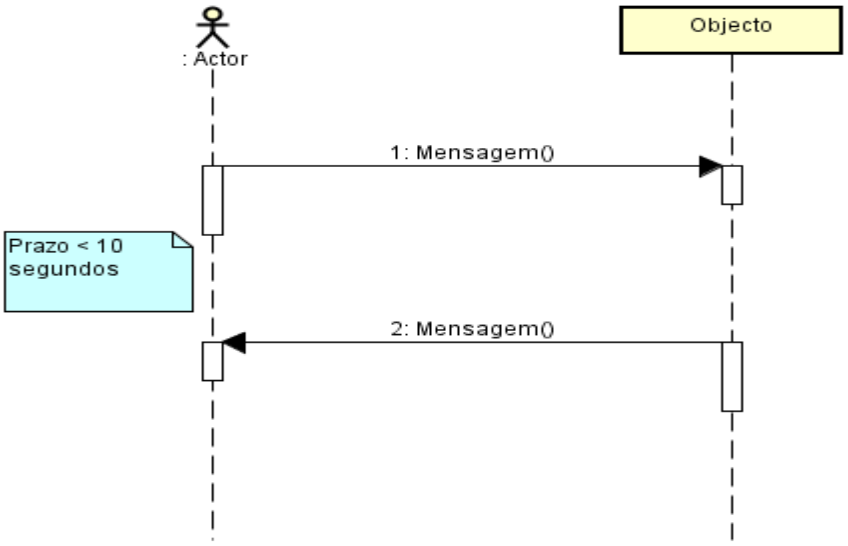


Figura 31- Exemplo do diagrama de sequência com anotação

3.5.6. Criação de objecto e destruição de mensagem

A criação é representada apontando a mensagem de criação sobre o rectângulo que simboliza o objecto criado. A destruição é indicada pelo final da linha de vida e por uma cruz (X), seja na altura da mensagem que causa a destruição, ou seja após a última mensagem enviada por um objecto que se suicida.

Observação:

Na modelagem por objecto, os diagramas de sequência são usados de duas maneiras diferentes, dependendo da fase do ciclo de vida e do nível de detalhe desejado.

- O primeiro uso corresponde à documentação de casos de uso; concentra-se na descrição da interacção, muitas vezes em termos próximos da utilização e sem entrar em detalhes de sincronização. A indicação nas setas corresponde então aos eventos que ocorrem no campo da aplicação. Nesta etapa da modelagem, as setas ainda não traduzem envios de mensagens no sentido de linguagens de programação.

- O segundo uso corresponde a um uso mais informático. O conceito de mensagem unifica todas as formas de comunicação entre objectos (chamada de procedimento, por exemplo).

3.5.7. Fragmento de interacção

Um fragmento de interacção dito combinado corresponde a um conjunto de interacções ao qual um operador é aplicado. Vários operadores foram definidos em UML: loop, alt, opt, etc.

- **Operador loop**

O operateur *loop* corresponde a uma instrução de iteração.

Exemplo :

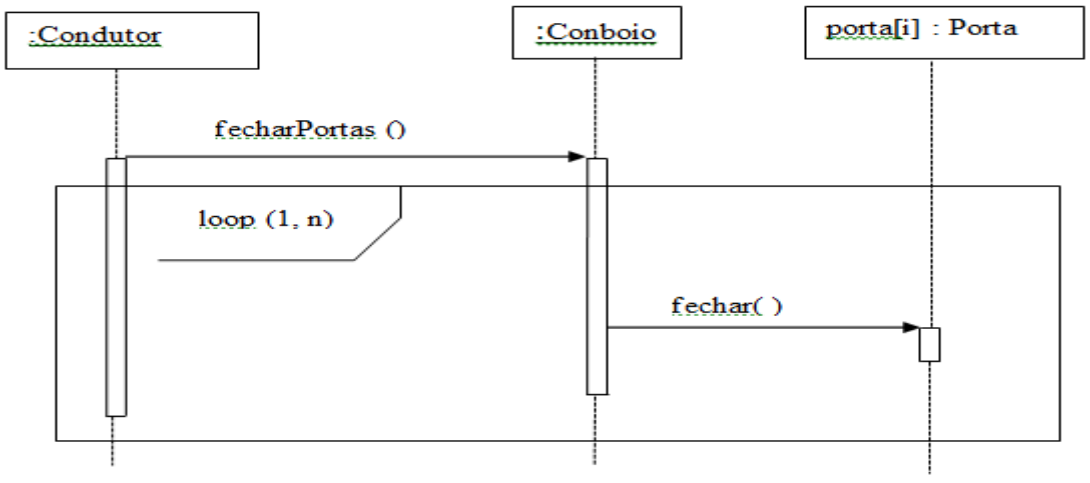


Figura 32- Ilustração de fragmento de interacção com *loop*

- **Operador alt**

O operador *alt* corresponde a uma instrução de teste com uma ou mais alternativas possíveis. Também é permitido usar cláusulas do tipo outra forma (*else*). É representado como um fragmento que possui pelo menos duas partes separadas por linhas pontilhadas.

3.6. Diagrama de comunicação

O diagrama de comunicação é outra representação das interacções diferente do diagrama de sequência. Na verdade, o diagrama de comunicação dá mais ênfase ao aspecto espacial das trocas do que ao aspecto temporal.

Exemplo :

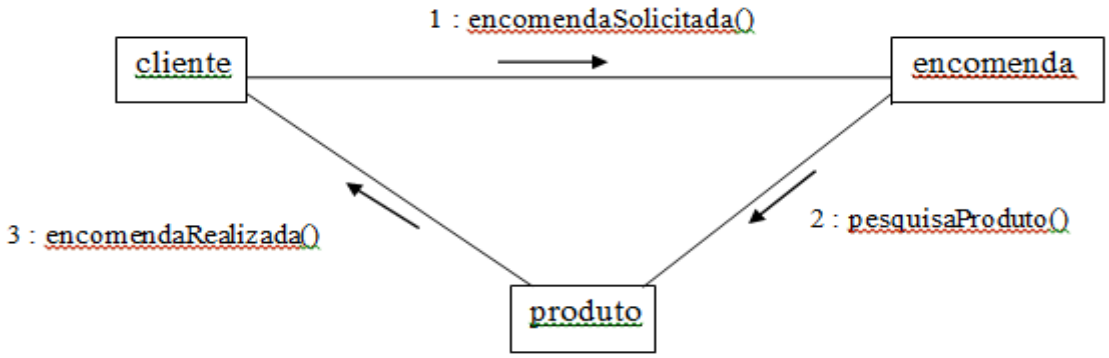


Figura 33- Ilustração de diagrama de comunicação

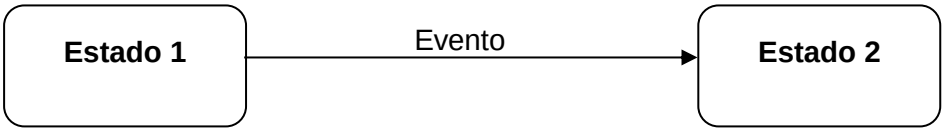
3.7. Diagramas de Estados - transições

Seu papel é representar os tratamentos (operações) que irão gerir o campo estudado. Eles definem a sequência de estados de classe e, assim, mostram o agendamento do trabalho.

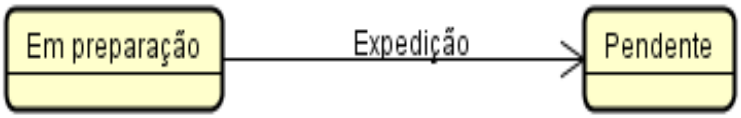
O diagrama de estados - transições é associado a uma classe para a qual diferentes estados são gerenciados: ele permite representar todos os estados possíveis, bem como os eventos que causam as mudanças de estado.

Notação gráfica :

Os diagramas de estados-transições visualizam autômatos de estados finitos, do ponto de vista de estados e transições. Os estados são representados por retângulos com cantos arredondados, enquanto as transições são representadas por arcos orientados ligando os estados.



Exemplo: Uma encomenda entrará no estado "Pendente" assim que tiver sido enviada.



a) **Estado**:

Noção abstrata que mostra a maneira como reage um objecto a um evento.

b) **Transição**:

Uma transição representa a passagem instantânea de um estado para um outro.

c) **Evento**:

Uma transição pode ser desencadeada por um evento, condicionada por "guardas" (**expressão booleana**), e / ou ser associada a uma ação.

Formalismo :

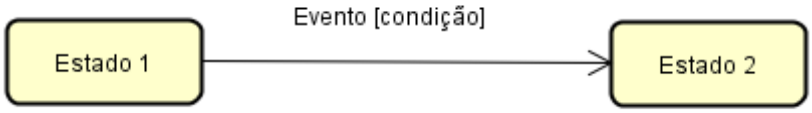
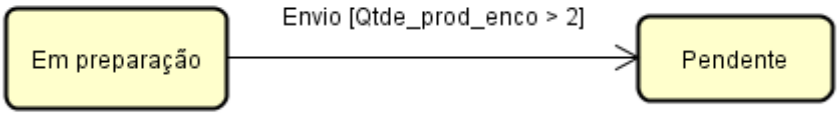


Figura 34- Formalismo do diagrama de estado-transição

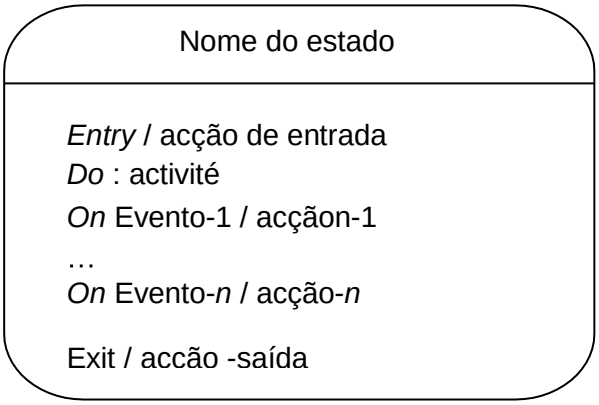
Exemplo: A encomenda é enviada somente se a encomenda contiver pelo menos 3 produtos.



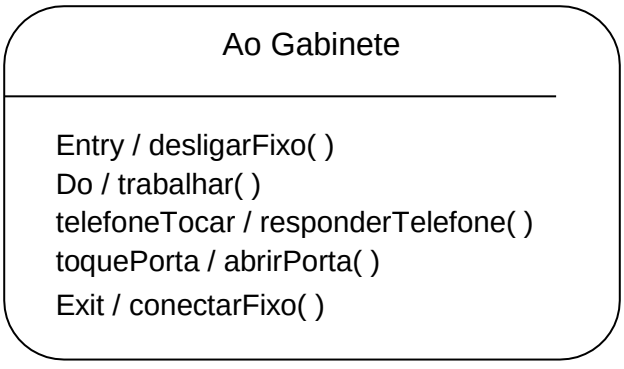
Forma geral de um estado

Os estados também podem incluir **acções**; estas são executadas ao entrar ou sair de um estado ou na ocorrência de um evento enquanto o objecto está no estado em questão. Assim como as classes, os estados podem ser divididos em compartimentos que contêm:

- O nome do estado;
- Transições internas, ou seja, a lista de ações ou actividades internas realizadas enquanto o elemento está neste estado



Exemplo:



Observação:

Uma acção é uma operação instantânea que não pode ser interrompida; está associada a uma transição.

Uma actividade é uma operação de certa duração que pode ser interrompida, está associada a um estado de um objecto.

3.8. Diagramas de actividades

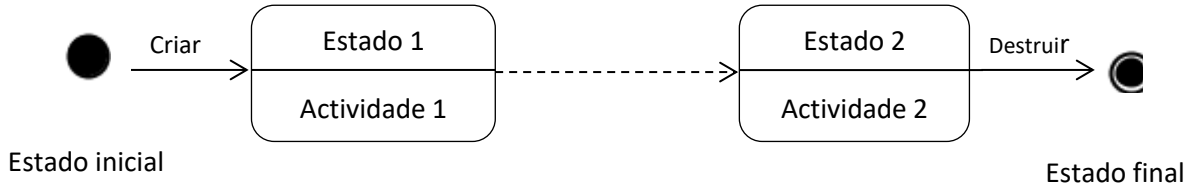
O diagrama de actividades é ligado a uma categoria de classe e descreve o desenrolamento das actividades desta categoria. O desenrolamento chama-se "**fluxo de controlo**". Indica a parte tomada por cada objecto na execução de um trabalho. Será enriquecido pelas condições de sequenciamento.

O diagrama de actividades poderá comportar sincronizações para representar os processamentos paralelos.

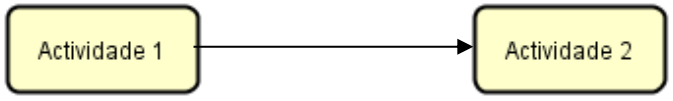
A noção de **corredor de actividade** vai descrever as responsabilidades repartindo as actividades entre os diferentes actores operacionais.

3.8.1. Desenrolamento sequencial das actividades

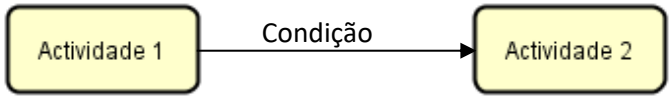
O diagrama de Estado-transição visto anteriormente apresenta já um sequenciamento das actividades de uma classe.



O diagrama de actividades vai modificar esta representação para conservar apenas o sequenciamento. A noção de estado desaparece e obtemos o seguinte grafo:



Como no diagrama de Estados-transições, a transacção pode ser completada por uma **condição**.

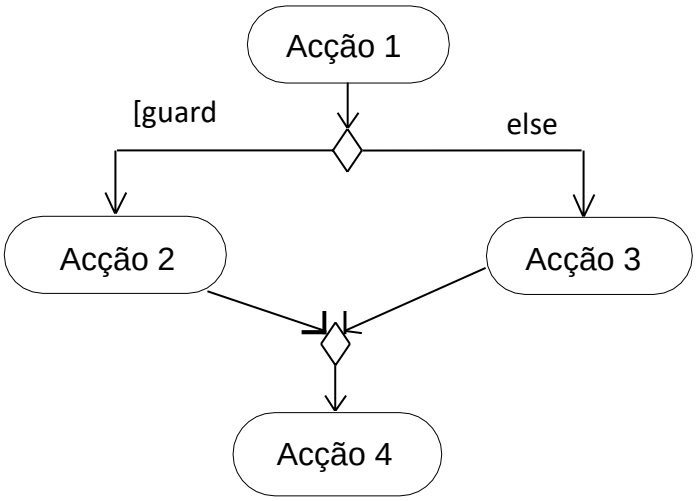


3.8.2. Decisão / Fusão

O comportamento condicional é descrito por decisões / fusões.

Uma decisão (ou ramificação) é usada para representar transacções condicionais usando guardas (expressões booleanas).

Uma fusão marca o fim do comportamento condicional.



3.8.3. Desconexão e junção (Sincronização)

É possível sincronizar as transições usando as barras de sincronização. Estas são usados para abrir (desconectar) ou fechar (juntar) ramificações paralelas em um fluxo de execução.

Desconexão: as transições que começam em uma ramificação ocorrem ao mesmo tempo.

Junção: Uma junção só é cruzada após todas as transições anexadas a ela terem sido concluídas.

Exemplo:

A figura 33 descreve o processamento de uma encomenda por um diagrama de actividades.

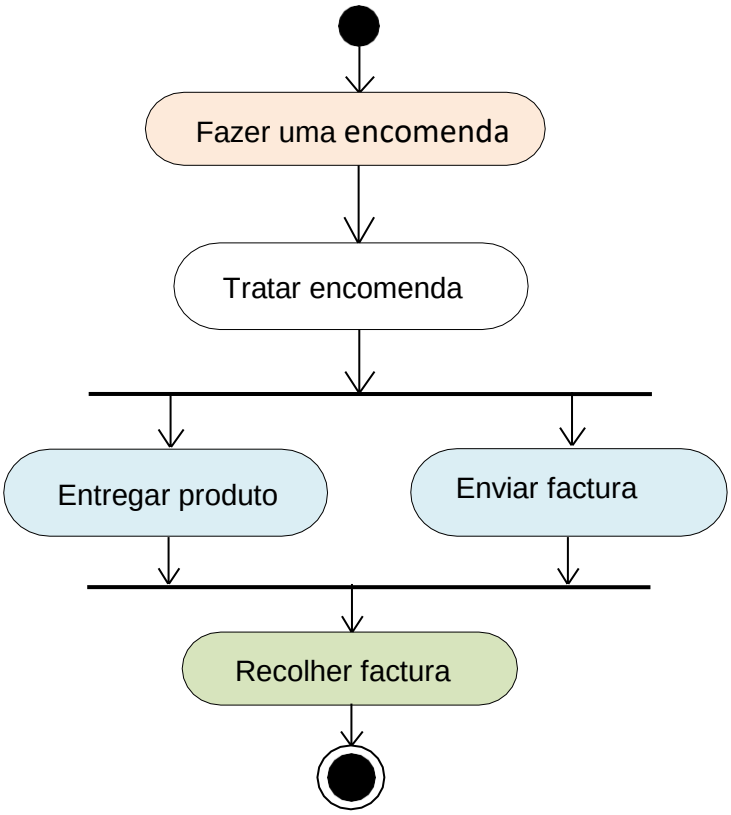


Figura 35- Diagrama de actividades de processamento de uma encomenda

3.8.4. Corredor de actividades

Os corredores de actividades são usados para organizar um diagrama de actividades de acordo com os actores ou gestores das actividades representadas (*Swimlanes*). É ainda possível identificar os objectos principais, que são manipulados de actividade em actividade.

Exemplo:

A figura 34 descreve o processamento de uma encomenda feita por um cliente ao seu fornecedor.

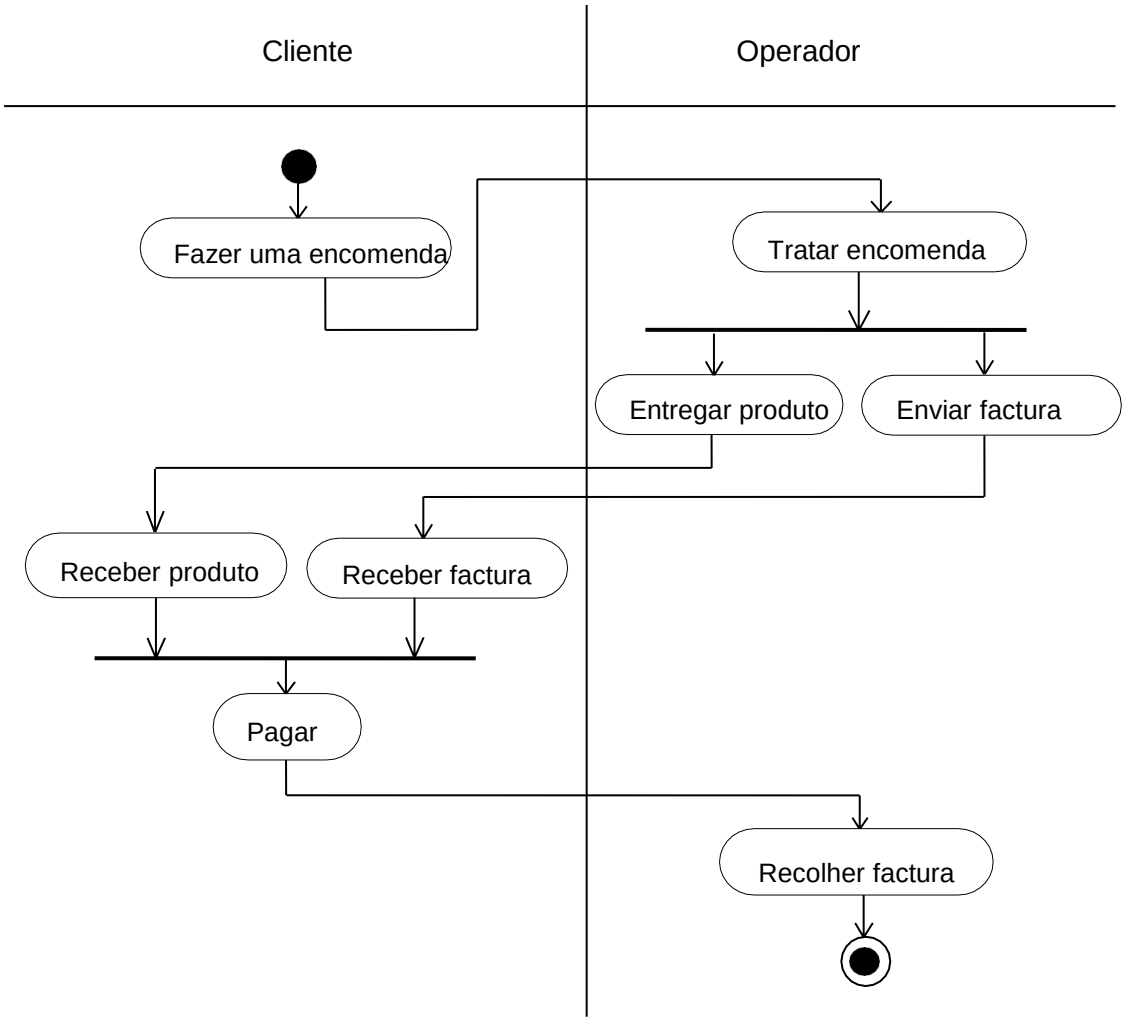


Figura 36- Processamento de uma encomenda por um diagrama de actividades com corredores de actividades

3.9. Diagramas de componentes

Os diagramas de componentes descrevem os componentes e suas dependências no ambiente de realização e em geral, são usados apenas para sistemas complexos.

Um componente é uma visão física que representa uma parte implementável de um sistema.

Um componente pode ser **código**, um **script**, um **arquivo de comandos**, um **arquivo de dados**, uma **tabela**, etc. Ele pode realizar um conjunto de interfaces que definem o comportamento oferecido a outros componentes.

Exemplo

Para um sistema de gestão de Stock (SGS), o diagrama de componentes pode ser o seguinte:

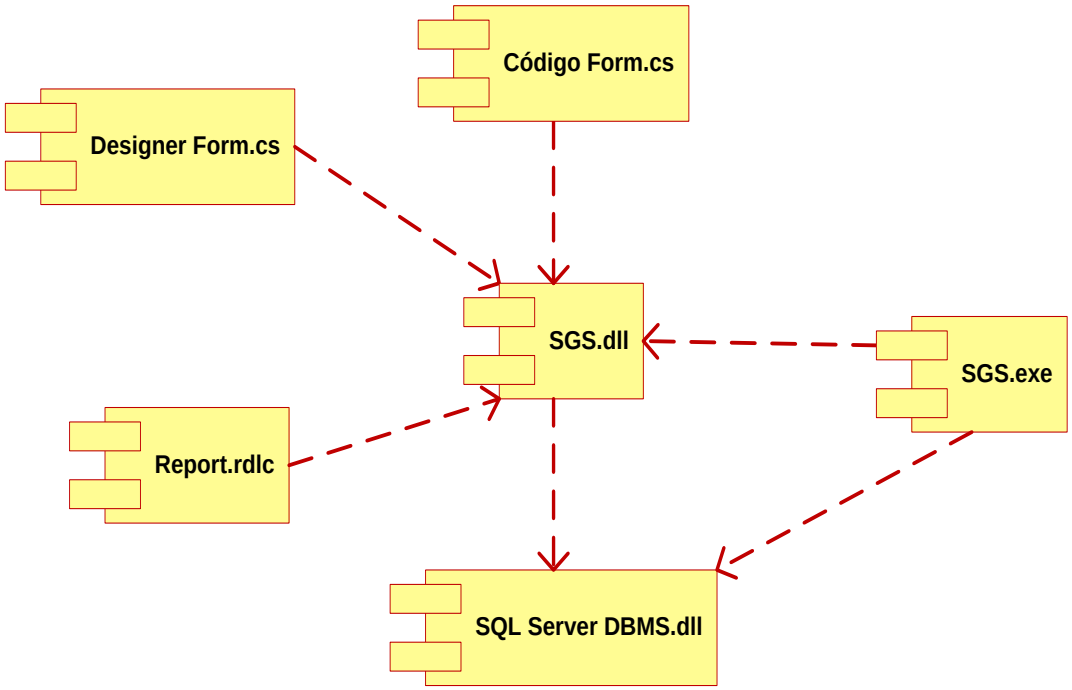


Figura 37- Exemplo diagramas de componentes

A UML define cinco (5) estereótipos para componentes:

- " **documento** ": qualquer documento
- " **executável** ": um programa que pode ser executado
- " **arquivo** ": um documento contendo um código-fonte ou dados
- " **biblioteca** ": uma biblioteca estática ou dinâmica
- " **tabela** ": uma tabela de banco de dados relacional.

Para mostrar instâncias de componentes, um diagrama de implantação pode ser usado.

CONCLUSÃO

Neste capítulo, apresentamos a linguagem UML. Essa linguagem foi projectada para ser uma linguagem de modelagem visual comum, e rica semanticamente e sintacticamente. Destina-se à arquitectura, a concepção e implementação de sistemas de *software* complexos por sua estrutura, bem como por seu comportamento. A UML é composta por diferentes tipos de diagramas. No geral, os diagramas UML descrevem o limite, a estrutura e o comportamento do sistema e dos objectos contidos nele. UML é a linguagem mais padronizada e adoptada na comunidade de desenvolvimento de *softwares* e é patrocinada e mantida pela organização OMG.

BIBLIOGRAFIA

B. Charroux, A. Osmani, Y. Thierry-Mieg (2009). UML 2, pratique de la modélisation, éditions synthex.
D. Gustafson (2003). Génie Logiciel, Dunod, Paris.

Grady Booch, James R. and Ivar Jacobson (2002). The UML Use Guide, Ed. Addison Wesley.

I. Sommerville (1992). Software Engineering, Addison Wesley Publishers, Anckland.

Ivar JACOBSON, Grady BOOCH, James RUMBAUGH. *Le Processus unifié de* Ken Arnold, James Gosling. *The Java Programming Language, développement logiciel*. Eyrolles, Paris, 2000.

J. Gabay, D.Gabay (2008). UML 2 Analyse et conception, Mise en œuvre guidée avec études de cas, Dunod.
M. Lemoine (1996). Précis de génie logiciel, Masson, Paris.

P. Roques (2006). UML 2 par la pratique - Etudes de cas et exercices corrigés, éditions Eyrolles, Paris.

P-A. Muller, N.Gaertner, Modélisation objet avec UML, éditions Eyrolles, 2003.
Patrice CLEMENTE (2003). Dicas de Engenharia de Software, 2° ano STI, ENSI BOURES.

Pierre-Alain MULLER, Nathalie GAERTNER (2003). Modélisation objet avec UML. 2. ed., Eyrolles. ISBN n° 2-212-11397-8.

Pressman Roger S. Engenharia de Software: Uma abordagem professional. 7. ed. McGrawHill, Rio de Janeiro, 2011.

Rumbaugh, J., Jacobson, I. & Booch, G. (2004). Uml 2.0 guide de référence. Campus Press.