# UNIFIED MODELING LANGUAGE

Depois de entender quais são os requisitos do sistema (que até então estão escritos), é feito uma modelagem delas, ou seja, é desenhar o software antes de começar a programar, usando formas abstratas. E para fazer esse desenho de software é usado um tipo de linguagem

A UML é uma LINGUAGEM DE MODELAGEM, voltado para a elaboração da ESTRUTURA DE PROJETOS DE SOFTWARES. Ele auxilia entender o software pelo diagrama, por ser mais rápido e mais fácil de ser entendido, economizando tempo, principalmente se o sistema for um sistema ORIENTADO A OBJETOS. E se destina principalmente para **SISTEMAS COMPLEXOS DE SOFTWARES.**

É importante seu uso, pois a comunicação esta sujeira a falhas, então o diagrama ajuda a ter o documento formalizado. Sendo que assim, o engenheiro deve fazer todo o planejamento para que o programador transformar em código (ele recebe a solução pronta). Essa modelagem segue um modelo ***PADRÃO***, sendo que cada símbolo gráfico possui um significado e função bem definida, assim um desenvolvedor pode usar a UML para escrever um modelo para outro desenvolvedor.

- ➔ Por ser ***PADRÃO, ela*** pode ser conectada com várias linguagens de programação;
- ➔ Ela abrange toda arquitetura do sistema e seus detalhes
- ➔ Expressa os requisitos
- ➔ Se destina a SISTEMAS COMPLEXOS, como SERVIÇOS BANCÁRIOS, TRANSPORTES, ELETRÔNICA MÉDICA, COMÉRCIO, SERVIÇOS WEB

Depois de entender quais são os requisitos do sistema (que até então estão escritos), é feito uma modelagem delas, ou seja, é desenhar o software antes de começar a programar, usando formas abstratas. **A UML PODE EXPRESSAR TAMBÉM FLUXO DE TRABALHO, ESTRUTURA, COMPORTAMENTO E PROJETO DE HARDWARE, NÃO PRECISA SER NECESSARIAMENTE SOFTWARE.**

Sendo assim, a UML tem como função:

***PROJETAR O SISTEMA → ENTENDO E COMPREENDENDO SEU COMPORTAMENTO → VERIFICANDO SE TEM ALGUM ERRO DESDE O INÍCIO DO PROJETO → para APRESENTAR AOS STAKEHOLDERS → para que possa ORIENTAR A IMPLEMENTAÇÃO!***

# DIAGRAMA DE CLASSE

**OS DIAGRAMAS DE CLASSE**, é um tipo de diagrama **que mostra um conjunto de classes, interfaces, colaboração e seus relacionamentos.** E classes **são blocos de construção que contém uma descrição de um objeto,** sendo representado por um RETÂNGULO divido em 3 sessões com 2 linhas.

1°SESSÃO – CLASSE

2°SESSÃO – ATRIBUTOS

3°SESSÃO: QUAIS AS FUNÇÕES

ATRIBUTO PRIVADO- so quem está dentro da classe consegue ver //

ATRIBUTO PÚBLICO- acessível para qualquer ponto do software

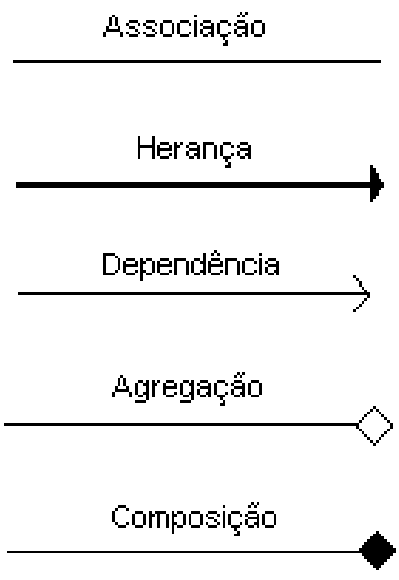
CLASSE → é o nome dado aquele "pacote" de informações, É o conceito, a Ideia

ATRIBUTOS → são as características daquela classe, seguem regras e são equivalentes as variáveis

OBJETOS → é o objeto em si feito baseado do Conceito

METÓDOS → são as ações que eles se realizam

Como cada **CLASSE** deve ser **REALIZAR APENAS 1 FUNÇÃO**, apenas **1 RESPONSABILIDADE**, representam uma **ABSTRAÇÃO DE ITENS** que podem se relacionar, tendo vários tipos de relacionamento:




ASSOCIAÇÃO: relação forte de dependência;

HERANÇA: generalização, uma funcionalidade geral vai ter várias funcionalidades especificas;

DEPENDENCIA: relação fraca, se uma não existe a outro pode existir;

AGREGAÇÃO: a classe tem um container com várias classes conectados a ela, te um relacionamento fraco;

COMPOSIÇÃO: relacionamento forte, se um container deixa de existir, os outros também desaparecem.

# DIAGRAMA DE CASO DE USO

**Os DIAGRAMAS DE CASO DE USO**, é um tipo de diagrama, ele é **COMPORTAMENTAL que mostra um conjunto de casos de uso, atores e relacionamentos. Determinando o comportamento do sistema, como esse comportamento vai ocorrer – gerando a possibilidade de comunicação dos clientes com os desenvolvedores.**

Elas são responsáveis em: **MODELAR O COMPORTAMENTO DO SISTEMA (ou SUBSISTEMA) OU CLASSE → REVELANDO FUNÇÕES ESPECÍFICAS QUE AS VEZES NÃO É VISTO → DEFININDO O ESCOPO → E IDENTIFICANDO QUEM INTERAGEM COM O SISTEMA**. Porém não detalha funcionalidades!

O DIAGRAMA DE CASO DE USO são compostos de:

- CASOS DE USOS representados por ELIPSES, representa uma sequência de ações que resulta em um resultado;
- OS ATORES, sendo representados por BONECO DE PALITO;
- UM RETÂNGULO GRANDE, representando o interior do Sistema;
- LINHAS DE RELACIONAMENTO:
  - **INCLUDE**
  - **EXTEND**
  - **GENERALIZAÇÃO ou HERANÇA**
  - **ASSOCIAÇÃO**

Todo **CASO DE USO** vai possuir um **NOM**E para diferenciar dos demais Casos De Uso, quando esse nome fica sozinho ele vai ser chamado de **NOME SIMPLES**.

Os **ATORES** vão ter um papel, eles podem ser: um ser humano, hardware ou até outro sistema. Eles **NÃO SÃO PARTE DO SISTEMA,** POIS **RESIDEM FORA DELE**. Os **ATORES** não precisam atuar como entidades separadas e podem se representados de forma esquematizadas pela seta de **GENERALIZAÇÃO,** definindo **GRUPOS GERAIS DE ATORES.**
Eles vão estar conectados com um Caso de Uso por meio da **SETA DE ASSOCIAÇÃO**, indicando que tem comunicação entre os dois.

O relacionamento de **INCLUSÃO – INCLUDE**, descreve o comportamento de um Caso de Uso, pegando as responsabilidades e o associando a um Caso de Uso (- - - - - - - - - >);

O relacionamento de **EXTENSÃO – EXTEND,** descreve quais as possíveis ações de outro caso de uso, descreve o comportamento ou os comportamentos que aquele caso de uso vai ter (--------- >);

O relacionamento de **GENERALIZAÇÃO,** descreve grupos, como por exemplo, grupo de atores. Quando tem vários processos **ESPECIFÍCOS** e podem ser representados por um processo **GERAL.**

# DIAGRAMA DE SEQUÊNCIA

Para resolver os problemas, os objetos devem trocar mensagens entre si, e para representar essa troca de mensagens, é usado o **DIAGRAMA DE SEQUÊNCIA,** esse tendo um altíssimo nível de abstração enquanto o **DIAGRAMA DE CLASSE TEM MÉDIA ABSTRAÇÃO. O DIAGRAMA DE SEQUÊNCIA pode ser considerado como um passo da programação, sendo que:**

**para cada CLASSE DE USO (tem um) → DIAGRAMA DE SEQUÊNCIA.**

Dessa forma acaba que por vezes que o diagrama de sequência ficando com muita informação, deixando o diagrama de sequência muito poluído, então se usa mais de um diagrama de sequência, mostrando como as interações ocorrem e em qual ordem ocorrem.

AJUDA EM IDENTIFICAR COMO OS OBJETOS SE COMUNICAM → IDENTIFICANDO QUAL VAI CHAMAR O OUTRO → VENDO OS REQUIITOS QUE ESSE OBJETO VAI TER → ENTENDENDO COMO O DIAGRAMA DE CASOS DE USO VAI SER RESOLVIDO (DEVE TER A DOCUEMTAÇÃO PARA NORMALIZAR – UML)

Ele é composto por:

- ATORES: são usados os mesmos atores usados do DIAGRAMA DE CLASSE DE USO, sendo tudo que fica para fora do sistema, NUNCA INTERAGEM COM O SISTEMA e eles precisam de uma INTERFACE GRÁFICA ----- CLASSE DE FRONTEIRA;
- OBJETOS: são a classes/elementos que se comunicam e vão se comunicar apenas com mensagens, os objetos estão dispostos no topo do diagrama, ESTÃO DISPOSTOS DA ESQUERDA PARA A DIREITA (DIREITA É SUBORDINADA DA ESQUERDA), dispostos no eixo x, o objeto vai ficar na esquerda e o objeto que vai ser chamado na direita;
- LINHA DE VIDA: é a linha que indica o tempo de vida daquele objeto;
- FOCO DE CONTROLE: é o retângulo que tem em cima da linha de vida e indica que está realizando uma tarefa;
- MENSAGENS: demonstra a comunicação entre objetos, mostra qual objeto está sendo requisitado, dividido em :
    - MENSAGEM ASSÍNCRONA: você manda uma mensagem que não precisa de resposta para continuar
    - MENSAGEM SÍNCRONA: você manda uma mensagem e precisa de resposta para continuar
    - MENSAGEM DE CRIAÇÃO DE PARTICIPANTE: é o pedido de criação de um novo objeto <<create>>
    - MENSAGEM DE EXCLUSÃO DE PARTICIPANTE: serve para eliminar um objeto
    - AUTO MENSAGEM: é a chamada para requisitar uma função do próprio aparelho

- MENSAGEM DE RESPOSTA/RETORNO: traz o retorno do pedido, a informação, é a reposta do “return”
- MENSAGEM DE GUARDA: usadas para fazer uma condição “IF”
- GATE: serve para indicar quando um DIAGRAMA DE SEQUÊNCIA ACABA e quando COMEÇA OUTRO, pois quando o diagrama armazena grande quantidade de informações, ele precisa de mais de um DIAGRAMA

- TIPOS DE OPERADORES DE INTERAÇÃO:
  - OPCIONAL
  - PARALELA
  - CONDICIONAL
  - LOOP

# MÉTODOS ÁGEIS

Em 1960 a 1970 estava ocorrendo a CRISE DO SOFTWARE nos computadores antigos com baixo poder de processamento, não tinha pessoas especializadas em desenvolvimento de software → assim deixando as empresas com problemas com orçamentos pois estavam dependentes de software → não cumprindo prazo, qualidade, requisitos. Ainda nesse período, um grupo de pessoas que entendiam de software se reuniu e pensaram que precisava de ordem e chegaram à conclusão que precisavam de um trabalho organizado e de forma sistemática.

E em 1980 a 1990, foi criado a área de ENGENHARIA DE SOFTWARE, que tinha um:

- PLANEJAEMENTO CUIDADOSO para o desenvolvimento
- Com uma QUALIDADE FORMALIZADA com METODOS DE ANÁLISE E DESIGN
- Usar Ferramentas que apoiam, como o CASE
- Ter um PROCESSO DE DESENVOLVIMENTO DE SOFTWARE RIGOROSO E DE QUALIDADE

Teve sucesso principalmente em áreas que precisavam, como SISTEMAS CRÍTICOS, como áreas do governo, militar, SISTEMAS DURADOUROS como bancários.

Em 2000, a situação mudou com a GLOBALIZAÇÃO, com a chegada da internet, surgiu um novo tipo de software que é desenvolvido mais rápido, com um ciclo de vida rápido, com um mercado mais dinâmico, com empresas menores produzindo software. E aqui nesse contexto, o TEMPO era mais precioso que a QUALIDADE (AS VEZES O SOFTWARE PODIA SER DESENVOLVIEMTNO EM MENOR TEMPO, MAS COM MENOS QUALIDADE).

E nesse contexto acontecia que:

- ✓ REQUISITOS INSTAVEIS
- ✓ REQUISITOS INICIAIS mudavam, como o sistema devia interagir com outros sistemas, como o usuário vai operar o sistema

Então se concluiu que se aplicasse os METODOS TRADICIONAIS (CASCATA, INCREMENTAL etc.) iria gastar muito tempo planejando do que produzindo, para tal contexto não funciona. Então os desenvolvedores denominaram:

METODOS TRADICIONAIS → PROCESSOS PESADOS

E criaram métodos denominando:

PROCESSOS LEVES → MÉTODOS ÁGEIS

Em 2001, houve o divisor de águas com o MANIFESTO ÁGIL, assinado por um conjunto de desenvolvedores, que descobriram maneiras de desenvolvedores de software e:

| SE VALORIZOU MAIS | MAS TAMBÉM NÃO SE DESVALORIZOU |
|---|---|
| • VALORIZARAM INDIVÍDUOS E INTERAÇÕES<br>• SOFTWARE EM FUNCIONAMENTO<br>• NEGOCIAR/COMUNICAÇÃO COM O CLIENTE<br>• RESPONDER AS MUDANÇAS | • PROCESSOS E FERRAMENTAS<br>• DOCUMENTAÇÃO ABRANGENTE<br>• NEGOCIAR/FAZER CONTRATO<br>• SEGUIR UM PLANO |

Nos MÉTODOS ÁGEIS ocorre:

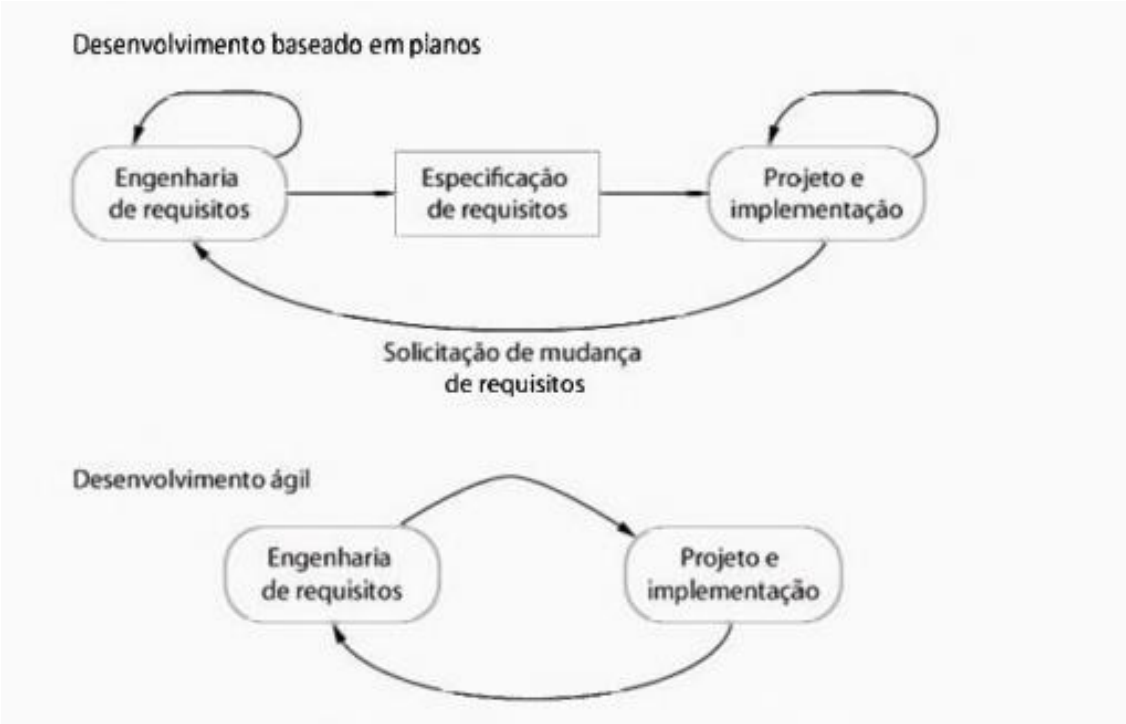
- ENVOLVIMENTO DO CLIENTE: cliente sempre deve estar ligado com a equipe de desenvolvimento;
- ENTREGA INCREMENTAL: é entregue o coração do software, e depois vem os incrementos com o tempo;
- PESSOAS, NÃO PROCESSOS: nos métodos ágeis, se dá liberdade para os desenvolvedores trabalharem, do que na forma sistemática da engenharia de software tradicional;
- ACEITAR MUDANÇAS: projetar o sistema para suportar mudanças no futuro;
- MANTER SIMPLICIDADE: sempre facilitar o código e não complicar.

Nos MÉTODOS ÁGEIS também tem algumas limitações, como:

- Cliente deve passar tempo com os desenvolvedores e ele pode não querer entrar com esse acordo;
- O representante pode não ser capaz de representar a organização como um todo;
- Membros pode ter dificuldade de se adequar com os novos métodos;
- A organização pode não concordar com as ideias;
- Priorizar mudanças pode ser difícil;
- Manter simplicidade pode ser complicado;
- As negociações podem ser complicadas, pois em um contrato as documentações devem ser mais complexas, tal fato que não é prioridade no MÉTODO ÁGIL;
- Os desenvolvedores podem ser imaturos para aplicar o MÉTODO.

Em contrapartida existe algumas VANTAGENS:

**TEM INDEPENDÊNCIA E PRODUTIVIDADE → EQUIPE MAIS MOTIVADA E ENVOLVIDA → TENDO PERSONALIZAÇÃO → COM MAIS QUALIDADE DE SERVIÇO → ECONOMIZANDO TEMPO E RECURSO → GERANDO SATISFAÇÃO DO CLIENTE**




## SCRUM SCRUM SCRUM

É um método ágil focado no gerenciamento do desenvolvimento iterativo, ao invés de técnicas. Consiste em 3 fases principais:

1) **PLANEJAMENTO GERAL:** onde se estabelece os objetivos gerais e arquitetura do software;
2) **SPRINT:** depois têm uma série de ciclos de SPRINT, cada ciclo tem um incremento;
3) **ENCERRAMENTO:** é feita a documentação exigida e avaliado as lições aprendidas.

// OBS: **o SPRINT** é a fase principal, pois é ali que o trabalho vai ser avaliado, os recursos são escolhidos e o software vai ser implementado, o cliente está envolvido no processo e pode pedir para inserir novas funções. Eles têm eventos de duração fixa (2 a 4 semanas). Dentro desse tempo**, 1° AVALIAR: é feito o BACKLOG**, para depois **2°SELECIONAR avaliar e ver as prioridades e os riscos. 3°DESENVOLVER: Equipe +Cliente escolhem funcionalidades e recursos a serem desenvolvidos. 4°REVISAR:** depois que todos estiverem de acordo, a equipe **SEM O CLIENTE** fazem reuniões todos os dias para cerificar como está o andamento do projeto e ver se carece de priorizar trabalho, todos canalizados no SCRUM MASTER. E se encerra um ciclo de SPRINT. //

As vantagens do SCRUM é que A **EQUIPE TEM VISÃO DE TUDO, MELHORANDO A COMUNICAÇÃO → COMO O PRODUTO É DIVIDIDO EM PARTES GERENCIAVEIS E COMPREENSÍVEIS → REQUISITOS INSTAVEIS NÃO GERAM ATRASO → CLIENTES VEEM A ENTREGA DE INCREMENTOS DENTRO DO PRAZO → CLIENTES PEGAM CONFIANÇA NA EQUIPE, GERANDO UMA CULTURA POSITIVA**

## KANBAN KANBAN KANBAN

No KANBAN, é um método ágil na qual as pequenas melhorias são incrementadas de acordo com o processo de desenvolvimento. Foi desenvolvido pela Toyota, ela é uma gestão visual onde o fluxo de trabalho ocorre da esquerda para a direita, sendo que cada etapa do processo é adicionada a um item que gera algum benefício ao cliente, como novas funcionalidades, correção/atualização ou algo do tipo, gerando um valor pelo produto somente no final do processo, quando se obtém o produto completo.

No KANBAN, o quadro vai ser dividido em **"TO DO", "DOING", "DONE",** podendo ser usado posts-its para completar esses espaços.

O KANBAN vai ser aplicado em um QUADRO KANBAN, onde vai ter cartões que pode ser carregado pela equipe. Cada item do projeto é representado por um cartão, que fica separado no quadro KABAN, nele se mostra: **QUEM É O RESPONSÁVEL DO ITEM - DESCRIÇÃO BREVE DA TAREFA - ESTIMATIVA DE TEMPO → → → → → → GERANDO → → → → → → AUTONOMIA A EQUIPE, FAZENDO A CRIAR CONFIANÇA → DISTRIBUINDO INFORMAÇÕES DE FORMA ÚNICA E CONSISTE → EXPLICANDO TODO O PROCESSO A TODAS AS PARTES ENVOLVIDAS → TENDO MAIOR VISIBILIDADE DO PROJETO → REGISTRANDO AS INFORMAÇÕES TRANSMITIDAS E O PROGRESSO DO PRODUTO → CONTROLANDO O FLUXO E CAPACIDADE DE TRABALHO → REDUZINDO DESPERDICÍOS E CUSTOS.**

## LEAN LEAN LEAN

O LEAN é um método ágil na qual vai ter foco no cliente, entregando o produto ao cliente de forma mais rápida, evitando desperdícios. Tendo origem no sistema de produção da Toyota **COM O OBJETIVO DE REDUZIR O DESPERCÍDIO.** Nesse método, a produção do produto é divido em partes para que os operários com poucas habilidades possam realizar o serviço e para isso usa-se o operário de alta precisão e um trabalho personalizado.

Ele vai trabalhar com alguns princípios, sendo:

1. **ELIMINAR DESPERDÍCIO:** vai ser elaborado uma linha do tempo que vai envolver desde o planejamento do software até a entrega dele para o cliente, desse modo, observando quais são os fatores desnecessários (desperdícios) e eliminando-os. **//DESPERDÍCIO É TUDO AQUILO QUE VAI IMPEDIR DO CLIENTE RECEBER OQUE ELE DESEJA!!!!!!//**
2. **INTEGRAR QUALIDADE:** parte do preceito que para um software ter qualidade, é preciso ter o controle do processo desde o início do seu desenvolvimento, inspecionando o produto após a incrementação de uma nova funcionalidade. É importante usar técnicas de TEST DRIVEN DEVELOPMENT para garantir que um código simples e limpo tenha valor para o cliente.

3. **CRIAR CONHECIMENTO:** mesmo com as empresas já com longa data de experiência, elas sempre geram novos conhecimentos a partir de um novo processo de desenvolvimento. É importante ter um processo que incentive o aprendizado sistemático, pois como nos métodos tradicionais, ficar preso em produção de documentação, pode impedir que os desenvolvedores criem práticas, melhorem aprendizados e consequentemente o processo de criação, tais esses comum nos métodos ágeis, mais especificamente no Lean.
4. **ADIAR COPROMETIMENTOS:** primeiro deve evitar criar decisões irreversíveis, porém um software não precisa ser totalmente flexível, e caso haja essas decisões, não pode ocorrer delas serem questionadas em cima no prazo de entrega.
5. **ENTREGA RÁPIDO:** parte do pressuposto que o desenvolvedor tem que produzir e entregar o software tão rápido com o intuito que não dê tempo de o cliente mudar algum dos seus requisitos. Envolvendo a questão que se o software for produzido de maneira rápida e eliminando desperdícios, impacta no custo, na qual o mais baixo sempre vai ser o fator primordial na escolha das empresas.
6. **RESPEITAR AS PESSOAS:** é de extrema importância ter respeito sobre o trabalho de outras pessoas, se o desenvolvedor acha que uma funcionalidade deve ser feita, o outro não pode impedi-lo so porque não acredita naquilo. Todos devem trabalhar juntos para achar uma solução!
7. **OTIMIZAR O TODO:** No Lean é importante otimizar todo o fluxo de valor, todos os processos devem ser otimizados para que o valor completo se torne um valor aceitável.

## XP EXTREME PROGRAMING XP

No EXTREME PROGRAMING, é destacado os pilares de:

1. **COMUNICAÇÃO:** pois o item primordial no desenvolvimento de um software em equipe é a comunicação, estão o XP mantém um fluxo de comunicação, como testes de unidade programação em pares e estimativa de esforço de cada tarefa;
2. **SIMPLICIDADE:** deve tratar o código com mais simplicidade, pois se acontece algum erro é mais fácil arrumar depois;
3. **FEEDBACK:** o cliente deve star envolvido para avaliar oque já foi feito e assim passa oque ele acha e assim descobrir um erro que deve ser arrumado ou alguma função a ser adicionado;
4. **CORAGEM:** é preciso ter coragem para descobrir e revelar um erro no projeto, e caso for necessário, apagar tudo e refazer o código tudo do zero para evitar que fique com algum erro ou algo tudo pendurado em cima do outro;
5. **RESPEITO:** todo a equipe deve assumir a responsabilidade perante os seus atos e confiar no parceiro que ele cumprirá a missão que foi imposta. Considerando o desenvolvedor também como uma pessoa, aceitando compromissos de responsabilidades.