Design de Interação

- Unidade 1 – ELEMENTOS DO DESIGN DE INTERAÇÃO

Tema 1: Sistemas interativos em contexto

1. [Qual é o papel dos sistemas interativos na atividade humana?](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-1-qual-e-o-papel-dos-sistemas-interativos-na-atividade-humana)
2. [O ser humano interage com o sistema, por meio do sistema, ou ambos?](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-1-o-ser-humano-interage-com-o-sistema-por-meio-do-sistema-ou-ambos)

Tema 2: Atributos de qualidade da interação nas perspectivas cognitiva e semiótica

1. [Perspectiva teórica do humano na interação humano-computador](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-perspectiva-teorica-do-humano-na-interacao-humano-computador)
2. [O contexto da experiência do usuário](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-o-contexto-da-experiencia-do-usuario)
3. [Atributos de usabilidade, comunicabilidade e acessibilidade em design de interação](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-atributos-de-usabilidade-comunicabilidade-e-acessibilidade-em-design-de-interacao)
4. [A interação pela lente cognitiva](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-a-interacao-pela-lente-cognitiva)
5. [A interação pela lente semiótica](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-a-interacao-pela-lente-semiotica)

- Unidade 2 – PROCESSO DE DESIGN DE INTERAÇÃO

Tema 1: Perspectivas de design

1. [Projetar de modo independente do usuário, projetar centrado no usuário ou projetar com o usuário?](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-1-projetar-de-modo-independente-do-usuario-projetar-centrado-no-usuario-ou-projetar-com-o-usuario)
2. [A tarefa de projetar consiste em empregar a técnica ou exercer a criatividade?](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-1-a-tarefa-de-projetar-consiste-em-empregar-a-tecnica-ou-exercer-a-criatividade)

Tema 2: A prática de design

1. [Modelagem de usuários e suas necessidades](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-modelagem-de-usuarios-e-suas-necessidades)
2. [Abordagens de prototipagem](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-abordagens-de-prototipagem)
3. [Cores, fontes e sua harmonização](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-cores-fontes-e-sua-harmonizacao)
4. [Componentes e recomendações ergonômicas](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-componentes-e-recomendacoes-ergonomicas)
5. [Princípios e regras de design](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-principios-e-regras-de-design)

- Unidade 3 – AVALIAÇÃO DA QUALIDADE DE SISTEMAS INTERATIVOS

Tema 1: Abordagens de avaliação

1. [Por que, quando e quem participa da avaliação de sistemas interativos?](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-1-por-que-quando-e-quem-participa-da-avaliacao-de-sistemas-interativos)

Tema 2: Métodos de avaliação

1. [Inspeção do percursos cognitivo](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-inspecao-do-percursos-cognitivo)
2. [Inspeção semiótica](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-inspecao-semiotica)
3. [Avaliação da acessibilidade](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-avaliacao-da-acessibilidade)
4. [Teste de usabilidade](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-teste-de-usabilidade)
5. [Investigação da usabilidade](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-investigacao-da-usabilidade)

- Unidade 1 – ELEMENTOS DO DESIGN DE INTERAÇÃO

Tema 1: Sistemas interativos em contexto

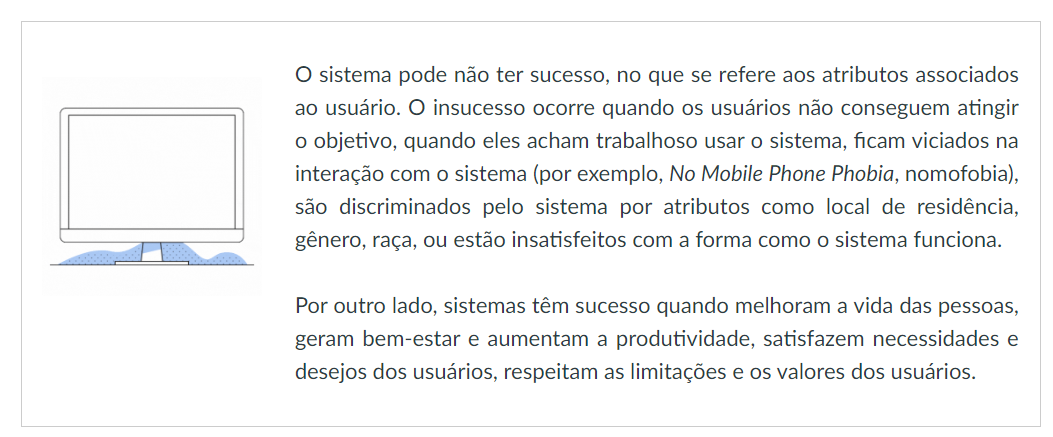
1. [Qual é o papel dos sistemas interativos na atividade humana?](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-1-qual-e-o-papel-dos-sistemas-interativos-na-atividade-humana)

Os seres humanos são caracterizados por projetarem artefatos que lhes ajudem a desempenhar suas atividades. Eles definem qual a forma e a função desses artefatos e os constroem. Isso aconteceu quando uma rocha foi adaptada para ser usada como um machado para cortar uma madeira. Analogamente, também ocorre na construção de sistemas computacionais que são criados para interagirem com seres humanos, os chamados *sistemas interativos*.

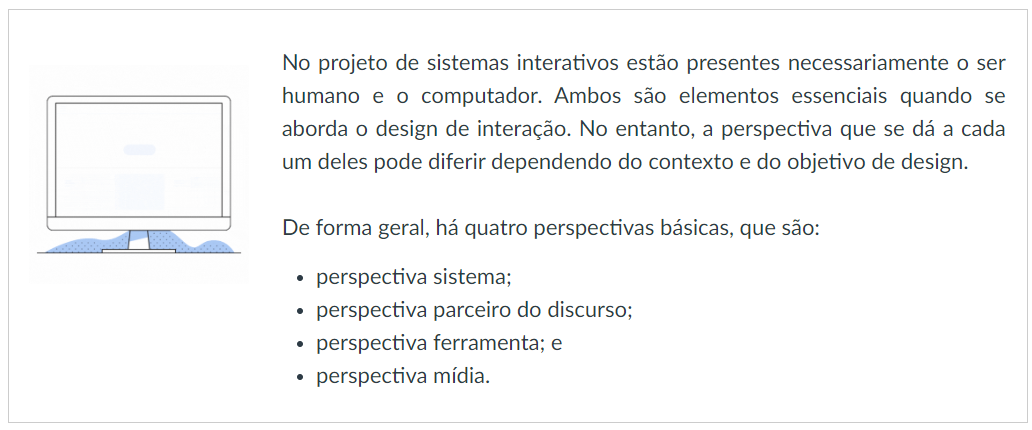
Ao serem usados, os sistemas interativos exercem influência sobre o seu usuário. Atualmente, tais sistemas influenciam amplamente:

1. como as pessoas relacionam entre si, esse é o caso de aplicativos de troca de mensagens, redes sociais e de videoconferência;
2. como as pessoas se informam, esse é o caso de sítios e portais de notícias;
3. Como as pessoas trabalham, prestam serviços, aprendem e ensinam, esse é o caso de sistemas como o Canvas. Enfim, cada vez mais, os sistemas interativos estão mais presentes nos diversos ramos da atividade humana.

No dia a dia, os sistemas interativos têm papel importante em "O que se faz", "Quem faz", "Onde faz", "Como se faz", "Quando faz" e "Por que faz". Isso ocorre porque, em busca de qualidades como eficiência, comodidade e lazer, as pessoas se apropriam dos sistemas, passando a considerá-los e também sendo influenciados pelas dinâmicas daqueles que são levadas a usar.



1. [O ser humano interage com o sistema, por meio do sistema, ou ambos?](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-1-o-ser-humano-interage-com-o-sistema-por-meio-do-sistema-ou-ambos)



Na **perspectiva sistema,** trata-se da interação usuário-sistema como a interação entre dois sistemas. Isso significa dizer que o usuário é percebido como um sistema tal como o computador. Característica marcante dessa perspectiva é o uso de comandos e atalhos que são formas codificadas de comunicação, como os comandos do Linux cd, ls, cp e os atalhos ctrl+c, ctrl+v, F1. Nesse caso, não se emprega a linguagem natural do usuário, mas sim a linguagem mais facilmente entendida pelo sistema. A ênfase da perspectiva sistema é a transmissão de dados. Fatores de qualidade são tempo de interação e número de erros cometido pelo usuário.

Na **perspectiva parceiro do discurso**, torna-se a interação usuário-sistema mais próxima do que é a interação entre seres humanos. Então, o sistema é elevado às características de um ser humano. Para isso, o sistema precisa ser capaz de fazer inferência e usar linguagem natural. Isso ocorre, por exemplo, quando o sistema tenta sugerir conteúdo e ações, fazer inferências e responder ao usuário. A ênfase da perspectiva parceiro do discurso é a conversa usuário-sistema. Fator de qualidade é adequação da resposta provida pelo sistema

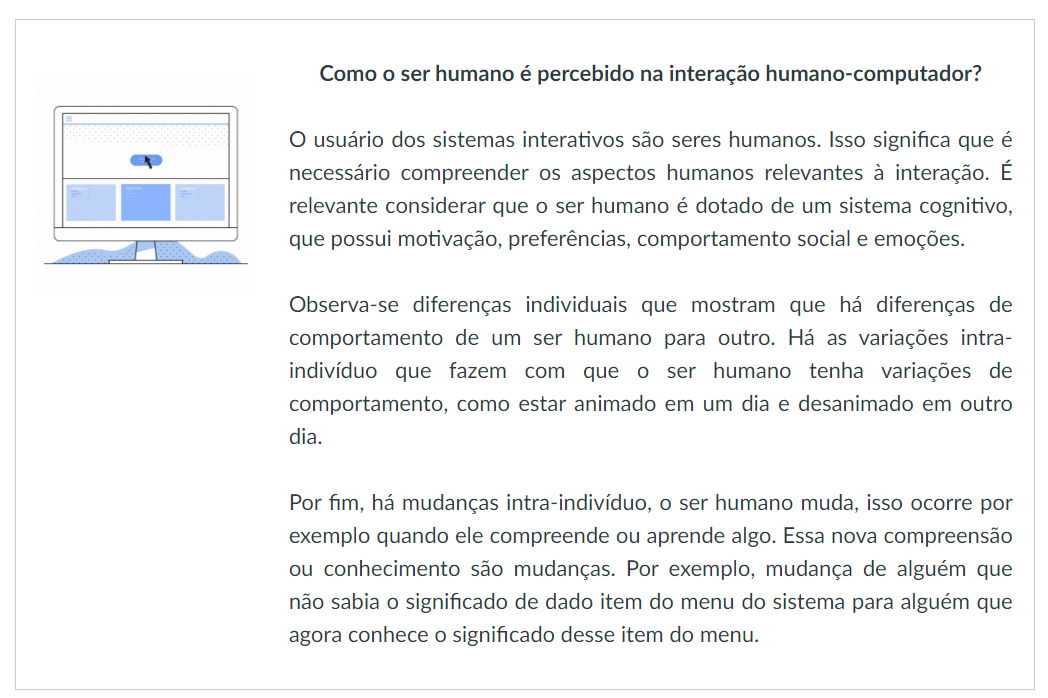
Na **perspectiva do tipo ferramenta**, o sistema é visto como um instrumento para se fazer algo. Nesse caso, tem-se a clara presença de um artefato gerado a partir da interação. Então o usuário interage com o sistema para produzir um artefato. Isso ocorre quando o usuário interage com o editor de texto para gerar um texto editado e formatado. Isso também ocorre quando o usuário interage como uma ferramenta de edição de imagens com o objetivo de gerar uma imagem editada com filtros e efeitos. Em ambos os casos, o sistema busca prover um conjunto de funcionalidades que permita gerar um artefato sofisticado. O resultado da interação é sempre um artefato. A ênfase da perspectiva ferramenta é a manipulação da ferramenta. Fatores de qualidade são funcionalidades relevantes ao usuário e facilidade de uso.

Por fim, **na perspectiva mídia,** o sistema atua como uma mídia, ou seja, meio pelo qual uma pessoa A e uma pessoa B se comunicam. Ao ser um meio de comunicação, o sistema busca um entendimento mútuo entre as pessoas que estão se comunicando. Isso ocorre em sistemas de videoconferência, em sistemas de troca de mensagens e também em redes sociais. A ênfase da perspectiva mídia é a comunicação entre usuários. Fator de qualidade é o entendimento mútuo entre os usuários.

Tema 2: Atributos de qualidade da interação nas perspectivas cognitiva e semiótica

1. [Perspectiva teórica do humano na interação humano-computador](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-perspectiva-teorica-do-humano-na-interacao-humano-computador)

Interação Humano-Computador (IHC), tradução do termo em inglês *Human-Computer Interaction*, é uma disciplina que foca nos sistemas computacionais interativos que são projetados para uso por seres humanos. Em outras palavras, quem interage com os sistemas considerados em IHC são seres humanos. O interesse de IHC centra-se, principalmente, no projeto de sistemas interativos, implementação de sistemas interativos, avaliação de tais sistemas e investigação dos fenômenos relacionados ao seu uso.



Buscando obter uma compreensão científica do ser humano, a área de IHC se baseia em abordagens teóricas. Há abordagens da psicologia experimental que permitem mensurar e modelar o comportamento humano observável e prever o desempenho dele. Há abordagens da psicologia cognitiva aplicada que são centradas nos aspectos cognitivos da interação por detrás do comportamento.

Por fim, há abordagens advindas da linguística que tratam dos signos utilizados na comunicação e metacomunicação por meio de artefatos. Entre as diversas abordagens teóricas tratadas no design de interação, pode-se citar: Leis de Hick-Hyman e de Fitts, Processador Humano de Informação, Princípios da Gestalt, Teoria da Atividade, Cognição Distribuída, Engenharia Cognitiva, Teoria da Ação, Engenharia Semiótica.

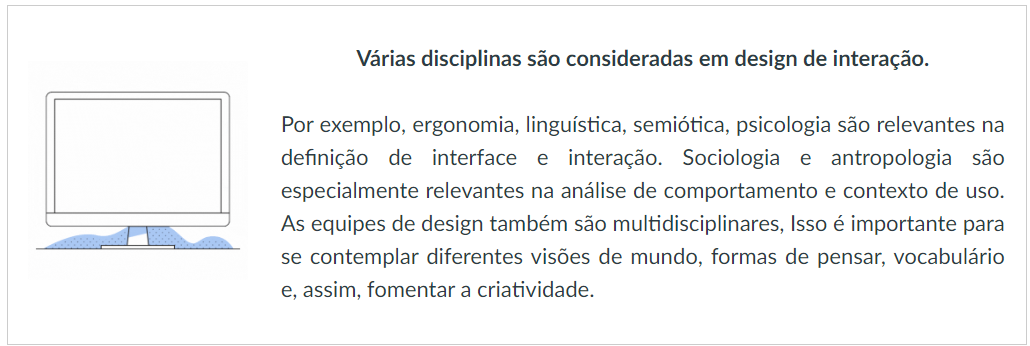
1. [O contexto da experiência do usuário](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-o-contexto-da-experiencia-do-usuario)

O **contexto de uso** inclui o tempo e ambiente físico, social e cultural em que a interação entre o usuário e sistema ocorre. O usuário interage com o sistema por que ele deseja atingir um objetivo. Os objetivos são dependentes do domínio e  podem ser, por exemplo, cadastrar funcionário, cadastrar aluno, etc.

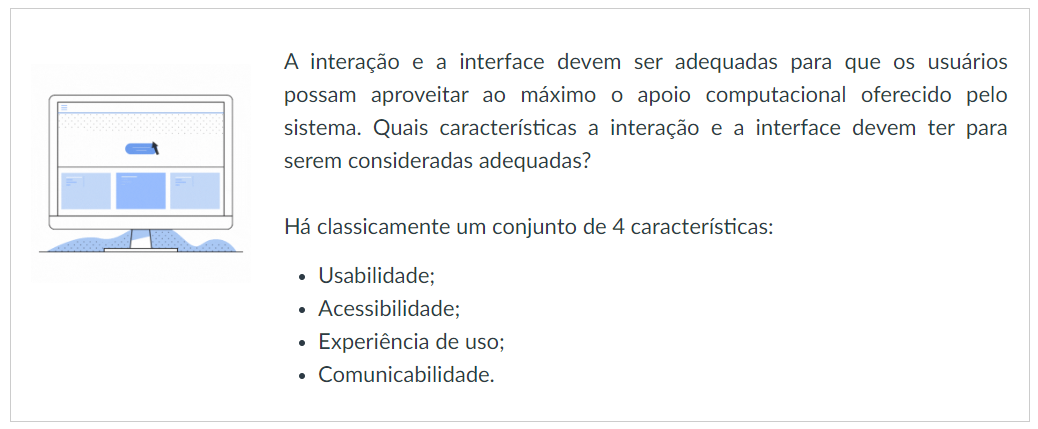
O usuário interage com o sistema por meio de uma **interface**. A interface é o único meio de contato entre o usuário e o sistema. Aquilo que não estiver acessível por meio da interface, não poderá ser usado pelo usuário. A interface é delimitadora na medida em que ela define o que usuário pode fazer, onde pode fazer e em que ordem. A interface também delimita a acessibilidade. Por exemplo, uma interface baseada em texto impõe desafios para que seja usada por alguém que não é alfabetizado. A **interação**, por sua vez, é um processo de manipulação, comunicação, conversa, troca e influência que ocorre durante o uso. Nesse sentido, interagir com o sistema significa prover informações a ele e receber informações dele.

O**contexto de uso** é o palco da Experiência do Usuário (User eXperience, ou UX), que envolve a interface, a interação e aquilo que está no ambiente em que a interação ocorre, incluindo elementos físicos como iluminação e barulho. Projetar a experiência do usuário (UX Design) é, portanto, projetar todos esses elementos. A experiência do usuário vai além do contexto de uso  de um sistema computacional.

A **Interface do Usuário** (User Interface, ou UI) é organizada como uma Arquitetura da Informação (Information Architecture, ou IA) que indica  onde a informação deve estar disposta, qual informação deve estar destacada no campo de visão do usuário, como definir submenus de um menu, etc. Por meio da interface e dentro do contexto de uso ocorre a interação, que é projetada como Design de Interação (Interaction Design, IxD).



1. [Atributos de usabilidade, comunicabilidade e acessibilidade em design de interação](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-atributos-de-usabilidade-comunicabilidade-e-acessibilidade-em-design-de-interacao)



A International Organization for Standardization (ISO) estabelece a norma de Ergonomia de Interação Usuário-Sistema, que na sua parte 11 define o conceito e **usabilidade**. A norma é identificada como ISO 9241-11 (2018) e define usabilidade como: "O grau em que um produto é usado por usuários específicos, para atingir objetivos específicos com eficácia, eficiência e satisfação em um contexto de uso específico".

Destaca-se, "usuário específico", "objetivo específico" e "contexto de uso específico". Ou seja, o sistema é feito para usuário, contexto e objetivos que não são gerais, mas específicos de uma dada situação. O usuário deve obter eficácia, eficiência e satisfação ao usar o sistema. Nesse contexto, eficácia é alcançar os objetivos corretamente, eficiência é a quantidade de recursos físicos e cognitivos necessários ao usuário para alcançar os objetivos e satisfação é o quanto usuários perceberam a interface e a interação como favorável aos seus objetivos.

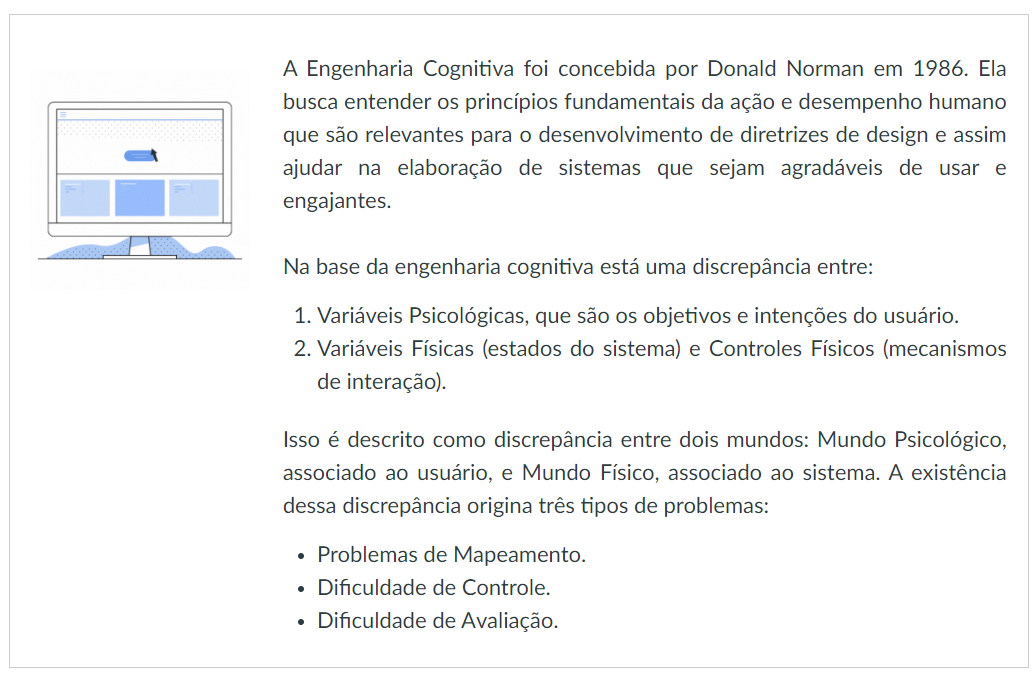
A norma ISO/IEC 25010:2011, que trata da qualidade de produto de software**,** apresenta atributos de usabilidade enquanto atributo de um software. Ela destaca os seguintes fatores que precisam ser considerados: capacidade de reconhecimento da adequação, facilidade de aprendizado, operabilidade, proteção contra erro do usuário, estética da interface de usuário e acessibilidade.

**Acessibilidade** é um atributo que busca permitir ao usuário interagir com o sistema sem que a interface imponha obstáculos. Isso significa permitir que mais pessoas possam perceber, compreender e utilizar o sistema para usufruir do apoio computacional oferecido por ele. Naturalmente, limitações físicas, mentais ou de aprendizado aumentam as chances do usuário encontrar barreiras que dificultam ou impedem a interação.

A **experiência de uso**, por sua vez, é um atributo fortemente influenciado pelas percepções do usuário. Cabe pensar, o que o usuário experimenta ao interagir com o sistema? A avaliação da experiência de uso engloba aspectos como emoções/sentimentos, imersão e engajamento do usuário com o sistema durante o uso. Além da interface e interação, todos os elementos do contexto de uso afetam a experiência de uso.

Por fim, **comunicabilidade** é a capacidade da interface de comunicar ao usuário a lógica que rege o seu funcionamento. Para ser dotada desse atributo, a interface deve ser efetiva na comunicação dos passos necessários para que o usuário alcance o objetivo por meio dela.

1. [A interação pela lente cognitiva](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-a-interacao-pela-lente-cognitiva)



O **Problema de Mapeamento** envolve analisar se as variáveis psicológicas são mapeadas adequadamente em variáveis físicas. Ou seja, a interface é clara? O usuário entende o que nela aparece?

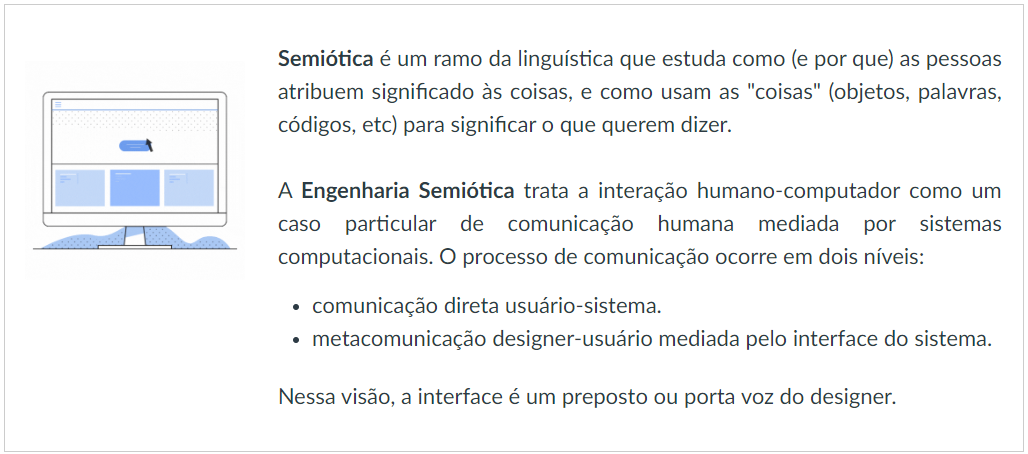
A **Dificuldade de Controle** envolve analisar o quão fácil é controlar as variáveis físicas para se atingir o objetivo desejado.

A **Dificuldade de Avaliação** envolve analisar o quão fácil é avaliar o resultado de variações nas variáveis físicas ou o resultado de manipulações na interface. Ou seja, o usuário compreende a reação e feedback provido pelo sistema?

A ocorrência dos problemas está relacionado ao fato de que a forma como o designer percebe o sistema é diferente da forma como o usuário percebe o sistema. Isso pode ser descrito por meio de três componentes: modelo de design, imagem do sistema e modelo do usuário. O modelo de design é um modelo conceitual do sistema tal como concebido pelo designer. Imagem do sistema é o sistema físico construído com base no modelo conceitual de design. O modelo do usuário é um modelo conceitual construído pelo usuário durante sua interação com o sistema.

A Engenharia Cognitiva é reinterpretada pela Teoria da Ação, na qual a discrepância entre o mundo psicológico e mundo físico é interpretada como a distância de travessia entre de golfos: golfo de execução e golfo de avaliação.

1. [A interação pela lente semiótica](https://pucminas.instructure.com/courses/68910/pages/unidade-1-tema-2-a-interacao-pela-lente-semiotica)



Engenharia Semiótica é a engenharia de significados e de representações através da **escolha de signos**. Signo é qualquer coisa que tem um significado para alguém. É uma coisa que está no lugar de outra coisa, para alguém, sob determinados aspectos ou capacidades. Significados são funções, comandos, resultados e erros que deseja expressar. Representações incluem janelas, botões, menus, páginas.

O designer é percebido como o emissor de uma mensagem que, por meio de um canal, chega ao usuário, que é o receptor. A mensagem que está codificada na interface do sistema chega ao usuário. Por exemplo, ao colocar o símbolo de uma impressora na interface, a mensagem que o designer está enviando ao usuário é que ele pode clicar naquele local para imprimir. Para isso ocorrer, tanto designer quanto usuário devem conhecer tal símbolo. Por isso, designer e usuário compartilham códigos em comum, que são palavras, ícones, figuras que ambos entendem.

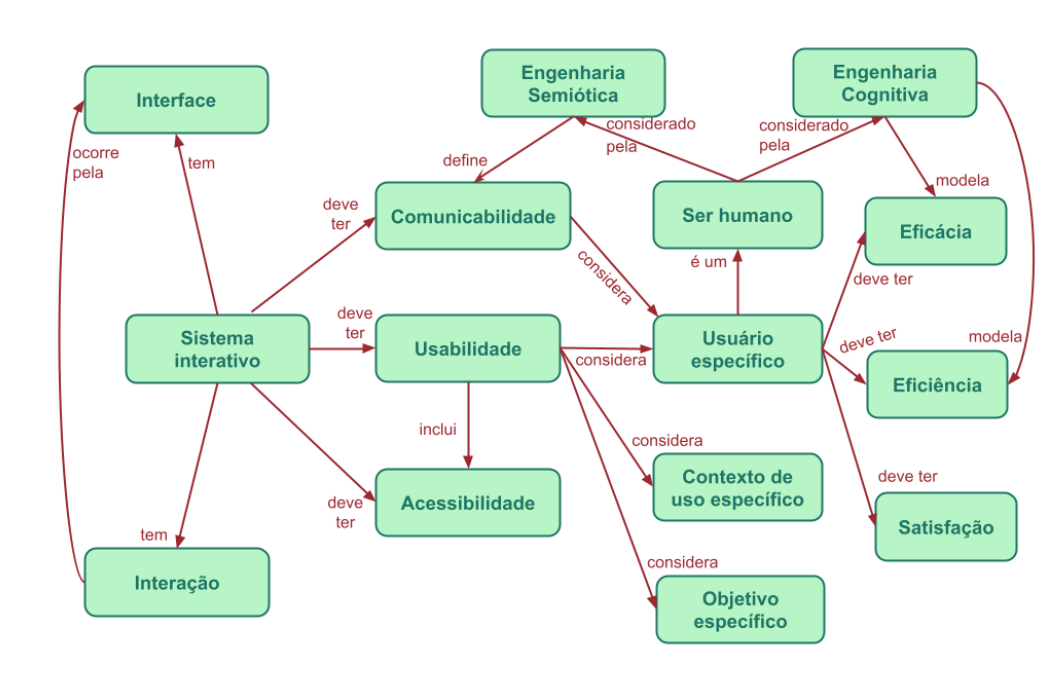
Há três tipos de signos: estáticos, dinâmicos e metalinguísticos.

**Signos estáticos** comunicam o seu significado integral em telas fixas (estáticas) do sistema.

**Signos dinâmicos** comunicam seu significado integral em sequência de telas ou com o tempo (dinamicamente), sendo que estaticamente não comunicam todo o seu significado.

**Signos metalinguísticos** são signos estáticos ou dinâmicos que explicam e ilustram outros signos estáticos ou dinâmicos.

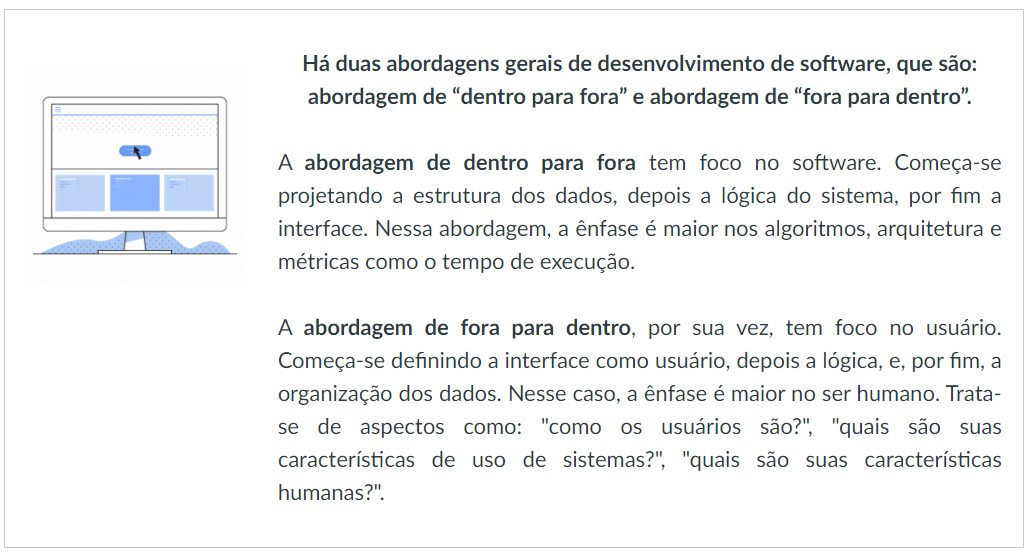
Há várias contribuições da Engenharia Semiótica para design centrado na comunicabilidade, incluindo o MoLIC, que é a Linguagem de Modelagem da Interação como Conversa; MAC, que é avaliação da comunicabilidade pela recepção da metamensagem; MIS, que é avaliação da comunicabilidade pela emissão da metamensagem.



- Unidade 2 – PROCESSO DE DESIGN DE INTERAÇÃO

Tema 1: Perspectivas de design

1. [Projetar de modo independente do usuário, projetar centrado no usuário ou projetar com o usuário?](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-1-projetar-de-modo-independente-do-usuario-projetar-centrado-no-usuario-ou-projetar-com-o-usuario)



**Projetar de modo “independente” do usuário é projetar sem considerar o usuário**.

O usuário não é conhecido e nem consultado durante o processo de construção do software. Nesse caso, após o software pronto, o usuário terá que aprender a usar o sistema “da forma como ele foi feito” e precisará “aceitar o que o sistema permite o que ele não permite”. Não é uma abordagem adequada para o projeto de sistemas interativos.

**Projetar “para” o usuário é o "Design Centrado no Usuário"**.

Nessa abordagem há o foco no usuário: “quem são?”, “quais seus objetivos?”, “quais suas preferências?”. Nesse processo, emprega-se métricas observáveis: há experimentos (estudos empíricos) e alguns usuários usam simulações ou protótipos do sistema. De forma geral, o processo contém quatro atividades:

* identificar necessidades e definir requisitos;
* fazer design ou redesign do sistema;
* construir uma versão interativa;
* e avaliar o artefato feito.

Sendo um processo interativo, cada uma dessas atividades pode ser executada mais de uma vez, em diferentes estágios do desenvolvimento do software. Projetar centrado no usuário inclui: conhecer as características dos usuários e considerá-las, pensar no que os usuários querem fazer em vez do que a tecnologia pode fazer, projetar novas maneiras de conectar pessoas, respeitar a diversidade de usuários.

**Projetar “com” o usuário é o "Design Participativo"**.

Ele é geralmente definido como "projetar com o usuário em vez de projetar para o usuário". Isso significa ampliar o design centrado no usuário de modo a:

* dar voz àqueles que são tratados como invisíveis;
* adequar o sistema ao contexto das pessoas;
* empregar ferramentas e técnicas de design que ajudem as pessoas a se expressarem (*design-by-doing*);
* promover visões alternativas acerca da tecnologia (*design-in-use*); e,
* empregar práticas democráticas e regras para equidade.

O Design Centrado no Usuário e do Design Participativo são processos iterativos, pois uma mesma atividade pode ser feita mais de uma vez para refinar a compreensão do problema de design e proposta de uma solução de design. Quando problemas/dificuldades encontrados são corrigidos.

1. [A tarefa de projetar consiste em empregar a técnica ou exercer a criatividade?](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-1-a-tarefa-de-projetar-consiste-em-empregar-a-tecnica-ou-exercer-a-criatividade)

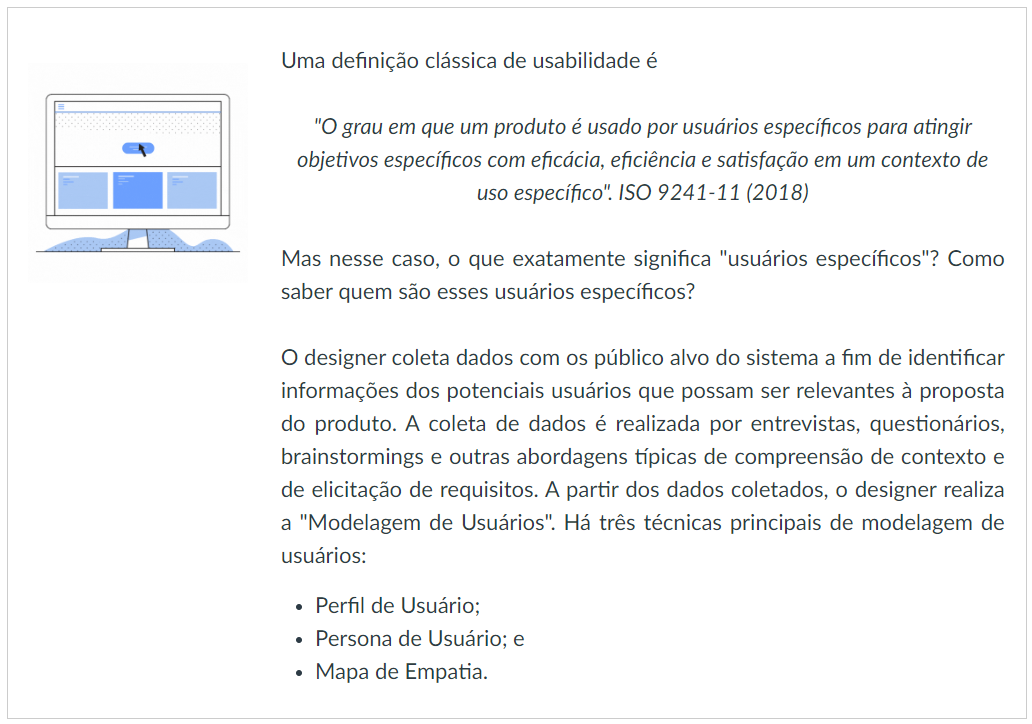
Projetar (to design) artefatos é projetar produtos artificiais. Artefatos, como sistemas interativos, não surgem espontaneamente na natureza. Eles são fruto da inteligência e do trabalho humano. Os seres humanos os constroem com um propósito em mente. Na perspectiva de design de interação, há duas abordagens de projeto de sistemas interativos: **Racionalismo Técnico**e **Reflexão em Ação**.

A corrente de **Racionalismo Técnico** defende que há problemas e soluções conhecidas e que os métodos de solução bem definidos a priori. Nesse caso, o papel do designer é enquadrar uma situação em um tipo geral de problema cuja forma de solução seja conhecida. Deve-se seguir princípios (ex.: Gestalt), Guias (ex.: componentes WIMP), Regras (ex.: regras de ouro) entre outros direcionamentos de solução. Por exemplo, seguir a Recomendação Ergonômica de "identificar visualmente os campos de preenchimento obrigatório" é uma abordagem de racionalismo técnico.

A corrente **Reflexão em Ação** defende que há problemas e soluções únicos e que os métodos e ferramentas servem para auxiliar o aprendizado do designer sobre problema e solução únicos. Dessa forma, o designer busca aprender sobre o problema em questão e a solução sendo concebida. A reflexão em ação ocorre por meio da "conversa com materiais”, pela qual o designer analisa os artefatos e se questiona “e se eu definir isso deste jeito?”, “posso utilizar essa mesma ideia em outro lugar?”, "isso não ficou bom, mas se eu tentar dessa outra forma?" Por exemplo, decidir qual cor usar no sistema é em grande medida uma tarefa de reflexão em ação. Nesse tipo de situação, há muito espaço para a criatividade do designer.

Tema 2: A prática de design

1. [Modelagem de usuários e suas necessidades](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-modelagem-de-usuarios-e-suas-necessidades)

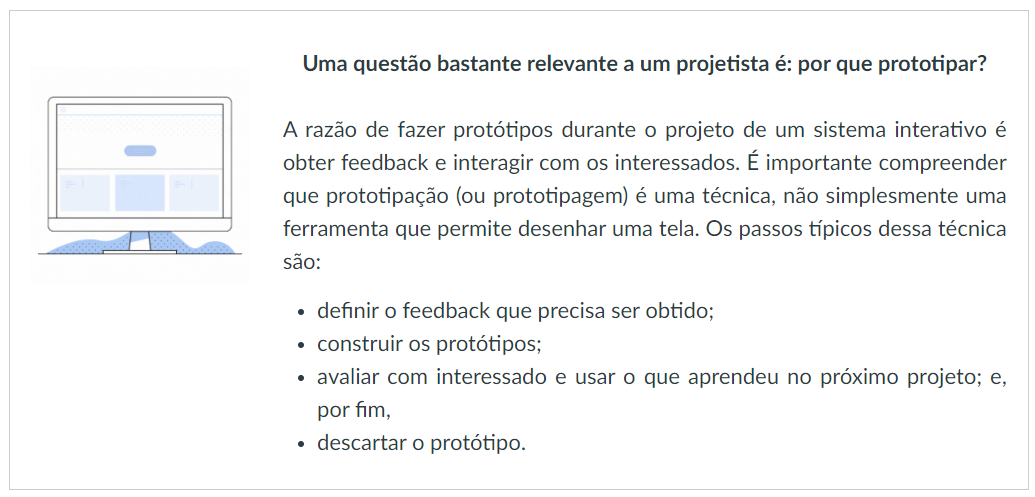


Um **Perfil de Usuário**é uma descrição detalhada das características dos usuários. Um perfil é um agregado. Por meio de um perfil, pode-se descrever um conjunto de usuários. Para gerar o perfil, deve-se agrupar usuários que possuem características semelhantes como idade (criança, jovem, adulto, terceira idade etc.), experiência (leigo/novato,  especialista), atitudes (gosta de tecnologia, não gosta de tecnologia), tarefas principais (compra, venda). Algumas características são destacadas e outras abstraídas dependendo do contexto do sistema.

Uma**Persona** é um personagem fictício baseado em um grupo de usuários reais e que descreve um usuário típico. À persona atribui-se várias características como identidade (nome, sobrenome, idade, foto), objetivos (quais são os objetivos desta persona?), status (primária, secundária, outro stakeholder) e habilidades (Qual é a especialidade da persona?).

Um**Mapa de Empatia** descreve um usuário por meio de atributos associados ao que ele fala, pensa, faz e sente no contexto do sistema. A dimensão "pensa" é o que o usuário acha, mas não está disposto a falar. A dimensão "sente" são sentimentos do usuário no contexto do sistema. A dimensão “fala” é aquilo que o usuário acredita e que, caso necessário, diria. Por fim, a dimensão “faz” são atitudes/ações do usuário relevantes ao contexto do sistema. O mapa de empatia é descrito em uma imagem, com uma representação do usuário ao centro e as quatro dimensões, como quadrantes, em volta.

1. [Abordagens de prototipagem](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-abordagens-de-prototipagem)



**De quem obter feedback?**

De colegas designers com os quais pode-se verificar: "este produto atende aos requisitos?". De clientes, aos quais, por meio do protótipo, pode-se questionar: “este produto atende aos requisitos? qual variação você prefere?” E, também, obtém-se feedback de usuários com questionamentos como: “funciona? está claro? consegue usar efetivamente? o que mudar?”.

* **Wireframe** é uma forma de prototipagem em telas de baixa fidelidade, que definem a arquitetura de informação e o*layout*, mas não incluem cor e muitos detalhes.
* **Mock-up** é uma forma de prototipagem em "maquetes", que são telas de alta fidelidade, que inclui detalhes estéticos e de funcionalidades.
* **Storyboards,** por sua vez, é uma forma de prototipagem que gera protótipos que se assemelham a diagramas de fluxo, que são guias que ilustram os detalhes importantes da sequência de interação.
* **Prototipação em Vídeo** (*vídeo prototyping*), por sua vez, pode-se representar uma dinâmica mais complexa, mas sem interação. Eles são usados em sistemas ubíquos e interação multimodal, que vão além da interface gráfica, como hologramas. Pode-se manter o vídeo e adicionar diferentes intervenções de projeto a fim de compará-las.
* Por fim, o **protótipo em papel**(*paper prototyping*) permite flexibilidade para ajustar, redesenhar e inserir novos itens rapidamente e facilmente, mas há baixa robustez para ideias complexas, ou seja, sistemas que têm muitos detalhes.

**É importante diferenciar prototipação horizontal e prototipação vertical.**

**Prototipação horizontal** tem menor profundidade e maior largura com relação ao número de características do sistema que são contempladas no protótipo. É mais usada quando o desenvolvimento do sistema está em um estágio inicial.

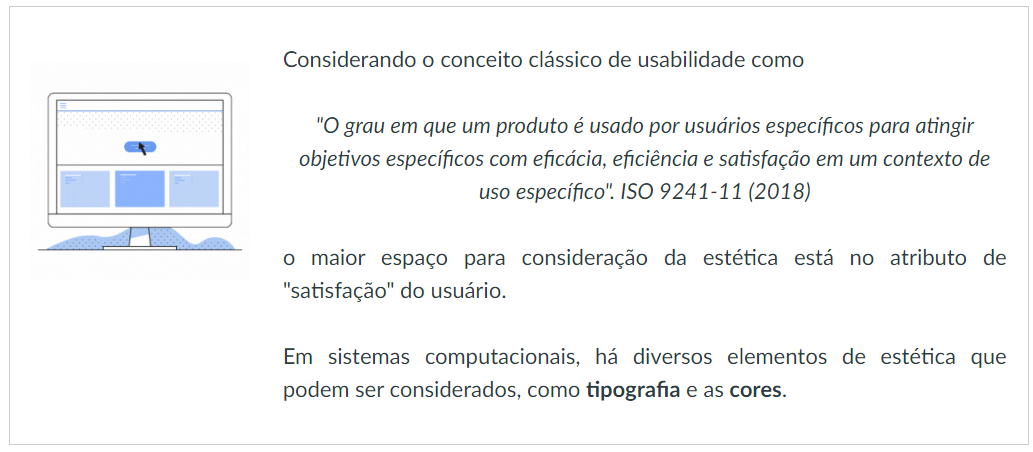
Por outro lado, a **Prototipação vertical**utiliza um menor número de características, mas as utilizadas são abordadas em maior profundidade. Prototipação vertical é mais usada quando o desenvolvimento do sistema está em um estágio mais adiantado.

**Também é importante diferenciar prototipação de baixa-fidelidade e de alta fidelidade.**

**Prototipação de Baixa-Fidelidade** (*Low Fidelity, Lo-Fi*) é uma representação artística com poucos detalhes. Esse tipo de protótipo são mais focados em ideias amplas e iniciais, têm baixo custo, desenvolvimento rápido e é descartado rapidamente.

**Prototipação de Alta-Fidelidade** (*High Fidelity, Hi-Fi*), por sua vez, é uma representação com muitos detalhes. O protótipo de alta fidelidade se assemelha ao produto final, útil na avaliação de detalhes. No entanto, tem alto custo, demanda muito tempo para ser feito.

1. [Cores, fontes e sua harmonização](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-cores-fontes-e-sua-harmonizacao)



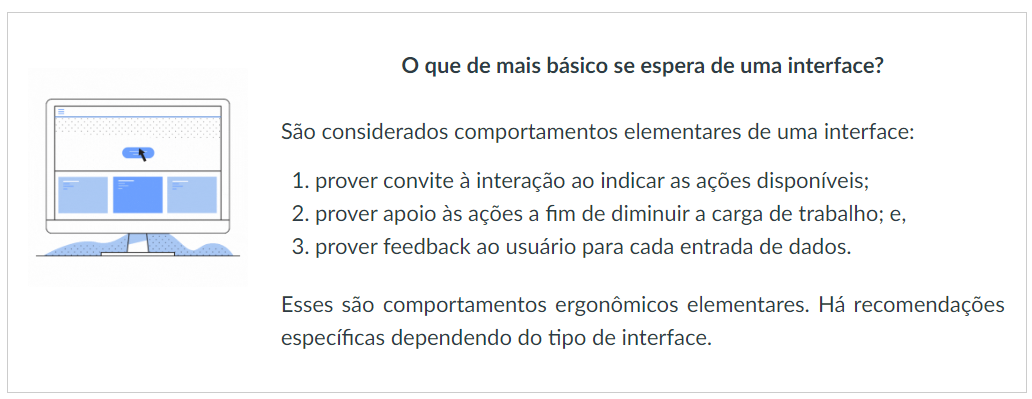
A **tipografia**geralmente é definida em termos de família (com serifa, sem serifa e decorativa), época (Clássica, contemporânea e moderna) e estilo (regular, negrito e itálico). Pode-se fazer o contraste entre família, época e estilo e também com tamanhos diferentes em que o texto de dada tipografia é exibido ao usuário.

**Cores** também desempenham um papel importante na estética. Há as cores quentes que tendem a ser mais luminosas e colocam em um plano à frente a imagem que ela preenche, servindo para destaque, mas não servindo, por exemplo, para *background*. Há cores frias que tendem a ser menos luminosas e colocam em um plano atrás a imagem que ela preenche, essas podem se adequar ao *background*. A escolha de cor pode seguir também o Círculo Cromático (ou roda cromática). Por meio dessa roda, pode-se escolher diversos padrões de cores.

* **Cores ditas complementares**são duas cores opostas no círculo cromático. Elas exibem entre si alto contraste e destaque uma da outra.
* **Cores ditas análogas** são três cores vizinhas no círculo cromático, sendo uma combinação harmônica geralmente encontrada na natureza.
* **Cores no esquema tríade**são 3 cores equidistantes no círculo cromático. Elas permitem obter um design colorido e ainda equilibrado.
* **Cores em divisão complementar** são 3 cores, sendo uma cor principal e uma cor de cada lado da cor que seria seu complemento (oposto) no círculo cromático. Se três cores são do esquema divisão complementar, duas cores têm alto contraste em relação à principal, mas o contraste não é tão grande quanto o da cor complementar.
* No **esquema quadrado**, tem-se dois pares de cores complementares, geralmente são cores balanceadas.
* Por fim, no **esquema retângulo**, tem-se quatro cores, sendo dois pares complementares e que também estão em dois pares análogos.

A escolha de cores também pode se dar por meio de escalas padrão.Como ocorre na **escala divergente**, que vai de um polo de cor a outro. A **escala sequencial,** por sua vez, representa gradações de um dado atributo como gradações de uma cor. A**escala qualitativa** reúne cores diferentes e é usada para representar elementos diferentes. Há também **cores com semânticas convencionais**, como é o caso da escala de Manchester, que é muito usada em Hospitais. Por ela, o vermelho indica uma situação de emergência e o azul indica uma situação não urgente.

1. [Componentes e recomendações ergonômicas](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-componentes-e-recomendacoes-ergonomicas)



**Graphical user interfaces** (GUIs) geralmente definidas pelo padrão Janelas, Ícones, Menus e Apontadores (WIMP, do inglês *windows*, *icons, menus, and pointing devices*) são muito utilizadas atualmente. Nesse tipo de interface tem-se o conceito de manipulação direta, que significa que o usuário pode clicar, arrastar, excluir itens que aparecem visualmente na interface. Essas interfaces seguem padrão de "reconhecer, em vez de recordar", que significa que o usuário não precisa lembrar o significado de um componente que aparece na interface, basta olhar o componente que se reconhece o que significa.

O **projeto de ícones** tem grande importância ao fazer sistemas que o usuário é capaz de reconhecer. Para que isso seja possível, três princípios devem ser seguidos:

1. metáfora, que consiste em transferir conhecimento de um domínio aplicando ele em outro domínio de modo facilitar a compreensão correta do seu significado;
2. mapeamento direto, que consiste em criar uma imagem parecida com o que o ícone pretende representar; e,
3. convenção, que significa que o ícone deve manter-se equivalente e consiste em um mesmo sistema e em diferentes aplicações e sistemas.

Outros elementos importantes de sistemas interativos são **mensagens de alerta e erros**. Há recomendações que devem ser seguidas nesse contexto:

* Evitar linguagem ameaçadora ou alarmante ("Deseja matar a tarefa?" versus "Deseja encerrar a tarefa?");
* Não usar duplas negativas, pois podem ser ambígua ("Não foi possível não salvar o arquivo");
* O sistema deve “assumir a culpa” ("comando ilegal" versus "comando não reconhecido");
* Não usar mensagens escritas exclusivamente em CAIXA ALTA, pois passa a ideia de que está gritando com o usuário.

**No caso de objetos de seleção**, como *listbox*, é importante ordenar os itens que são exibidos. A seleção por *radiobutton* deve ser sempre de itens mutuamente exclusivos e a seleção por *checkbox* deve sempre permitir a seleção de mais de um item.

**No caso de objetos de apresentação**, como texto, deve-se atentar para:

* não usar hifenização no final das linhas;
* usar no máximo 60 caracteres por linha;
* usar espaçamento entre parágrafos;
* usar bordas implícitas ou explícitas;
* evitar textos em letras exclusivamente maiúsculas;
* fontes com serifa são usadas em textos longos, em títulos e rótulos curtos, deve-se empregar fontes sem serifa.

Em caso de **preenchimento de formulários** (*Form Fill*), é necessário a diferenciação visual entre rótulos e dados, entre dados default e dados definidos pelo usuário, entre os campos de preenchimento opcional e os obrigatórios. Além disso, deve-se prover feedback sobre quando os dados são salvos (enviados) e se alguma ação é necessária ou não. Também deve-se ordenar a sequência de movimentação da tecla Tab, pois facilita a navegação do usuário no formulário.

Sobre o **tipo de interface**, há alguns pontos a destacar quanto a interfaces Web e para dispositivos móveis. No caso de **interfaces Web**, é necessário maior atenção à arquitetura da informação do site, à estrutura e a organização do site, à indicação de como as pessoas se movem no site, à indicação de como as pessoas descobrem o que há no site e, também, à indicação de como as pessoas descobrem onde estão. No caso de **dispositivos móveis**, é importante minimizar os níveis de navegação, minimizar a quantidade de conteúdo exibido e reduzir a quantidade de entradas de dados requeridas aos usuários. Lembre-se também que conexões não são estáveis e que a bateria é limitada, então a aplicação deve ser tolerante a interrupções para que o usuário não perca dados da interação.

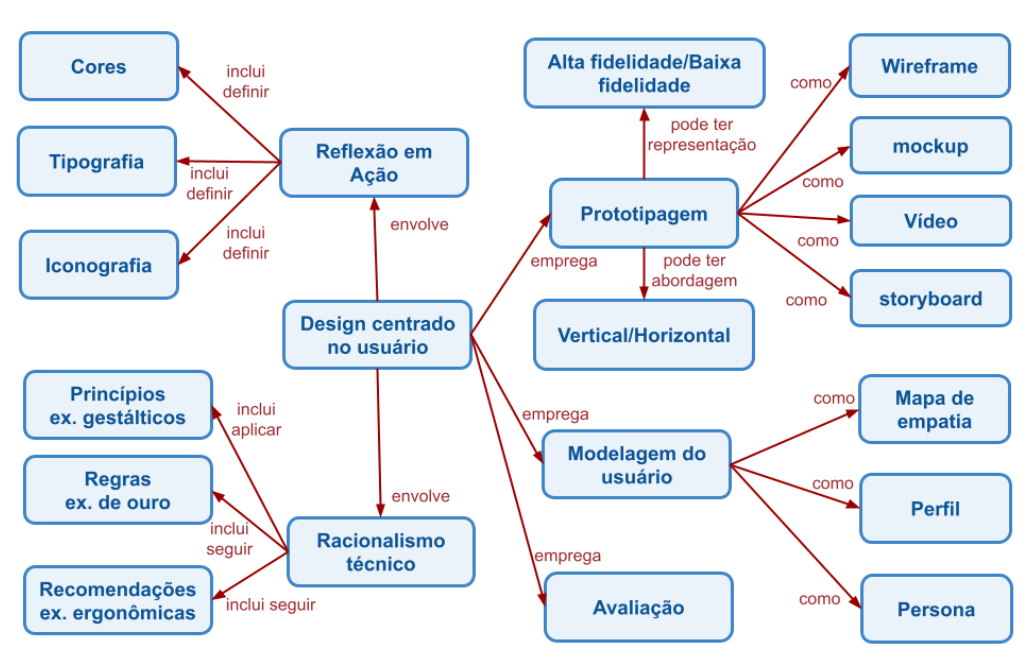
1. [Princípios e regras de design](https://pucminas.instructure.com/courses/68910/pages/unidade-2-tema-2-principios-e-regras-de-design)

**Os princípios gestálticos são:**

* **Proximidade:**as entidades visuais que estão próximas umas das outras são percebidas como um grupo ou unidade;
* **Boa continuidade (alinhamento):**traços contínuos são percebidos mais prontamente do que contornos que mudem de direção rapidamente;
* **Simetria:** objetos simétricos são mais prontamente percebidos do que objetos assimétricos;
* **Similaridade:** objetos semelhantes são percebidos como um grupo;
* **Destino comum:**objetos com a mesma direção de movimento são percebidos como um grupo;
* **Fecho:** a mente tende a fechar contornos para completar figuras regulares, “completando as falhas” e aumentando a regularidade;
* **Região comum:** objetos dentro de uma região especial confinada são percebidos como um grupo;
* **Conectividade:**objetos conectados por traços contínuos são percebidos como relacionamento.

**As oito regras de ouro são:**

1. Perseguir a Consistência;
2. Fornecer Atalhos;
3. Fornecer Feedbacks Informativos;
4. Marcar o Final dos Diálogos;
5. Fornecer Prevenção de Erros e Forma Simples de Corrigi-los;
6. Permitir a Reversão de Ações;
7. Fornecer Controle de Iniciativa do Usuário
8. Reduzir a Carga de Memória de Trabalho.



- Unidade 3 – AVALIAÇÃO DA QUALIDADE DE SISTEMAS INTERATIVOS

Tema 1: Abordagens de avaliação

* 1. [Por que, quando e quem participa da avaliação de sistemas interativos?](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-1-por-que-quando-e-quem-participa-da-avaliacao-de-sistemas-interativos)

O primeiro contraste é entre Avaliação Formativa e Avaliação Somativa. A **Avaliação Formativa** é realizada durante o processo de projeto de sistema. Nesse caso, o designer faz uma avaliação com o propósito de obter um feedback que norteie o processo de design. É uma forma de saber se está seguindo o caminho certo. A **Avaliação Somativa**, por sua vez, é realizada no final, quando o sistema já está feito. É pela avaliação somativa que o designer verifica se os objetivos de design foram atingidos. Nesse caso, avalia-se se o sistema atende ao propósito a que ele se destina e se atende ao que o usuário precisa. Então, é uma avaliação final, antes de se implantar o sistema.

O segundo contraste é entre Avaliação por Inspeção, Avaliação por Observação e Avaliação por Investigação. A **Avaliação por Inspeção** não envolve a participação de usuários. Um avaliador tenta se colocar no lugar do usuário enquanto examina (ou inspeciona) o sistema. Essa abordagem permite identificar problemas que os usuários podem vir a ter quando interagirem com o sistema e permite propor soluções para tais problemas.  Na **Avaliação por Observação**, os usuários participam da avaliação.  Pode existir um sistema pronto e instalado ou apenas uma representação abstrata dele. O uso do sistema pode se dar em contexto ou em laboratório. Por fim, na **Avaliação por Investigação**, há a participação do usuário. Essa abordagem envolve o uso de técnicas como questionários, entrevistas, grupos de foco e estudos de campo. Ela permite ao avaliador ter acesso, interpretar e analisar concepções, opiniões, expectativas.

Tema 2: Métodos de avaliação

1. [Inspeção do percursos cognitivo](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-inspecao-do-percursos-cognitivo)

A **Avaliação Heurística** é um método de avaliação por inspeção. Ela se baseia em um conjunto de 10 heurísticas, sendo que cada heurística é uma característica que se deseja que o sistema tenha. Para execução da avaliação, deve-se seguir o protocolo de preparação, coleta de dados, interpretação, consolidação e relato dos resultados. As 10 Heurísticas de Nielsen são:

1. Visibilidade do estado do sistema
2. Correspondência Sistema - Mundo Real
3. Controle e Liberdade do Usuário
4. Consistência e Padronização
5. Reconhecimento em Vez de Memorização
6. Prevenção de Erros
7. Flexibilidade e Eficiência de Uso
8. Projeto Estético e Minimalista
9. Ajudar os Usuários com os Erros
10. Ajuda e Documentação

A **Avaliação do Percurso Cognitivo** é um método de avaliação por inspeção. Ela se baseia em um conjunto de 4 questões, sendo que cada questão é um aspecto cognitivo do usuário que deseja investigar se o sistema contempla. Para execução da avaliação, deve-se seguir o protocolo de preparação, coleta de dados, interpretação, consolidação e relato dos resultados. As 4 questões do método de Avaliação do Percurso Cognitivo são:

1. O usuário tentaria alcançar o efeito desejado?
2. O usuário vai notar que a ação correta está disponível?
3. O usuário conseguirá associar a ação correta com o efeito que está tentando atingir?
4. O usuário perceberá que está progredindo em direção à conclusão da tarefa?
5. [Inspeção semiótica](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-inspecao-semiotica)

O **Método de Inspeção Semiótica (MIS)** é um método de avaliação por inspeção. Ele avalia a comunicabilidade considerando a emissão da metacomunicação. Para cada tipo de signo, o avaliador inspeciona a interface e a documentação disponível para o usuário. O protocolo do MIS consiste em:

a) inspeção da metacomunicação de recursos de ajuda e documentação do sistema;  
b) inspeção da metacomunicação em signos estáticos do sistema;  
c) análise da metacomunicação em signos dinâmicos do sistema;  
d) confrontar e comparar os resultados das etapas anteriores;  
e) análise da qualidade da metacomunicação.

Os elementos a serem inspecionados são:

1. quem você, usuário, é
2. quer ou precisa fazer
3. de que maneiras prefere fazer
4. Este, portanto, é o sistema que projetei para você
5. a forma como você pode ou deve utilizá-lo
6. alcançar uma gama de objetivos

O**Método de Avaliação da Comunicabilidade** **(MAC)**é um método de avaliação por observação. Ele avalia a comunicabilidade pela recepção da metacomunicação do designer codificada na interface. As rupturas de comunicabilidade são situações em que o usuário não compreende o que é emitido pelo designer ou não compreende como expressar seus próprios significados. Elas são identificadas como 13 etiquetas que colocam palavras na “boca do usuário”, simulando o que faria sentido o usuário ter dito diante do problema de comunicação que estava experimentando. As etiquetas são:

1. “Cadê?”
2. “E agora?”
3. “O que é isto?”
4. “Epa!”
5. “Onde estou?”
6. “Ué, o que houve?”
7. “Por que não funciona?”
8. “Assim não dá.”
9. “Vai de outro jeito.”
10. “Não, obrigado!”
11. “Pra mim está bom.”
12. “Socorro!”
13. “Desisto”
14. [Avaliação da acessibilidade](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-avaliacao-da-acessibilidade)

Segundo WCAG, para que um sistema Web esteja aderente às diretrizes de acessibilidade, é necessário:

**1°Princípio: Perceptível**  
    1.1 Fornecer alternativas em texto para todo o conteúdo não textual;  
    1.2 Fornecer alternativas para multimídia dinâmica ou temporal;  
    1.3 Criar conteúdo que possa ser apresentado de diferentes formas sem perder informação ou estrutura;  
   1.4 Facilitar aos usuários a audição e a visão dos conteúdos nomeadamente através da separação do primeiro plano em relação ao plano de fundo.

**2° Princípio: Operável**  
     2.1 Fazer com que toda a funcionalidade fique disponível a partir do teclado;  
     2.2 Proporcionar aos usuários tempo suficiente para lerem e utilizarem o conteúdo;  
     2.3 Não criar conteúdo de uma forma que se sabe que pode causar convulsões;  
     2.4 Fornecer formas de ajudar os usuários a navegar, localizar conteúdos e determinar o local onde estão.

**3° Princípio: Compreensível**  
     3.1 Tornar o conteúdo textual legível e compreensível;  
     3.2 Fazer com que as páginas da Web apareçam e funcionem de forma previsível;  
     3.3 Ajudar os usuários a evitar e a corrigir os erros.

**4°Princípio: Robusto**  
     4.1 Maximizar a compatibilidade com os agentes de utilizador atuais e futuros, incluindo as tecnologias de apoio.

1. [Teste de usabilidade](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-teste-de-usabilidade)

Dessa forma, durante um teste de usabilidade, para cada tarefa realizada por cada participante, é possível medir: o grau de sucesso da execução, o total de erros cometidos, quantos erros de cada tipo ocorreram, quanto tempo foi necessário para concluí-lo. A forma como a observação e a medição ocorre é definida pelo designer, para cada situação. Pode consistir desde uma observação visual, feita por uma pessoa, até instrumentalização com câmeras, microfones e ferramentas de clickstream e eyetracking.

Na **etapa de preparação para o teste**, é necessário definir as tarefas que os participantes executarão, definir o perfil dos participantes e recrutá-los, preparar o material para observar e registrar o uso, além de executar o teste-piloto daquilo que será usado durante a sessão de uso. A **etapa de coleta de dados** consiste em observar e registrar a performance durante as sessões de uso. A **etapa de interpretação** **e consolidação dos resultados** consiste em reunir, contabilizar e sumarizar os dados que foram coletados nas seções de uso. Por fim, relata-se os resultados.

1. [Investigação da usabilidade](https://pucminas.instructure.com/courses/68910/pages/unidade-3-tema-2-investigacao-da-usabilidade)

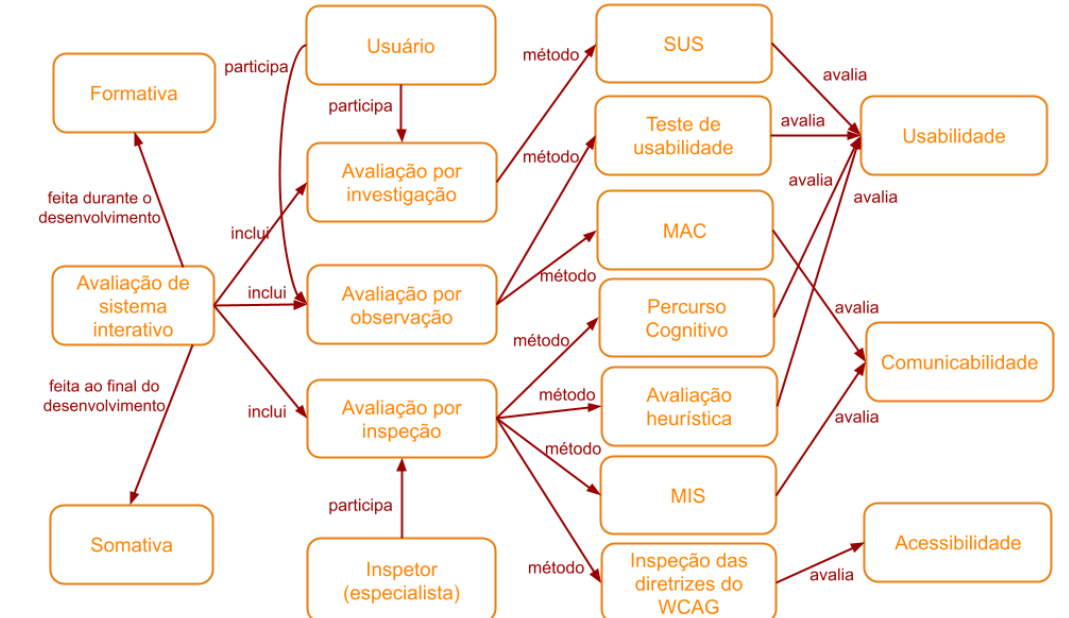
**Itens que compõe o questionário SUS**  
1.    Eu acho que gostaria de usar esse sistema com frequência.  
2.    Eu acho o sistema desnecessariamente complexo.  
3.    Eu achei o sistema fácil de usar.  
4.    Eu acho que precisaria de ajuda de uma pessoa com conhecimentos técnicos para usar o sistema.  
5.    Eu acho que as várias funções do sistema estão muito bem integradas.  
6.    Eu acho que o sistema apresenta muita inconsistência.  
7.    Eu imagino que as pessoas aprenderão como usar esse sistema rapidamente.  
8.    Eu achei o sistema atrapalhado de usar.  
9.    Eu me senti confiante ao usar o sistema  
10.    Eu precisei aprender várias coisas novas antes de conseguir usar o sistema.

**Opções de resposta a cada item**  
1.    Discordo Completamente  
2.    Discordo Parcialmente  
3.    Não Concordo e Nem Discordo  
4.    Concordo Parcialmente  
5.    Concordo Completamente

**Cálculo da Pontuação**  
Primeiro faz-se a conversão:  
- Para itens ímpares (1, 3, 5, 7, 9) subtraia 1 da pontuação que o usuário respondeu.  
- Para itens pares (2 , 4, 6, 8, 10) subtraia a resposta do usuário de 5.

Por fim, some todos os valores das 10 respostas convertidas e multiplique por 2.5. Essa é a pontuação final, que pode ir de 0 a 100.

Esse é o procedimento para as respostas de 1 usuário participante da avaliação. Idealmente, deve-se fazer isso para mais de um usuário e fazer a média do valor final. O valor de referência para análise do resultado final é 68 pontos, que é um valor obtido como típico em avaliação com mais de 400 produtos. Um resultado final abaixo de 50 é um sinal de que problemas de usabilidade podem levar ao insucesso do sistema.



- UNIDADE 1 - PROGRAMAÇÃO ORIENTADA PARA OBJETOS

Tema 1: Fatores da qualidade de software

1. [Introdução à Programação Modular](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-1-introducao-a-programacao-modular)
2. [Fatores Externos](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-1-fatores-externos)
3. [Fatores Internos](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-1-fatores-internos)

Tema 2: Classes e objetos

1. [Atributos e métodos](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-2-atributos-e-metodos)
2. [Construtores](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-2-construtores)
3. [Atributos estáticos e propriedades](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-2-atributos-estaticos-e-propriedades)
4. [Destrutores](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-2-destrutores)

Tema 3: Encapsulamento

1. [Princípio da Ocultação da Informação](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-3-principio-da-ocultacao-da-informacao)
2. [Modificadores de acesso](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-3-modificadores-de-acesso)
3. [Métodos de acesso e Propriedades](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-3-metodos-de-acesso-e-propriedades)
4. [Namespaces e partial classes](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-3-namespaces-e-partial-classes)

Tema 4: Herança

1. [Generalização e especificação](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-4-generalizacao-e-especificacao)
2. [Construtores em classes filhas](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-4-construtores-em-classes-filhas)

 - UNIDADE 2 - POLIMORFISMO E PRINCÍPIOS DO PROJETO MODULAR

Tema 1: Polimorfismo de Inclusão

1. [Sobreposição de métodos (virtual e override)](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-1-sobreposicao-de-metodos-virtual-e-override)
2. [Classes abstratas](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-1-classes-abstratas)
3. [Classes e membros selados (sealed)](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-1-classes-e-membros-selados-sealed)

Tema 2: Polimorfismo paramétrico

1. [Tipos genéricos](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-2-tipos-genericos)
2. [Coleções](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-2-colecoes)
3. [Delegates, funções lambda e eventos](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-2-delegates-funcoes-lambda-e-eventos)

Tema 3: Projeto Orientado para Objetos

1. [Princípios SOLID](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-3-principios-solid)
2. [Padrões de Projeto](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-3-padroes-de-projeto)

- UNIDADE 1 - PROGRAMAÇÃO ORIENTADA PARA OBJETOS

Tema 1: Fatores da qualidade de software

1. [Introdução à Programação Modular](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-1-introducao-a-programacao-modular)

**A PROGRAMAÇÃO ESTRUTURADA**

**Estrutura básica de um algoritmo**

Um algoritmo é definido como uma sequência de passos não ambíguos para resolver um problema em tempo finito. Em teoria, qualquer problema que possua uma solução algorítmica pode ser programado em uma linguagem de programação. Nos primórdios da computação Corrado Bőhm e Guiseppe Jacopini (1966) diziam que qualquer algoritmo poderia ser expressado com apenas 3 estruturas de controle: comandos sequenciais, comandos de seleção (ou condicionais) e comandos de repetição (ou iterativos).

Nesta época, os programas eram pequenos, rodavam em um único computador, com um único processador, que executava uma única tarefa de cada vez, para um único usuário específico. Por isto, a preocupação do programador se limitava a entregar programas corretos, ou seja, que executavam a funcionalidade solicitada conforme havia sido especificado.

**Surgimento das funções e procedimentos**

Entretanto, programar é uma tarefa repetitiva. Vários programas executam funcionalidades semelhantes, com pequenas variações. Além do mais, na medida em que a computação evolui, um programa de computador passa a executar cada vez mais funções. Surgem então, nas linguagens de programação, duas abstrações: funções e procedimentos.

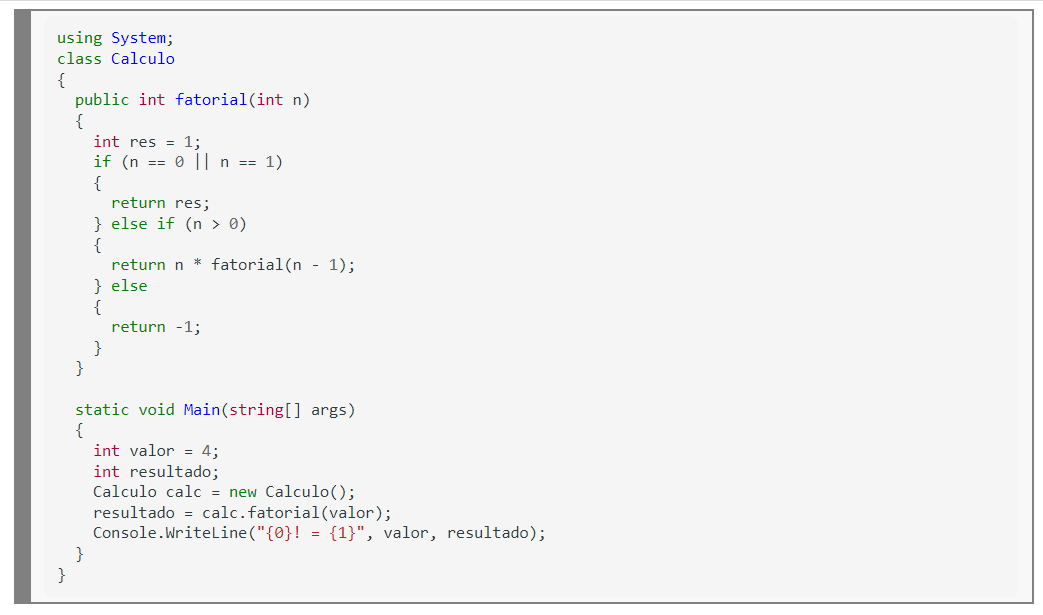
* **Funções:**são o que chamamos de***abstração de expressão***. Elas representam o mapeamento de entradas em um resultado, assim como as funções algébricas da matemática, que mapeiam dados de um domínio em uma imagem.

                Exemplo: função Fatorial (representada pela exclamação)

                Na matemática:

                       0! = 1  
                       n! = n \* (n-1)!  
                       4! = 4 \* (4-1) \*(4-2) \* (4-3) \* (4-4)  
                       4! = 4 \* 3 \* 2 \* 1 \* 1  
                       4! = 24

Em computação (usando a linguagem C#):



* **Procedimentos:** são o que chamamos de ***abstração de comandos***. Um procedimento representa um conjunto de passos que descreve como uma tarefa é feita. Ao contrário das funções, um procedimento pode ou não produzir um resultado ou afetar o estado global do sistema.

Exemplo: procedimento imprimir menu (usando a linguagem C#):



Surge então o 1º critério de decomposição de programas ou modularização: transformar cada passo principal do processamento em um módulo.

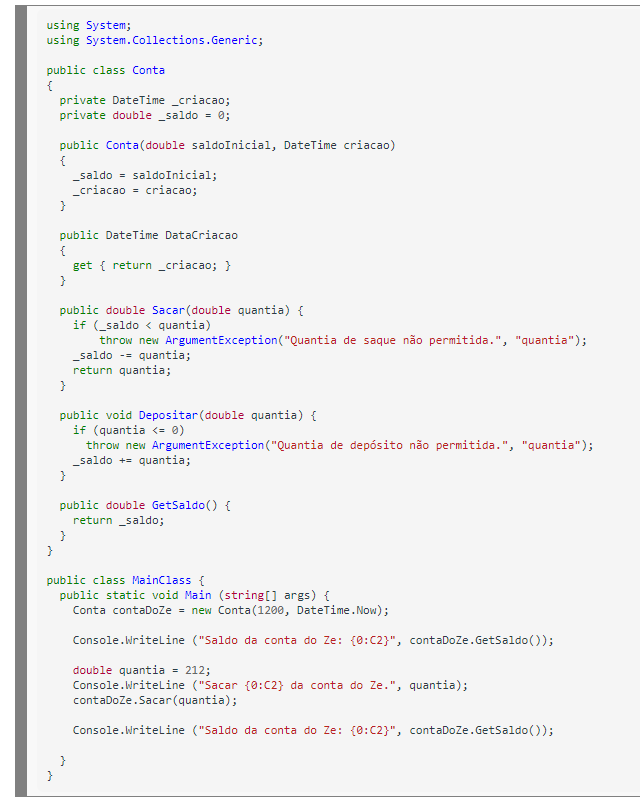
**TIPOS ABSTRATOS DE DADOS**

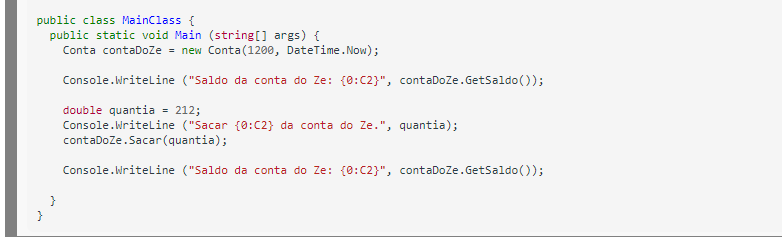
Um tipo de dado, representa um conjunto de valores e as operações que são definidas sobre esses valores. Assim como na teoria dos conjuntos da matemática, todos os valores do mesmo tipo compartilham propriedades e as formas como podem ser manipulados.

Um tipo abstrato de dados encapsula (ou esconde) a forma como seus elementos são implementados, mas disponibiliza operações para que seus elementos sejam manipulados. Nestes casos, um módulo passa a ser um tipo abstrato de dados, com atributos e operações que podem ser aplicadas sobre os dados armazenados nestes atributos.

A programação orientada para objetos é o principal paradigma de programação que implementa Tipos Abstratos de Dados. Neste paradigma, cada tipo representa uma classe de objetos, e cada objeto representa uma instância desta classe.

 Exemlo: Classe Conta bancária (usando a linguagem C#):





**PROGRAMAÇÃO ORIENTADA PARA OBJETOS**

Com os tipos abstratos de dados, um sistema pode ser decomposto em unidades independentes que provêm uma funcionalidade específica sobre uma entidade de dados. Um módulo deve obedecer ao princípio da ocultação da informação, onde os atributos de um módulo devem ser ocultados dos usuários deste módulo.

A orientação para objetos abstrai o mundo real utilizando objetos que interagem entre si. A orientação para objetos melhora a reusabilidade e extensibilidade dos softwares. Ou seja, objetos podem ser reutilizados em outros sistemas que precisam da mesma funcionalidade, e novos objetos podem ser acrescentados ao sistema sem causar impacto aos objetos já existentes.

1. [Fatores Externos](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-1-fatores-externos)

FATORES DE QUALIDADE DE PRODUTO   
   
FUNCIONALIDADE   
• Adequação   
• Completude   
• Correção   
   
DESEMPENHO   
• Tempo de resposta   
• Utilização de recursos   
• Capacidade   
   
COMPATIBILIDADE   
• Coexistência   
• Interoperabilidade   
   
USABILIDADE   
• Adequação   
• Facilidade de aprendizado   
• Facilidade de uso   
• Proteção contra erros do usuário   
• Estética   
• Acessibilidade   
   
CONFIABILIDADE   
• Maturidade   
• Disponibilidade   
• Tolerância a falhas   
• Recuperabilidade   
   
SEGURANÇA   
• Confidencialidade   
• Integridade   
• Não-repúdio   
• Autenticidade   
• Responsabilidade   
   
MANUTENIBILIDADE   
• Modularidade   
• Reusabilidade   
• Analisabilidade   
• Modificabilidade   
• Testabilidade   
   
PORTABILIDADE   
• Adaptabilidade   
• Instabilidade   
• Substutibilidade

**FATORES EXTERNOS DE QUALIDADE**

Os **fatores externos**de qualidade de software são aqueles com os quais os usuários interagem, ou seja, impactam os usuários durante o seu uso. Estes fatores são extremamente relevantes, pois eles geram a percepção de valor do software.

Entretanto, nem todos os fatores externos de qualidade influenciam no estilo de programação de um sistema. Iremos discutir alguns desses fatores que são afetados pelo projeto modular de uma aplicação.

**Correção e Robustez**

Estes dois fatores de qualidade andam sempre juntos, por serem de certa forma complementares.

**Correção:**Um sistema deve executar exatamente sua função, conforme definida pelos requisitos e especificação. Um sistema é correto se ele garante o funcionamento do que foi previsto.

**Robustez:**Um sistema é considerado robusto se ele funcionar mesmo em condições anormais. Se a correção garante as respostas corretas no caso de situações previstas, a robustez garante o funcionamento do sistema em situações imprevistas, em casos de exceção ou de erro.

Evidentemente, em caso de erro, o sistema terá sua funcionalidade limitada. Por exemplo, se um aplicativo de rede social fica sem conexão com a internet, ele não será capaz de carregar novas postagens de usuário, mas pode continuar funcionando com as postagens que já foram carregadas.

Esta habilidade do sistema continuar a operar mesmo no caso de falha de alguns dos seus componentes é chamada de graceful degradation (degradação suave). Nesta situação, algumas funcionalidades não estarão disponíveis, causando uma degradação do uso, mas o aplicativo como um todo não irá travar ou terminar de forma inesperada.

**Sistemas que se adaptam a novas situações**

O Manifesto para Desenvolvimento Ágil de Software expressa que mudanças nos requisitos de um software são bem-vindas, mesmo tardiamente no desenvolvimento. Para atender estas e outras demandas do desenvolvimento ágil de software, a programação modular busca alcançar outros fatores esternos de qualidade: extensibilidade, reusabilidade e compatibilidade.

**Extensibilidade:**se refere à medida da facilidade com que o software pode ser adaptado para atender a mudanças na sua especificação. Ela pode ser facilitada por projetos simples e módulos autônomos.

**Reusabilidade:** sugere que os módulos ou partes de um sistema devem poder ser usados em novas aplicações. Para que isso seja possível, é importante que um módulo seja coeso e com baixo acoplamento, ou seja, possua um conjunto mínimo de dependências de outros módulos.

**Compatibilidade:** se refere a medida da facilidade com que um software pode ser combinado com outros. O uso de padrões no desenvolvimento de software facilita a compatibilidade.

**Desempenho e sua relação com a correção e a robustez**

Um outro fator externo de qualidade é a Eficiência, que consiste no bom uso dos recursos de hardware. A eficiência é fundamental, principalmente em serviços que precisam ser escaláveis, ou seja, que precisam atender a muitos usuários simultaneamente.

A programação modular busca produzir módulos autocontidos que, por não compartilharem estado, já devem produzir uma aplicação relativamente eficiente do ponto de vista do uso de recursos. Mas no desenvolvimento de sistemas complexos, a correção e a robustez são recursos prioritários.

1. [Fatores Internos](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-1-fatores-internos)

Os fatores externos de qualidade são os percebidos pelos usuários e, por isso, são considerados mais importantes. Entretanto, eles só podem ser atingidos por meio dos fatores internos.

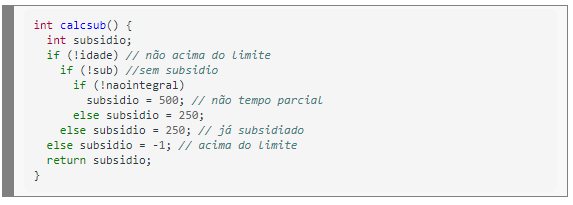
Os fatores internos são aqueles com os quais os programadores interagem. Os principais fatores internos que iremos analisar são:

* Modularidade;
* Legibilidade;
* Manutenibilidade.

**Legibilidade** consiste na facilidade de se entender o código-fonte. Ela pode ser alcançada a partir da nomeação correta de componentes e organização do código.

Um dos aspectos mais importantes da legibilidade é a inteligibilidade, que consiste na facilidade de se compreender a coerência de um software em um nível superior do que a legibilidade proporciona. Enquanto a legibilidade está preocupada com a facilidade com que um programador consegue ler e identificar os elementos de um software, a inteligibilidade está preocupada com a capacidade deste desenvolvedor identificar as relações entre os componentes deste software. Por isto, as linguagens de programação possuem padrões de codificação (Coding Standards) que devem ser seguidos, para garantir uma compreensão universal do código. As convenções de codificação em C# são uma leitura obrigatória para o desenvolvedor que deseja desenvolver código de inteligível em C#. Elas criam uma aparência consistente, para que os desenvolvedores possam se concentrar no conteúdo, e permitem um entendimento mais rápido dos elementos do código.

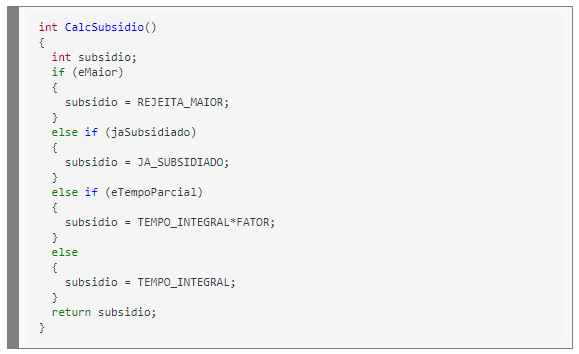
Exemplo: Função de cálculo de subsídio com baixa legibilidade



Problemas de legibilidade encontrados no exemplo acima:

1. Nomes não seguem o padrão de maiúsculas e minúsculas nem de PascalCase ou CammelCase, que facilitam a leitura.
2. Excesso de comentários já sugere que o código não está legível.
3. Valores das variáveis não deixam claro o que eles significam. Este problema ainda dificulta a manutenção no caso de reajuste desses valores.
4. Os nomes das variáveis não estão apropriados e dificultam o entendimento das expressões lógicas.
5. A falta das chaves dificulta a leitura dos caminhos de execução das estruturas condicionais.
6. Expressões lógicas escritas na forma negativa dificultam o seu entendimento.

Exemplo: Função de cálculo de subsídio com alta legibilidade.



1. Uso de PascalCase e CamelCase nos nomes da função e das variáveis facilitam a leitura.
2. Variáveis lógicas expressam exatamente o que está sendo testado, com nomes que representam perguntas cuja resposta é sim ou não.
3. Os valores foram substituídos por parâmetros constantes, que indicam seu significado e podem ser facilmente modificados no caso de necessidade.
4. Organização lógica das estruturas condicionais permitem um único caminho sequencial.

**Manutenibilidade** consiste na facilidade para se modificar um software ou corrigir problemas. Se um projeto modular é extensível, espera-se que a manutenção seja necessária apenas em situações de correção de erros, mudanças na especificação ou adaptação a novos ambientes.

**Modularidade** é o fator de qualidade relacionado com a divisão do sistema. Um módulo é um grupo de elementos de código ou comandos com uma funcionalidade bem definida e o mais independente possível em relação ao resto do sistema. Por isso, a modularidade pressupõe duas características de projeto fundamentais: princípio da unidade funcional e a coesão.

O princípio da unidade funcional expressa que um módulo deve exercer apenas uma função. Se o módulo exerce mais de uma função, ele deveria ser decomposto em vários submódulos.

Por outro lado, um módulo é dito coeso se todo o código necessário para implementar a sua função está contida no módulo.

Um sistema então será composto de um módulo principal de controle e módulos específicos para as funções do sistema.

Tema 2: Classes e objetos

1. [Atributos e métodos](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-2-atributos-e-metodos)

**CLASSES E OBJETOS**

A Programação Orientada para Objetos abstrai o mundo real, utilizando objetos que interagem entre si.

Um objeto possui:  
                    Estado, comportamento, identidade.

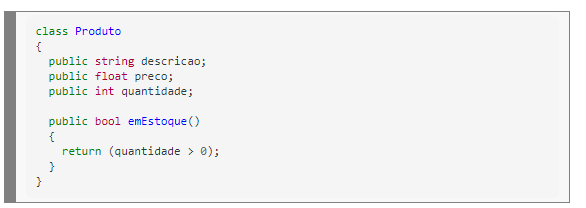
Classes representam um grupo de objetos que compartilham as mesmas propriedades e comportamentos. O vídeo a seguir mostra como podemos declarar uma classe com seus atributos e métodos.

Considere que desejamos implementar uma classe “Produto” que representa uma entidade Produto em um Sistema de Gerenciamento de Estoque. A classe Produto poderia ter os seguintes atributos:

               Descrição: do tipo texto  
               Preço: do tipo real  
               Quantidade: do tipo inteiro

A classe produto poderia ainda ter o seguinte método:  
               emEstoque( ): do tipo lógico (retorna verdadeiro se a quantidade em estoque é maior que 0).

A seguir temos o código da classe Produto em C#:

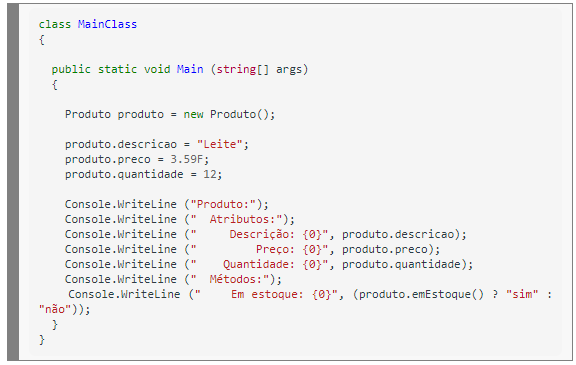


**SEMÂNTICA DE REFERÊNCIA**

Um objeto é uma instância de uma Classe, que possui um nome e uma posição na memória. Nesta posição de memória são armazenados os valores de seus atributos. Classes podem ter múltiplas instâncias de Objetos.

Em C# a declaração de um Objeto, na verdade, cria uma referência para o objeto, mas não o objeto propriamente dito. Uma referência armazena o endereço onde o objeto se encontra. Após declaramos uma referência, precisamos criar efetivamente o objeto.

A classe a seguir instancia um objeto do tipo Produto e inicializa os atributos do produto criado.



**Em resumo:**

       Produto produto  - cria uma referência para Produto, mas não aloca a memória para armazenar o objeto. O valor inicial da variável produto é null.

       produto = **new** Produto()  - cria efetivamente o objeto do tipo Produto e retorna o endereço onde o objeto foi alocado. O endereço é armazenado na referência produto.

       produto.descricao  - acessa diretamente o atributo de produto.

       produto.emEstoque()  - executa o método de produto.

1. [Construtores](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-2-construtores)

**CONSTRUTORES**

As linguagens que possuem coletores de lixo, como o C#, quando inicializam um objeto, os atributos são inicializados com 0, no caso de atributos numéricos, ou **null**, no caso de atributos que são referências para objetos.

Por exemplo, o comando:

Produto produto = new Produto();

 Irá produzir um produto com os seguintes valores:  
           Descrição: null  
           Preço: 0.0  
           Quantidade: 0

Por mais que a quantidade de um produto em estoque possa ser zero, não é correto um sistema permitir a criação de um produto sem nome e sem preço.

Construtores devem ser utilizados para inicializar objetos com valores diferentes do padrão, e garantir que o sistema esteja sempre em um estado válido. Vamos entender melhor como os construtores devem ser utilizados.

Em C#, os construtores devem possuir o mesmo nome da classe, e não possuem valores de retorno.

Uma classe pode ter de 0 a muitos construtores, desde que a assinatura dos construtores seja diferente. A assinatura de um método ou construtor é definida pela quantidade e tipo de seus argumentos, na ordem em que são declarados.

Se a classe não possui nenhum construtor, o C# fornece um construtor *default* (construtor padrão sem parâmetros), que inicializa os atributos do objeto com 0 ou **null**. A partir do momento que um desenvolvedor adiciona um construtor à classe, o C# não mais fornece o construtor vazio, e passa a respeitar a forma de inicialização implementada pelo desenvolvedor.

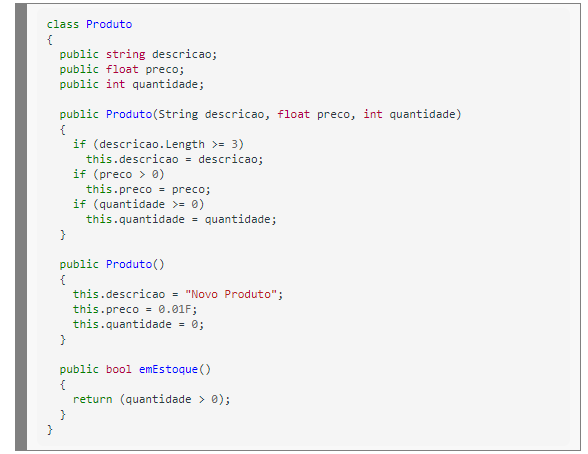
**RAZÕES PARA SE USAR CONSTRUTORES ESPECIALIZADOS**

* Algumas classes não possuem estado inicial aceitável sem parâmetros e, portanto, os construtores são necessários para inicializar o objeto com um estado válido.
* Em alguns casos, fornecer um estado inicial é conveniente e aceitável quando da construção de alguns tipos de objetos. Por exemplo, normalmente, quando um sistema emite um boleto, a data de emissão do boleto corresponde à data do sistema, e este valor pode ser convenientemente atribuído no momento da construção.
* Quando um objeto possui muitos atributos, ou os valores dos atributos são complexos, por exemplo, construir um objeto aos poucos pode ser desgastante de forma que pode ser conveniente que se tenha um estado inicial correto quando forem criados.

**CONSTRUTORES NA CLASSE PRODUTO**

Vamos adicionar robustez à classe Produto, criando construtores diferentes do padrão.

O código a seguir exibe a classe Produto com 2 construtores. Um construtor default, que não recebe parâmetros e inicializa com valores válidos para nome e preço. E um construtor com parâmetros, que recebe valores diferentes do padrão e inicializa os atributos, se estes valores atenderem às regras de negócio estabelecidas para os atributos.



A aplicação agora pode construir um objeto da classe produto de duas maneiras diferentes, conforme os códigos a seguir.

Produto produto1 = new Produto();

Produto produto2 = new Produto("Leite", 3.59F, 12);

1. [Atributos estáticos e propriedades](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-2-atributos-estaticos-e-propriedades)

**MEMBROS ESTÁTICOS**

O **escopo** de uma variável descreve a região do código em que aquela variável pode ser acessada. O termo **escopo** é também comumente chamado como visibilidade da variável, pois compreende a parte do código em que uma variável é “reconhecida” pelo programa.

Um membro estático é definido como um membro com tempo de vida global, e escopo local. São atributos ou métodos que são comuns a todos os objetos de uma classe. Quando declaramos um atributo ou método estático, ele passa a ser um membro de classe, e é compartilhado por todos os objetos daquela classe.

Membros estáticos são úteis para implementar contadores ou identificadores de autoincremento.

Em C#, para implementarmos um membro estático utilizamos o modificador **static** na declaração.

Membros estáticos têm este nome porque são alocados no instante em que o código da classe é carregado na memória, e seu endereço permanece o mesmo ao longo de todo o programa, com todos os objetos acessando o mesmo endereço.

Por este motivo, membros estáticos existem antes mesmo de se existir um objeto daquela classe.

**EXEMPLO DE CONTADOR AUTOINCREMENTO**

Suponha que desejamos adicionar um novo atributo à classe Produto, que consiste em um código que é incrementado a cada vez que uma nova instância de Produto é criada. Iremos utilizar atributos estáticos para resolver este problema.

O código em C# para a implementação do contador autoincremento se encontra a seguir:

class Produto

{

public int id;

public string descricao;

public float preco;

public int quantidade;

public static int contador;

public Produto(String descricao, float preco, int quantidade)

{

this.id = ++Produto.contador;

if (descricao.Length >= 3)

this.descricao = descricao;

if (preco > 0)

this.preco = preco;

if (quantidade >= 0)

this.quantidade = quantidade;

}

public Produto()

{

this.id = ++Produto.contador;

this.descricao = "Novo Produto";

this.preco = 0.01F;

this.quantidade = 0;

}

public bool emEstoque()

{

return (quantidade > 0);

}

}

Como um atributo estático pertence ao escopo da classe, sempre que quisermos acessá-lo devemos referenciá-lo a partir da classe:

Console.WriteLine("Contador de produtos: {0}", Produto.contador);

1. [Destrutores](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-2-destrutores)

**DESTRUTORES**

Um destrutor é um método que é chamado toda vez que um objeto é destruído e a memória ocupada por este objeto é desalocada. O destrutor já existe por padrão, e apenas limpa a memória que o objeto estava ocupando. Entretanto, é possível se incluir um destrutor em uma classe, se for necessário executar algum procedimento sempre que instâncias da sua classe são destruídas.

Construtores em C# possuem o mesmo nome da classe, precedido por um til (~).

Não é possível se chamar um destrutor diretamente. Quando um objeto não é mais acessível a partir de nenhuma referência disponível, ele permanece na memória até que o coletor de lixo seja executado. O coletor de lixo executa esporadicamente em paralelo com a aplicação, em uma Thread separada, liberando os recursos de memória que não podem mais ser utilizados.

Apesar de desaconselhável, é possível solicitar ao .NET Framework, que executa as aplicações C#, para dedicar um esforço adicional para executar o coletor de lixo usando o comando ***GC.Collect( )***.

Veja a seguir a classe Produto com um destrutor.

class Produto

{

*// ...*

public static int contador = 0;

public static int instancias = 0;

public Produto(String descricao, float preco, int quantidade)

{

Produto.instancias++;

*// ...*

}

public Produto()

{

Produto.instancias++;

*// ...*

}

~Produto()

{

Console.WriteLine("Executando o destrutor...");

Produto.instancias--;

}

*// ...*

}

Tema 3: Encapsulamento

1. [Princípio da Ocultação da Informação](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-3-principio-da-ocultacao-da-informacao)

**PRINCÍPIO DA OCULTAÇÃO DA INFORMAÇÃO**

**Ocultação de informação**  
Toda informação a respeito de um módulo deve ser privativa do módulo, exceto se for explicitamente declarada como pública.

Parnas introduziu o conceito por volta de 1972, argumentando que a modularização de um sistema deveria ocultar decisões críticas de projeto, minimizando o impacto de mudanças nos demais módulos do sistema.

Um dos principais custos de manutenção de software é incorporar mudanças na especificação. Uma mudança que requer a alteração de múltiplos módulos é mais cara e mais suscetível a erros.

**Interface**  
Corresponde a tudo aquilo que um usuário do objeto vê e acessa.

No princípio da ocultação da informação, o objetivo é projetar interfaces que capturam os aspectos estáveis e deixar os aspectos dinâmicos para a implementação.



Um modulo pode ocultar:

* Algoritmos
* Representação de dados

Interfaces possuem duas partes:

* Assinatura: os nomes e tipos das informações sobre as funções do módulo.
* Especificação: uma descrição precisa da semântica dos elementos de um módulo (significado e comportamento).

Exemplo simples de ocultação da informação: função raiz quadrada  
Interface:

* Assinatura: public double sqrt(double n)
* Semântica: calcula a raiz quadrada do número **n**

Implementação:

* Implementação pelo método da bisessão
* Implementação pelo método de Newton
* Implementação pelo método da fatoração

Outro exemplo é a classe DateTime em C#, cuja interface expõe a representação em dia, mês e ano, mas a implementação utiliza um inteiro longo sem sinal.

**ENCAPSULAMENTO**

Todo módulo produz encapsulamento. Encapsulamento permite alterar a implementação de um objeto sem impactos em outros módulos do sistema. Permite ainda que seus dados sejam protegidos de acesso ilegal e que valores sejam validados antes de serem atribuídos aos atributos.

Por exemplo, acessar diretamente o campo quantidade da classe Produto e atribuir um valor negativo pode invalidar todo o funcionamento da classe Produto.

**Encapsulamento e o princípio da caixa preta:**Um módulo deve consistir de um conjunto de comandos com uma função bem definida, e o mais independente possível em relação ao resto do sistema.

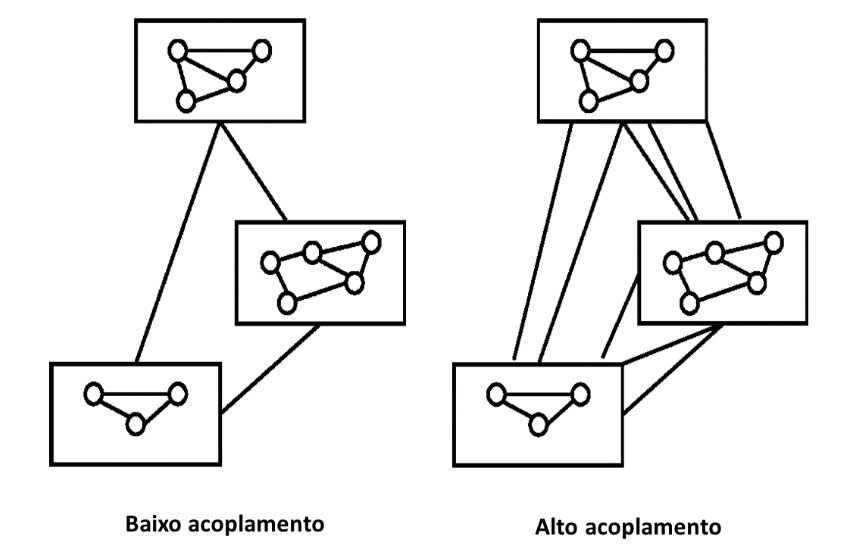
**Independência funcional:**Cada módulo deve cuidar de uma função específica, servindo a um propósito específico.

**Coesão:**Atributo de um módulo em que todas as suas partes estão ligadas umas às outras. Em software, todas as partes estão coerentemente relacionadas. O objetivo de um módulo em programação modular é obter alta coesão interna.

**Acoplamento:**Medida da interconexão entre os elementos de software. A dependência pode ser medida pela quantidade de conexões entre os módulos de um sistema. A situação ideal é que um sistema possua baixo acoplamento.

Indicadores de baixo acoplamento

* Tamanho: quantidade de parâmetros e métodos públicos.
* Visibilidade: uso de parâmetros x uso de variáveis globais.
* Flexibilidade: facilidade na chamada



1. [Modificadores de acesso](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-3-modificadores-de-acesso)

**MODIFICADORES DE ACESSO**

O encapsulamento é o recurso da programação modular que permite implementar o princípio da ocultação da informação. Com o encapsulamento, é possível se alterar a implementação de um objeto sem impactar outros módulos do sistema. Outra vantagem do encapsulamento é que ele permite que seus dados sejam protegidos de acesso ilegal.

As linguagens de programação modernas, como o C#, permitem controlarmos a visibilidade de membros de uma classe, de forma a ocultarmos determinados dados e ou  métodos do usuário da abstração.

Vários níveis de acesso podem ser atribuídos aos atributos e métodos de uma classe. Em C#, os três níveis mais comuns de acesso são: privado, protegido e público. Esses níveis são definidos a partir de modificadores de acesso, que são palavras-chave da linguagem que são adicionadas na declaração destes membros de classe.

**Privado**(modificador de acesso private)

               Membros declarados com acesso privado são acessíveis apenas na própria classe.

**Protegido**(modificador de acesso protected)

           Membros declarados com acesso protegido são acessíveis na própria classe e adicionalmente por suas subclasses (ou classes filha).

**Público**(modificador de acesso public)

               Membros declarados com acesso público são acessíveis de qualquer lugar do programa.

**Nota:** O nível de acesso padrão para membros da classe é privado quando nenhum modificador de acesso é especificado.

No caso da classe Produto, por exemplo, o natural seria que todos os seus atributos fossem privados, e apenas os construtores e o método produto.emEstoque() pudessem ser acessados externamente pelos usuários da classe Produto.

class Produto

{

private int id;

private string descricao;

private float preco;

private int quantidade;

private static int contador = 0;

private static int instancias = 0;

public Produto(String descricao, float preco, int quantidade)

{…

}

public Produto()

{…

}

~Produto()

{…

}

public bool emEstoque()

{…

}

}

**REGRAS DE ENCAPSULAMENTO**

Para a correta aplicação do princípio da ocultação da informação, algumas regras básicas (*rules of thumb*) devem ser seguidas, como as apresentadas a seguir.

* Use o nível de acesso mais restrito e que faça sentido para um membro particular.
* Use private a menos que haja uma boa razão para não o fazer.
* Evite campos public exceto para constantes. Campos públicos aumentam o acoplamento em relação a uma implementação específica e reduz a flexibilidade do sistema a mudanças.

Vários níveis de acesso podem ser atribuídos aos atributos e métodos de uma classe. Em C#, os três níveis mais comuns de acesso são: privado, protegido e público. Esses níveis são definidos a partir de modificadores de acesso, que são palavras-chave da linguagem que são adicionadas na declaração destes membros de classe.

**Privado**(modificador de acesso private)

               Membros declarados com acesso privado são acessíveis apenas na própria classe.

**Protegido**(modificador de acesso protected)

           Membros declarados com acesso protegido são acessíveis na própria classe e adicionalmente por suas subclasses (ou classes filha).

**Público**(modificador de acesso public)

               Membros declarados com acesso público são acessíveis de qualquer lugar do programa.

**Nota:** O nível de acesso padrão para membros da classe é privado quando nenhum modificador de acesso é especificado.

No caso da classe Produto, por exemplo, o natural seria que todos os seus atributos fossem privados, e apenas os construtores e o método produto.emEstoque() pudessem ser acessados externamente pelos usuários da classe Produto.

class Produto

{

private int id;

private string descricao;

private float preco;

private int quantidade;

private static int contador = 0;

private static int instancias = 0;

public Produto(String descricao, float preco, int quantidade)

{…

}

public Produto()

{…

}

~Produto()

{…

}

public bool emEstoque()

{…

}

}

**REGRAS DE ENCAPSULAMENTO**

Para a correta aplicação do princípio da ocultação da informação, algumas regras básicas (*rules of thumb*) devem ser seguidas, como as apresentadas a seguir.

* Use o nível de acesso mais restrito e que faça sentido para um membro particular.
* Use private a menos que haja uma boa razão para não o fazer.
* Evite campos public exceto para constantes. Campos públicos aumentam o acoplamento em relação a uma implementação específica e reduz a flexibilidade do sistema a mudanças.

1. [Métodos de acesso e Propriedades](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-3-metodos-de-acesso-e-propriedades)

**MÉTODOS DE ACESSO**

O encapsulamento é o princípio da programação orientada para objetos que expressa que devemos ocultar os atributos e publicarmos os métodos. Realmente, não faz sentido ocultarmos os atributos se não pudermos acessá-los.

Métodos de acesso são criados para permitir que os usuários de uma classe possam acessar os valores dos atributos, sem, no entanto, expor esses atributos à possíveis atribuições de valores inválidos.

Os métodos de acesso são chamados de getters e setters.

**Métodos get:**acessam o valor de um atributo privado. Eventualmente, os valores podem ser tratados antes de serem exibidos.

Por exemplo: um atributo booleano pode ser exibido como V ou F, e um atributo numérico pode ser retornado no formato string correspondente.

**Métodos set:** atribuem um valor a um atributo privado. Os valores passados como argumento devem ser validados/tratados antes de serem atribuídos.

Por exemplo: o número do dia numa classe Data depende do atributo mês.

Por questão de legibilidade, getters e setters possuem uma nomenclatura bem definida. Eles possuem a palavra Get ou Set seguida do nome do atributo. Por exemplo:

O atributo

private float preco;

Irá produzir os métodos de acesso:

public float GetPreco()

{

return preco;

}

public void SetPreco(float preco)

{

if (preco > 0)

{

this.preco = preco;

}

}

**PROPRIEDADES**

As propriedades permitem expor atributos de forma pública, ao passo que permite implementar o código dos métodos de acesso de forma flexível e com menos linhas de código.

As propriedades podem ter os seguintes métodos:

**init:** atribui um valor inicial ao atributo no momento da criação do objeto.

**set:** atribui um novo valor ao atributo enquanto verifica a validade do valor a ser atribuído.

**get:**retorna o valor do atributo, diretamente, ou após processamento ou formatação.

Se transformarmos o atributo **preco**em uma propriedade, a declaração seria a seguinte:

public float Preco {

get { return preco; }

set { if (value > 0) preco = value; }

}

**value** representa o valor passado ao método de acesso**set.**

As propriedades possuem métodos assessores padrão que não fazem nenhuma validação ou transformação de dados. Por exemplo: a propriedade Preco a seguir

public float Preco { get; set; }

retorna o valor de preco e inicializa o preco com o valor do argumento value, sem verificar se este valor é válido.

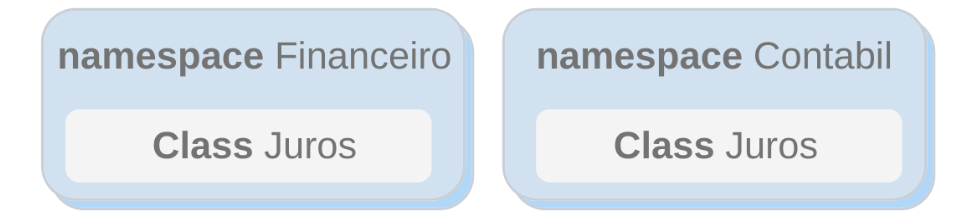
1. [Namespaces e partial classes](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-3-namespaces-e-partial-classes)

**NAMESPACES**

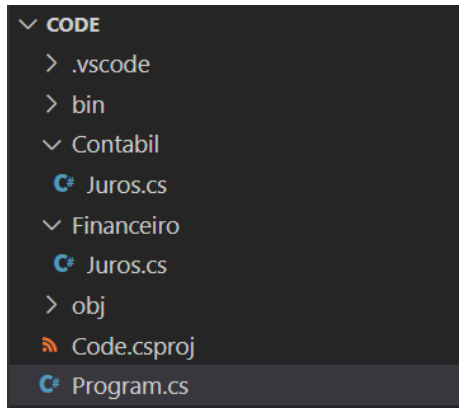
**Namespaces**, ou espaço de nomes, é um recurso que permite agrupar classes relacionadas em uma unidade lógica maior, criando módulos mais complexos. Muitas vezes, os **namespaces** são utilizados para a criação de bibliotecas de código (code libraries).

Por convenção, quando se deseja criar seu próprio **namespace**, todas as classes de um mesmo **namespace**devem ser armazenadas na mesma pasta, em arquivos separados onde cada arquivo possui o mesmo nome da classe nele contido. O nome da pasta deve ser o mesmo nome do **namespace**.

A figura a seguir busca ilustrar a organização lógica de um projeto com **namespaces**.



A estrutura de pastas e arquivos deve ficar assim:



Cada classe deve ser declarada em seu **namespace**:

namespace Contabil

{

public class Juros

{

public float Valor { get; set; }

}

}

E os outros módulos poderão utilizar outros **namespaces** com a cláusula using

using Financeiro;

using Contabil;

class MainClass

{

public static void Main(string[] args)

{

Financeiro.Juros jurosDeMora = new Financeiro.Juros();

jurosDeMora.Valor = 120.0F;

Contabil.Juros jurosAReceber = new Contabil.Juros();

jurosAReceber.Valor = 231987.23F;

Console.WriteLine($"Juros de mora: {jurosDeMora.Valor:C2}");

Console.WriteLine($"Juros a receber: {jurosAReceber.Valor:C2}");

}

}

Além de organizar o projeto em módulos maiores, **namespaces** evitam conflitos de nomes típicos de desenvolvimento em larga escala com múltiplos desenvolvedores. No exemplo apresentado, o desenvolvedor responsável pelo módulo Contábil utilizou a classe Juros com o significado de Juros a Receber, que é um ativo. O desenvolvedor do módulo Financeiro, utilizou a classe Juros para calcular Juros de Mora, que ocorrem quando há atraso no pagamento de títulos.

**PARTIAL CLASSES**

Por padrão, o uma classe deve ser implementada em um único arquivo em C#, e em praticamente todas as linguagens de programação modernas. Existe inclusive um princípio de qualidade de código que sugere que uma classe não deve ter mais de aproximadamente 100 linhas, ou deveria ser decomposta em várias classes menores.

Entretanto, uma classe parcial, ou partial class, pode ser definida em mais de um arquivo. Isto permite a organização do conteúdo de classes muito grandes. Elas são especialmente úteis em desenvolvimento de interface gráfica, onde há muito código gerado automaticamente pelo compilador. Em projetos do Windows Forms, por exemplo, classes parciais são utilizadas para expor as estruturas que devem ser manipuladas pelos desenvolvedores, enquanto códigos complexos gerados automaticamente são “escondidos” em outros arquivos.

Classes parciais são declaradas com a palavra-chave **partial**. Por exemplo:

public partial class ClasseParcial {

}

Pode ser desenvolvido em vários arquivos, como por exemplo:  
                ClasseParcial1.cs  
                ClasseParcial2.cs

e assim por diante.

Veja um exemplo de implementação de classe parcial.

Arquivo ClasseParcial1.cs

public partial class ClasseParcial {

public string tarefa { get; set; }

}

Arquivo ClasseParcial2.cs

using System;

public partial class ClasseParcial

{

public void ExibeTarefa()

{

Console.WriteLine("Eu estou " + this.tarefa);

}

}

Arquivo Program.cs

using System;

class MainClass

{

public static void Main(string[] args)

{

ClasseParcial classeParcial = new ClasseParcial();

classeParcial.tarefa = "comendo";

classeParcial.ExibeTarefa();

}

}

Tema 4: Herança

1. [Generalização e especificação](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-4-generalizacao-e-especificacao)
2. [Construtores em classes filhas](https://pucminas.instructure.com/courses/68911/pages/unidade-1-tema-4-construtores-em-classes-filhas)

 - UNIDADE 2 - POLIMORFISMO E PRINCÍPIOS DO PROJETO MODULAR

Tema 1: Polimorfismo de Inclusão

1. [Sobreposição de métodos (virtual e override)](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-1-sobreposicao-de-metodos-virtual-e-override)
2. [Classes abstratas](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-1-classes-abstratas)
3. [Classes e membros selados (sealed)](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-1-classes-e-membros-selados-sealed)

Tema 2: Polimorfismo paramétrico

1. [Tipos genéricos](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-2-tipos-genericos)
2. [Coleções](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-2-colecoes)
3. [Delegates, funções lambda e eventos](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-2-delegates-funcoes-lambda-e-eventos)

Tema 3: Projeto Orientado para Objetos

1. [Princípios SOLID](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-3-principios-solid)
2. [Padrões de Projeto](https://pucminas.instructure.com/courses/68911/pages/unidade-2-tema-3-padroes-de-projeto)