

# Requisitos

**Os requisitos expressam as necessidades e restrições dos utilizadores que são colocadas no sistema.**

Um requisito, normalmente, é:

- Uma condição ou uma capacidade que alguém precisa para resolver um problema ou para alcançar um objetivo.
- Uma condição ou uma capacidade que deve ser verificada ou possuída por um sistema ou por uma componente do sistema para satisfazer um contrato, norma, especificação, ou outros documentos formalmente impostos.
- Uma representação documentada de uma condição ou capacidade.

Os requisitos podem ser divididos em:

1. Requisitos Funcionais
2. Requisitos Não-Funcionais

A classificação de um requisito como funcional ou não funcional depende do ponto de vista do observador.

Um requisito que para um interveniente pode ser visto como sendo funcional, pode ser considerado como não funcional para um interveniente diferente.

## Requisitos Funcionais

**Um requisito funcional descreve uma funcionalidade a ser disponibilizada aos utilizadores do sistema.**

Este tipo de requisitos não deve mencionar qualquer questão tecnológica.

Idealmente, os requisitos funcionais devem ser independentes dos aspectos de concepção e implementação.

Colectivamente, o conjunto de requisitos funcionais deve ser completo e coerente.

O conjunto de requisitos é:

- **coerente**, se não houver contradições entre os seus elementos.
- **completo**, se considerar todas as necessidades que o cliente deseja ver satisfeitas.

## Requisitos Implícitos

Os requisitos que são óbvios são frequentemente esquecidos e como tal não são documentados, nem negociados.

Uma vez que o cliente pode não estar ciente destes requisitos, eles são geralmente referidos como requisitos implícitos.

**Um requisito implícito é um requisito incluído pela equipa de desenvolvimento, com base no conhecimento do domínio que possui, apesar de não ter sido explicitamente solicitado pelas partes interessadas.**

Uma exigência explícita refere-se a uma exigência que foi solicitada pelos clientes e que está representada na documentação.

## Requisitos Não-Funcionais

**Um requisito não funcional corresponde a um conjunto de restrições impostas ao sistema a ser desenvolvido.**

Inclui restrições de tempo, restrições no processo de desenvolvimento, ou adoção de normas.

Exemplo de restrição tecnológica: "o produto deve ser desenvolvido em C++"

Um requisito não-funcional não altera a essência das funcionalidades do sistema.

Os requisitos funcionais permanecem os mesmos, independentemente dos requisitos não-funcionais associados ao sistema.

**Os requisitos não funcionais são aplicáveis ao sistema como um todo e não apenas a algumas das suas partes.**

Geralmente, os requisitos não-funcionais não podem ser modularizados.

Os requisitos não-funcionais são frequentemente propriedades emergentes do sistema em questão.

Uma propriedade emergente de um sistema é uma propriedade que pode ser associada com o sistema como um todo, mas não individualmente a cada um dos seus componentes.

Se o sistema for concebido apenas com base nos requisitos funcionais, ele pode existir como uma entidade monolítica.

**Requisitos não-funcionais são cruciais para decidir a arquitetura do sistema.**

O cumprimento de um requisito não-funcional não pode ser alcançado de forma isolada.

Não se pode maximizar um determinado requisito não-funcional sem sacrificar alguns outros requisitos não-funcionais.

O nível seleccionado para a satisfação de um determinado requisito não funcional afeta a satisfação de outros. Classificação de requisitos não funcionais:

- **requisitos do produto:** caracteriza aspectos do comportamento do próprio sistema (fiabilidade, desempenho, eficiência, portabilidade, usabilidade, testabilidade, e legibilidade).
- **requisitos organizacionais:** vêm de estratégias e procedimentos estabelecidos no contexto do processo de fabrico do sistema ou da organização cliente (normas de processo e requisitos de implementação).
- **requisitos externos:** têm origem em fatores externos ao sistema e ao processo de desenvolvimento (interoperabilidade, requisitos legais e éticos).

Tipos de requisitos não funcionais:

1. **Aparência:** o aspecto e a estética do sistema.
2. **Usabilidade:** a facilidade de utilização do sistema e tudo o que permite uma experiência mais amigável para o utilizador.
3. **Desempenho:** aspectos de velocidade, tempo real, capacidade de armazenamento, e correcção de execução.
4. **Operacional:** características sobre o que o sistema deve fazer para funcionar corretamente no ambiente onde está inserido.
5. **Manutenção e suporte:** atributos que permitem reparar ou melhorar o sistema e acrescentar novas funcionalidades.
6. **Segurança:** questões relacionadas com o acesso, confidencialidade, proteção e integridade dos dados.
7. **Culturais e políticas:** fatores relacionados com a cultura e hábitos das partes interessadas.
8. **Legal:** leis que se aplicam ao sistema para que este possa funcionar.

## Requisitos do Sistema e do Utilizador

Os requisitos do utilizador estão relacionados com o domínio do problema e são normalmente expressos numa linguagem natural.

Um requisito do sistema é orientado para o domínio da solução e é uma especificação detalhada de um requisito, geralmente sob a forma de um modelo formal do sistema.

## Engenharia de Requisitos

O termo engenharia de requisitos é relativamente novo.

Designa todas as atividades relacionadas com a descoberta de requisitos, negociação, documentação, e manutenção.

A engenharia de requisitos consiste no estudo de um problema que leva ao desenvolvimento do sistema, antes de tomar qualquer ação de concepção ou implementação.

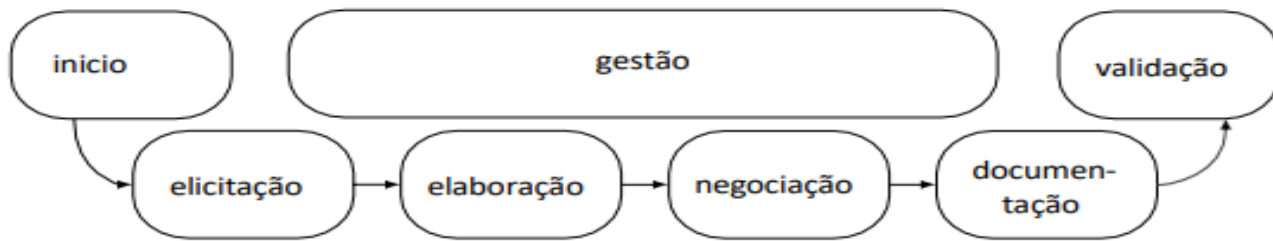
A engenharia de requisitos é o processo de descoberta, análise, documentação e verificação dos serviços e restrições relacionadas com o funcionamento e desenvolvimento de sistemas de software.

A engenharia de requisitos procura assegurar os três seguintes objectivos:

- Todos os requisitos relevantes são explicitamente conhecidos e compreendidos ao nível de detalhe pretendido.
- Um acordo razoável e amplo sobre os requisitos é obtido entre as partes interessadas.
- Todos os requisitos estão devidamente documentados, em conformidade com os formatos e modelos estabelecidos.

**A engenharia de requisitos determina o que o sistema deve fazer para satisfazer as necessidades dos utilizadores e não como deve ser construído.**

**É desejável manter os requisitos estritamente separados das suas próprias soluções.**



- **Início** – Alguém deve iniciar o processo, com base em alguma necessidade ou expectativa de negócio. No final, o engenheiro de requisitos deve ser capaz de descrever qual é a visão do cliente e o retorno do investimento.
- **Elicitação** – Esta atividade trata da forma como os requisitos devem ser capturados. As técnicas de elicitação de requisitos devem:
  - ◆ identificar as fontes dos requisitos.
  - ◆ ajudar os vários intervenientes a descrever correctamente os requisitos.
 Esta actividade é intrinsecamente comunicacional, uma vez que requer uma interação profunda com as partes interessadas
- **Elaboração** – Esta atividade tem como objetivo analisar e classificar os requisitos solicitados, mas ainda não tratados. É habitual organizar os requisitos em grupos coesos. O analista deve intervir, sempre que os requisitos:
  - ◆ não fazem sentido
  - ◆ estão em contradição entre eles
  - ◆ são incoerentes
  - ◆ são incompletos
  - ◆ são vagos.
- **Negociação** – É inevitável que surjam situações de conflito entre as exigências. O seu resultado pode ter um impacto significativo na aceitação do sistema final. Outra forma de lidar com conflitos consiste em adoptar técnicas de priorização, para sustentar a escolha do subconjunto de requisitos a ser implementado em cada instante.
- **Documentação** – Os documentos de requisitos servem como referência principal para as fases subsequentes do processo de desenvolvimento. O documento de requisitos está organizado de acordo com duas perspectivas distintas:
  - ◆ requisitos do utilizador, que descrevem as expectativas e as necessidades dos utilizadores
  - ◆ requisitos do sistema, que estabelecem o acordo entre o cliente e a equipa de desenvolvimento.
- **Validação** – O objectivo é assegurar que os requisitos definem o sistema desejado pelo cliente.
- **Gestão** – Os requisitos estabelecidos estão em constante mudança. A atividade de gestão de requisitos procura ajudar a equipa de desenvolvimento a identificar, controlar e rastrear os requisitos e as suas alterações.

## Escrever Requisitos numa Língua Natural

O uso de requisitos escritos numa linguagem natural é comum em projetos de engenharia.

Promove a comunicação entre os vários intervenientes.

A escrita é uma tarefa sujeita a erros e problemas, uma vez que o que é escrito muitas vezes é interpretado de uma forma diferente da pretendida.

Foi descrito um conjunto de recomendações práticas para escrever bons requisitos em inglês.

Ter os requisitos escritos de uma forma clara e padronizada resulta em vantagens para todos os participantes num determinado projecto.

A estrutura de um modelo padrão para documentar os requisitos, com 27 secções possíveis, foi discutida.

São discutidas questões relacionadas com o ambigüismo em línguas naturais.

A ambiguidade acontece quando existem duas ou mais interpretações possíveis para uma frase.

Sumário 36/36 A ambiguidade é abordada para realçar alguns aspectos a serem tidos em conta quando se escrevem requisitos.

# Elicitação

**Requisitos de Elicitação (Alocação de Requisitos)** – Permite compreender quais são os requisitos de um determinado sistema. Permite compreender as necessidades e expectativas que as partes interessadas têm em relação a um determinado sistema.



## Identificação dos Stakeholders

**Participante do Sistema** – Alguém que pode ser materialmente afetado pela implementação desse sistema. Uma parte interessada negativa é alguém que deseja que o sistema não seja desenvolvido, e pode estar em oposição pacífica à hostilidade ativa em relação ao desenvolvimento do sistema.

## Identificação dos Utilizadores

**Usuário** – Qualquer pessoa que opera e interage diretamente com o sistema, sempre que este esteja em operação efetiva no seu ambiente.

## Identificação dos Clientes

**Cliente** – A entidade que encomenda e paga pelo desenvolvimento de um sistema.

## Identificação dos Consumidores

**Consumidor** – Alguém que paga pela aquisição de um sistema, sempre que disponível.

## Técnicas para Alocação de Requisitos

Competências:

- Questionar
- Observar
- Discutir
- Negociar
- Suportar

Grupo de Técnicas:

- Marketing
- Psicologia e Sociologia
- Desenho Participativo
- Fatores humanos e interação homem-máquina
- Qualidade
- Métodos Formais

Técnicas de acordo com o grupo de pessoas:

- Individuais:
  - ◆ Entrevistas
  - ◆ Inquéritos
  - ◆ Introspecção
- Grupos:
  - ◆ Dinâmica de grupo
- Artefactos
  - ◆ Análise de domínio
  - ◆ Protótipo
  - ◆ Persona

# Arquitetura de Software

## Principais Conceitos

Um princípio é uma lei abrangente e fundamental, doutrina ou suposição.

Os princípios de concepção de software são noções chave que fornecem a base para muitas abordagens/conceitos.

- abstracção
- acoplamento e coesão
- decomposição e modularização
- encapsulamento e ocultação de informação
- separação da interface e implementação
- suficiência, completude e primitivismo
- separação das preocupações

## Concepção de Software

A concepção ou invenção de um esquema para transformar uma especificação de software de computador em software operacional. A atividade que liga os requisitos à codificação e aos testes.

## Arquitetura de Software

Um conjunto de estruturas necessárias para raciocinar sobre o sistema, que compreende elementos de software, relações entre eles, e propriedades de ambos.

As estruturas estáticas de um sistema definem os seus elementos de tempo de concepção interna e a sua disposição.

As estruturas dinâmicas de um sistema definem os seus elementos de tempo de execução e as suas interacções.

As propriedades fundamentais de um sistema manifestam-se de duas maneiras diferentes:

- O comportamento externamente visível define as interações funcionais entre o sistema e o seu ambiente.
- Uma propriedade de qualidade é uma propriedade não-funcional.

Os requisitos funcionais não determinam a arquitectura.

As arquitecturas são influenciadas por:

- as partes interessadas
- a organização em desenvolvimento
- os antecedentes e experiência dos arquitectos
- o ambiente técnico

Uma arquitetura de software é o resultado de influências técnicas, empresariais e sociais.

Cada projeto enfrenta riscos, por isso não há uma única forma correcta de fazer arquitectura de software.

As escolhas de arquitectura são mais importantes quando:

- O espaço de solução é pequeno.
- O risco de falha é elevado.
- Existem requisitos de atributos de qualidade difíceis.
- Um novo domínio está a ser considerado.

A fase de concepção pode ser dividida em duas partes:

- O design de arquitetura é as partes macroscópicas do design, tais como módulos e como eles estão ligados.
- O desenho detalhado cobre tudo o resto.

O resultado do design é um conjunto de artefactos que restam as decisões que foram tomadas; a lógica de cada decisão não trivial é explicada.

Cada componente desempenha um papel específico no sistema.

Minimizar as dependências entre os componentes é importante para criar uma arquitetura com um acoplamento solto.

## Boas Arquitecturas

Uma arquitetura de software de 3 níveis permite aos sistemas de TI localizar alterações e lidar com cargas transacionais.

Uma arquitetura de processos de cooperação é bem adequada aos sistemas operativos porque isola as falhas.

Os desenvolvedores devem prestar atenção à funcionalidade, mas um sistema também tem atributos de qualidade.

A arquitetura de um sistema permite ou inibe qualidades tais como segurança ou desempenho.

Arquitetura e funcionalidade podem ser equilibradas. A arquitetura é na sua maioria ortogonal para a funcionalidade do sistema.

Uma má escolha de arquitetura pode tornar a funcionalidade e os atributos de qualidade difíceis de alcançar.

É provável que a arquitetura exija mais atenção em sistemas com grande escala ou alta complexidade.

O que faz uma boa arquitetura?

1. A arquitetura deve ser o produto de um pequeno grupo de arquitectos com um líder identificado.
2. O arquitecto deve basear a arquitectura numa lista prioritária de atributos de qualidade.
3. A arquitetura deve ser documentada usando pontos de vista.
4. Os pontos de vista devem abordar as preocupações dos intervenientes mais importantes.
5. A arquitetura deve ser avaliada pela sua capacidade de fornecer os importantes atributos de qualidade do sistema.
6. A arquitetura deve prestar-se a uma implementação incremental.

Uma arquitetura presuntiva é uma família de arquitecturas que domina um domínio particular.

Os desenvolvedores nesse domínio podem ter de justificar uma escolha que difere da arquitetura presuntiva.

Uma arquitetura de referência é uma família de arquitecturas que descreve uma solução arquitetónica para um problema.

As arquiteturas presuntivas têm sucesso porque são uma boa combinação para os riscos comuns no domínio.

## Riscos Arquitetónicos

### Abordagem Orientada ao Risco

O conceito de fracasso é central para o processo de concepção, e é pensando em termos de evitar o insucesso que os projetos bem sucedidos sejam alcançados.

A abordagem orientada para o risco ajuda os programadores a construir software de qualidade rapidamente e a baixo custo.

A abordagem orientada para o risco ajuda a responder a duas questões:

- Quanto trabalho de arquitetura de software deve ser feito?
- Que técnicas devem ser usadas?

A abordagem orientada para o risco segue três passos:

1. Identificar e priorizar os riscos
2. Seleccionar e aplicar um conjunto de técnicas
3. Avaliar a redução do risco

Os arquitetos não o querem fazer perder tempo com técnicas de baixo impacto e ignorar os riscos de ameaça ao projeto.

Os arquitectos querem fazê-lo construir sistemas de qualidade tomando um caminho que passe o seu tempo de forma mais eficaz e abordar os riscos através da aplicação de técnicas de concepção, apenas quando são motivados por riscos.

### Riscos

Na engenharia, o risco é normalmente definido como a probabilidade de falha vezes o impacto dessa falha

$risco = probabilidade\ de\ falha \times impacto$ .

Os projetos enfrentam muitos tipos diferentes de riscos.

Tipos de riscos:

- Riscos de gestão de projetos:
  - ◆ Desenvolvedor de chumbo atropelado por autocarro
  - ◆ Necessidades do cliente não compreendidas
- Riscos de engenharia de software:
  - ◆ Não tem uma alta escalabilidade
  - ◆ Há falhas na análise das mensagens de alerta
  - ◆ O sistema pode deixar de funcionar após uma alteração
- Riscos prototípicos:
  - ◆ Domínio de TI – problema complexo e mal compreendido
  - ◆ Domínio de sistemas – Desempenho, segurança
  - ◆ Domínio Web – Segurança, escalabilidade

Técnicas de risco:

- Engenharia de software:
  - ◆ Padrões de design
  - ◆ Modelagem de domínio
  - ◆ Modelagem de rendimento
  - ◆ Análise de segurança
  - ◆ Prototipagem
- Outras áreas de engenharia:
  - ◆ Cálculo do stress
  - ◆ Teste do ponto de ruptura
  - ◆ Análise do therman
  - ◆ Testes de fiabilidade
  - ◆ Prototipagem

Uma técnica específica é boa para reduzir alguns riscos, mas não outras.

**A abordagem orientada para o risco utiliza este princípio: os esforços de arquitetura devem ser proporcionais ao risco de fracasso.**

O esforço gasto na concepção da arquitetura deve ser proporcional aos riscos enfrentados pelo projeto.

## Estilos de Design

**Na concepção evolutiva, a concepção do sistema cresce à medida que o sistema é implementado.**

**No desenho planeado, a arquitectura é detalhada antes da construção.**

**Poucas pessoas defendem a concepção planeada de todo um sistema de software.**



**Um estilo híbrido é um design planeado mínimo.**

É possível equilibrar o design planeado e evolutivo.

Alterações futuras às exigências podem ser tratadas através de um design local, ou com um design evolutivo.

A abordagem orientada para o risco é compatível com os três estilos de design.

No BDUF, esse tempo é adiantado.

Em NDUF, significa fazer o desenho de arquitetura durante o desenvolvimento, sempre que um risco se torna suficientemente grande.

Aplicá-lo ao LDUF é uma combinação dos outros dois.

## Táticas de Design

### Táticas

Os requisitos de qualidade especificam as respostas do sistema de software para realizar os objetivos de negócio.

As táticas são usadas pelo arquiteto para criar um projeto.

As táticas ligam os requisitos de atributos de qualidade com as decisões arquitetônicas.

**Uma tática é uma decisão de concepção que tem impacto em atributos de qualidade específicos.**

A concepção de um sistema consiste numa coleção de decisões. Algumas destas decisões ajudam a controlar as respostas dos atributos de qualidade. Outras garantem a realização da funcionalidade do sistema.

Tipos de táticas:

1. Disponibilidade
2. Modificabilidade
3. Desempenho
4. Segurança
5. Testabilidade
6. Usabilidade

Uma falha ocorre quando o sistema já não fornece um serviço que seja consistente com a sua especificação.

**As táticas de disponibilidade visam evitar que as falhas se tornem falhas ou pelo menos reduzir os efeitos da falha e tornar a reparação possível.**

Todas as abordagens para abordar a disponibilidade envolvem algum tipo de:

1. redundância,
2. monitorização da saúde para detectar uma falha,
3. recuperação quando uma falha é detectada.

A monitorização ou recuperação é automática ou manual.

**Redundância activa** – Todos os componentes redundantes respondem a eventos em paralelo.

**Redundância passiva** – Um componente (o primário) responde aos eventos e informa os outros componentes (os standbys) das atualizações de estado que devem fazer.

**As táticas de desempenho geram uma resposta a um evento que chega ao sistema dentro de algum limite de tempo.**

O evento pode ser único ou um fluxo e é o gatilho para um pedido para efetuar um cálculo.

A latência é o tempo entre a chegada de um evento e a geração de uma resposta ao mesmo.

**Tempo bloqueado** – Um cálculo pode ser bloqueado de usar um recurso devido à sua contenção ou indisponibilidade.

Três categorias táticas abordam o desempenho:

1. procura de recursos
2. arbitragem de recursos
3. gestão de recursos

Dois aspectos principais na procura de recursos são:

- O tempo entre os eventos de um fluxo de recursos
- Quanto de um recurso é consumido por cada pedido

**Aumentar a eficiência computacional** – Um passo no processamento de um evento é a aplicação de algum algoritmo.

A melhoria dos algoritmos diminui a latência.

**Reduzir a sobrecarga computacional** – Se não houver pedido de um recurso, as necessidades de processamento são reduzidas.

## Padrões de Design

Princípios de design:

- Identificar os aspetos da sua candidatura que variam e separá-los do que permanece igual.
- Programa para a interatividade, não para a implementação.
- Princípio aberto-fechado – uma classe/módulo deve ser aberto para extensão, mas fechado para modificação

Um padrão é uma descrição textual de uma solução genérica para um problema recorrente num determinado contexto.

Tipos de padrões de design:

- Criativo
- Estrutural
  - ◆ Estratégico
- Comportamental
  - ◆ Observador

O padrão de estratégia define uma família de algoritmos, encapsula cada um deles e torna-os então intercambiáveis. Ele permite que o algoritmo varie independentemente do cliente que o utiliza.

O padrão do observador define uma dependência de um para muitos entre objetos. Sempre que um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

O padrão decorador atribui responsabilidades adicionais a um objeto de forma dinâmica. Ele fornece uma alternativa flexível à subclassificação para ampliar a funcionalidade.



# Estilos Arquitetônicos

## Estilos

Um estilo arquitectónico no software é análogo a um estilo arquitectónico no edifício (por exemplo, Manuelino).

Um estilo arquitectónico consiste em:

- Um conjunto de tipos de componentes (por exemplo, processo, procedimento) que desempenham alguma função em tempo de execução.
- Uma disposição topológica dos componentes mostrando as suas relações de tempo de execução
- Um conjunto de constrangimentos semânticos.
- Um conjunto de conectores (por exemplo, fluxos de dados, tomadas) que medeiam a comunicação entre um sistema que esteja em conformidade com um determinado estilo deve usar esses tipos, o que restringe o espaço de desenho.

## Restrições

As restrições podem atuar como guias que apontam um sistema para onde você quer que ele vá.

Pode-se pensar num estilo como um conjunto pré-fabricado de restrições que podem ser reutilizadas.

## Platónico vs. Incorporado

Um estilo arquitectónico platónico é uma idealização.

Encontra-se nos livros e só raramente no código fonte.

Existe um estilo arquitectónico incorporado em sistemas reais.

Muitas vezes viola as restrições estritas encontradas nos estilos platónicos.

Pode-se encontrar versões incorporadas do estilo em que os servidores ocasionalmente empurram os dados para os clientes.

Dependendo de como isto é implementado, pode resultar num servidor que depende dos clientes.

## Padrões vs. Estilos

Os padrões de design estão a uma escala menor do que os estilos arquitectónicos.

Os padrões podem aparecer em qualquer parte do seu desenho, e vários padrões podem aparecer no mesmo desenho. Um sistema normalmente tem um único estilo arquitectónico dominante.

Se um sistema tem um estilo arquitectónico clientes-servidor, espera-se ver os componentes cliente e servidor nas vistas de design de nível superior.

O sistema também poderia utilizar padrões arquitectónicos, tais como o padrão REST.

## Catálogo de Estilos

	Viewtype	Elements & Relations	Constraints / guide rails	Qualities Promoted
<b>Layered</b>	Module	Layers, uses relationship, callback channels	Can only use adjacent lower layers	Modifiability, portability, reusability
<b>Big Ball of Mud</b>	Module	None	None	None, but many inhibited
<b>Pipe-and-Filter</b>	Runtime	Pipe connector, filter component, read & write ports	Independent filters, incremental processing	Reconfigurability (modifiability), reusability
<b>Batch-Sequential</b>	Runtime	Stages (steps), jobs (batches)	Independent stages, non-incremental processing	Reusability, modifiability
<b>Model-Centered (Shared Data)</b>	Runtime	Model, view, and controller components; update and notify ports	Views and controllers interact only via the model	Modifiability, extensibility, concurrency
<b>Publish-Subscribe</b>	Runtime	Publish and subscribe ports, event bus connector	Event producers and consumers are oblivious	Maintainability, evolvability
<b>Client-Server &amp; N-Tier</b>	Runtime	Client and server components, request-reply connectors	Asymmetrical relationship, server independence	Maintainability, evolvability, legacy integration
<b>Peer-to-Peer</b>	Runtime	Peer components, request-reply connectors	Egalitarian peer relationship, all nodes clients and servers	Availability, resiliency, scalability, extensibility
<b>Map-Reduce</b>	Runtime & allocation	Master, map, and reduce workers; local and global filesystem connectors	Divisible dataset amenable to map & reduce functions, allocation topology	Scalability, performance, availability
<b>Mirrored, Farm, &amp; Rack</b>	Allocation	Varies	Varies	Varies: Performance, availability

# Refactoring

Refactoring é uma forma disciplinada de limpar o código que minimiza as hipóteses de introdução de bugs.

Quando você refatorar está a melhorar o design do código que tem sido usado.

Refactoring altera um sistema software de tal forma que não altera o comportamento externo do código mas melhora a sua estrutura interna.

Com a refactoring pode pegar num mau, até mesmo caótico, design, e retrabalhá-lo em código bem desenhado.

**Refactoring é uma série de pequenos passos, cada um dos quais altera a estrutura interna do programa sem alterar o seu comportamento externo.**

Razões para refactoring:

- O código é duplicado.
- Uma rotina é demasiada longa.
- Um laço é demasiado longo ou demasiado profundo.
- Uma classe tem pouca coesão.
- Uma interface de classe não fornece um nível consistente de abstracionismo.
- Uma lista de parâmetros tem demasiados parâmetros.
- As mudanças dentro de uma classe tendem a ser compartimentadas.
- As mudanças requerem paralelos modificações a múltiplas classes.
- As hierarquias sucessórias têm de ser modificadas em paralelo.
- As declarações do caso têm de ser modificadas em paralelo.
- Os dados relacionados que são utilizados em conjunto não estão organizados em classes.
- Uma rotina usa mais características de outra classe do que da sua própria classe.
- Um tipo de dados primitivo está sobrecarregado.
- Uma turma não faz muito.
- Uma cadeia de rotinas passa dados de vagabundos.
- Um objeto intermediário não está a fazer nada.
- Uma turma é excessivamente reclusa com outra.
- Uma rotina tem um nome pobre.
- Os membros dos dados são públicos.
- Uma subclasse utiliza apenas uma pequena percentagem das rotinas dos seus pais.
- Os comentários são usados para explicar o código difficult..
- São utilizadas variáveis globais.
- Uma rotina usa o código de configuração antes de uma chamada rotina ou código de takedown ser uma chamada rotina.
- Um programa contém um código que parece ser necessário um dia.

A refactoring é um complemento ao design.

A refactoring pode ser uma alternativa ao design inicial

- Os códigos do programador, e depois refactorá-lo em forma (XP prática).
- Os engenheiros de software fazem o design inicial, mas não tentam encontrar a solução; eles precisam de uma solução razoável.

A refactoring pode levar a desenhos mais simples sem sacrificar flexibilidade.

Isto torna o processo de desenho mais fácil e menos estressante.

## Modelação

A modelação é um ingrediente essencial em todos os campos da engenharia.

É um tarefa altamente criativa

A modelação é o processo de:

- Identificar conceitos apropriados.
- Selecionar abstrações para construir um modelo que reflita adequadamente um dado universo de discurso.

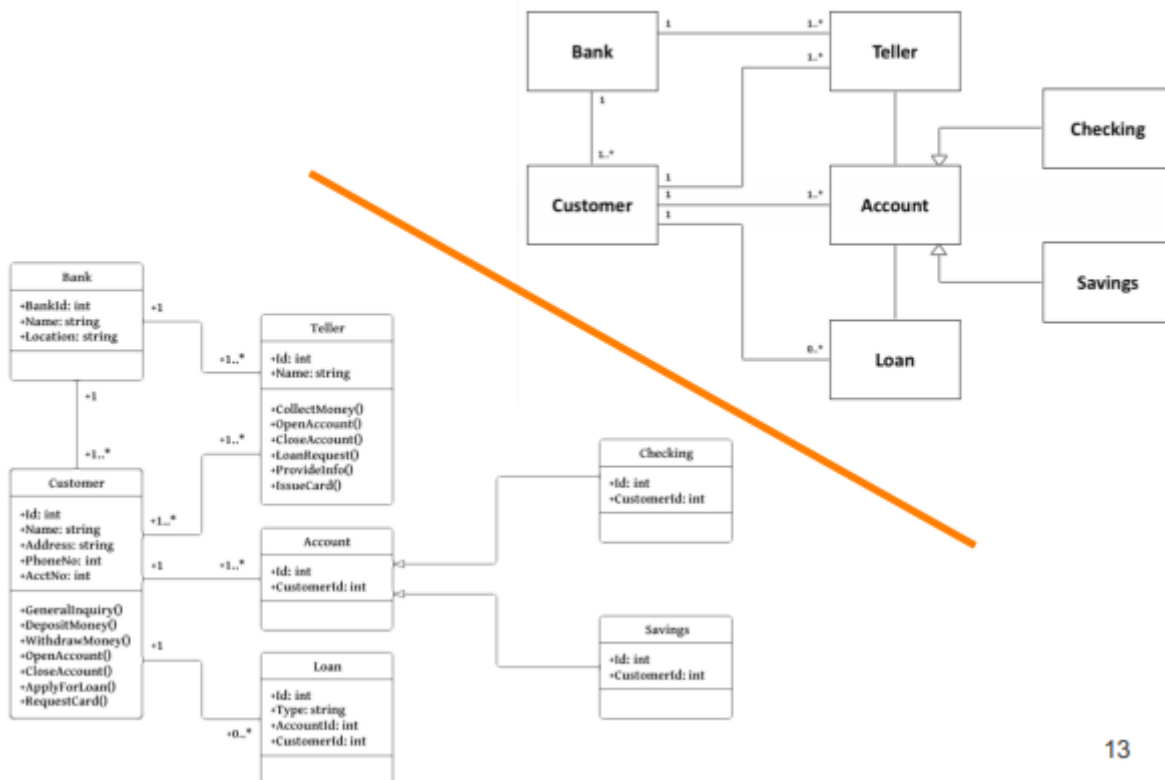
A modelação permite a utilização rentável do modelo em vez do objeto/processo do mundo real para algum fim.

Para ser útil, um modelo não deve representar todos os aspectos da realidade.

A modelação está relacionada com a abstração, simplificação e formalização.

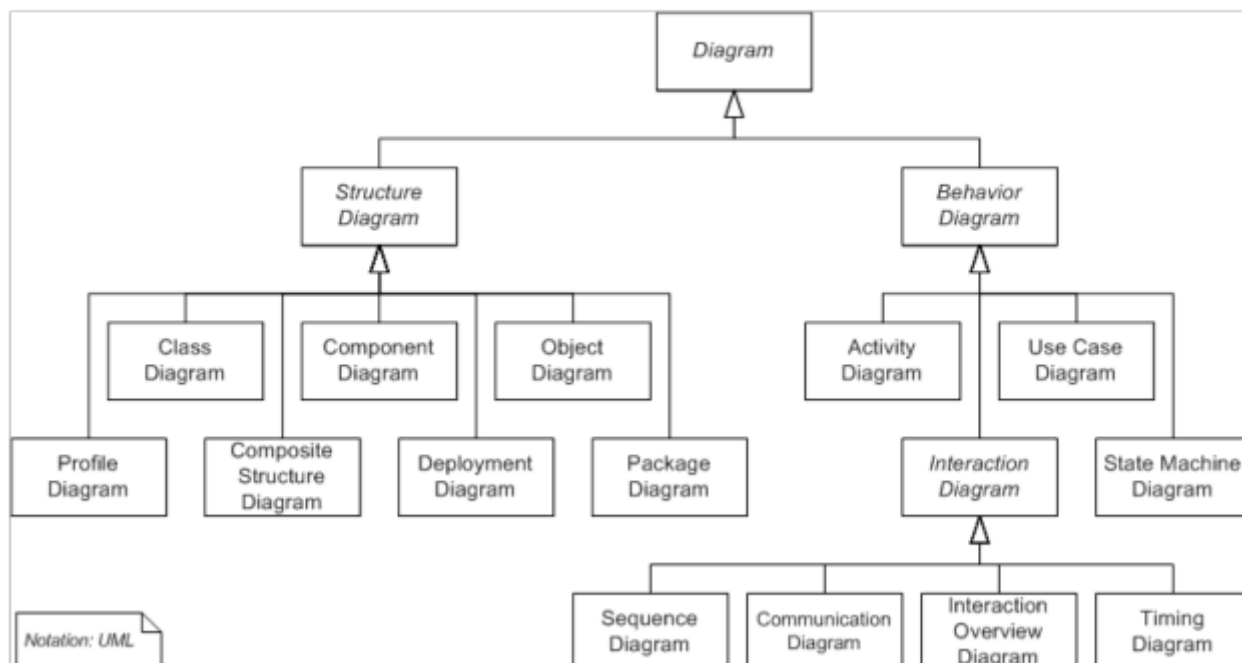
A operação inversa da abstração é o refinamento ou concretização.

## Modeling: Abstraction



13

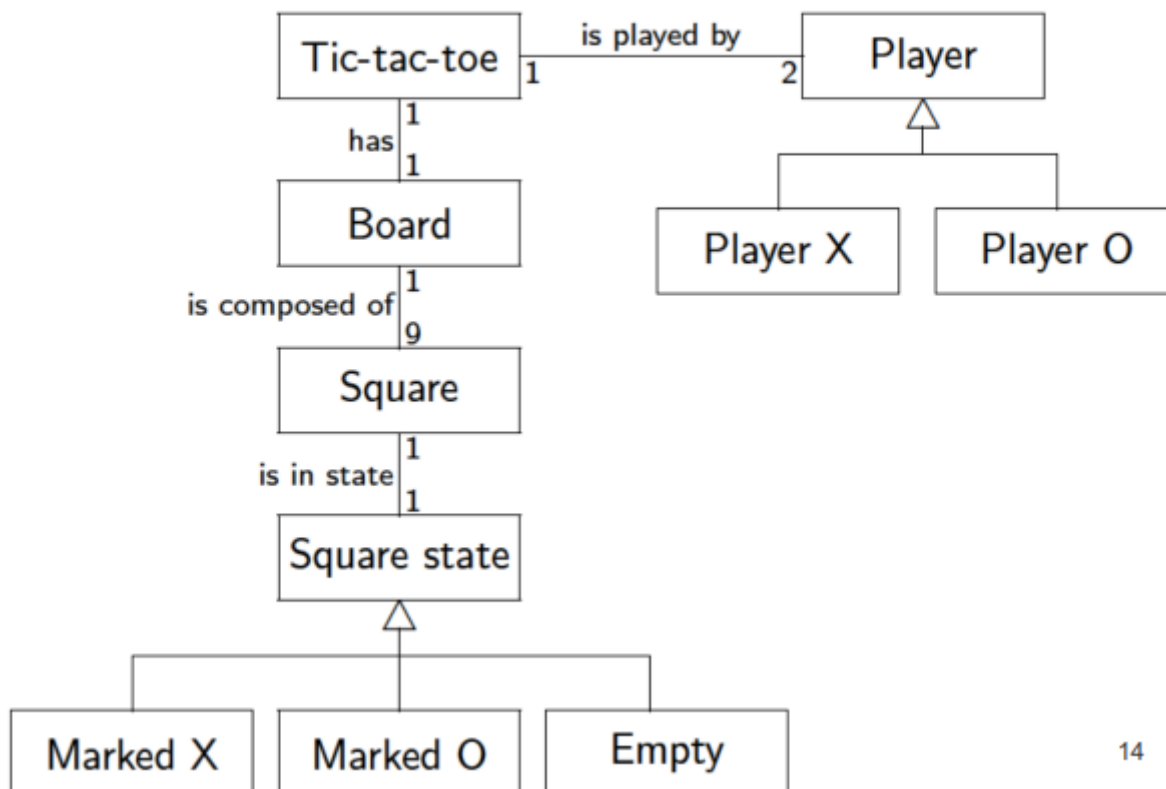
## UML: Diagrams



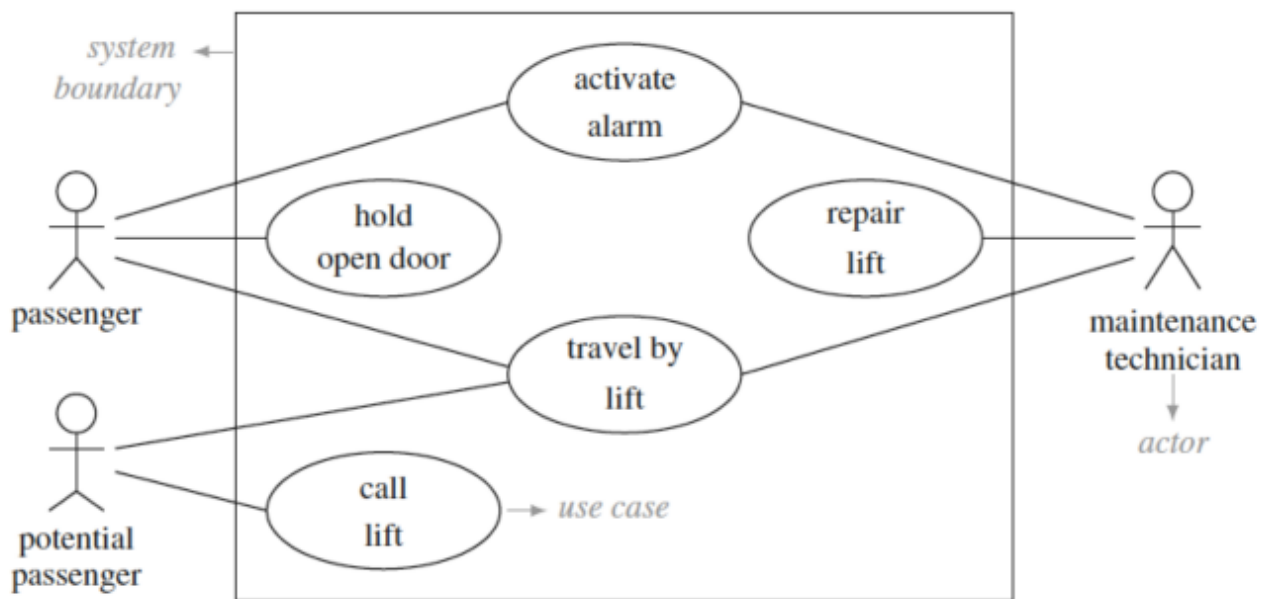
## Essential UML

Model	Purpose
Domain	Describe the vocabulary, concepts of the domain and characteristics of the systems that can be developed for the considered domain
Use case	Describes the proposed functionalities of a given system
Interaction	Show how the various objects or entities collaborate, emphasising the flow of control and data among them
Class	Present a set of concepts, types and classes and the respective relations
State	Specify the behaviour of an entity or indicate the various states (or modes) through which it transits throughout its life
Activity	Show the control flow among the activities of a process

### Essential UML: Domain model



## Essential UML: Use case model



Os *use cases* não modelam processos/fluxos de trabalho.

Os *actors* não fazem parte do sistema. Eles são posicionados fora dos limites do sistema.

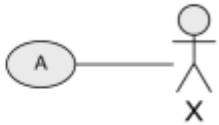
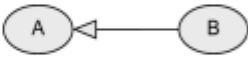
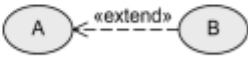
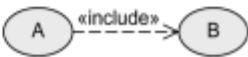
Muitos *use cases* pequenos com o mesmo objetivo podem ser agrupados para formar um (mais abstrato) *use case*.

As várias etapas fazem parte dos *use cases*, não *use cases* separados → **Não** há decomposição funcional.

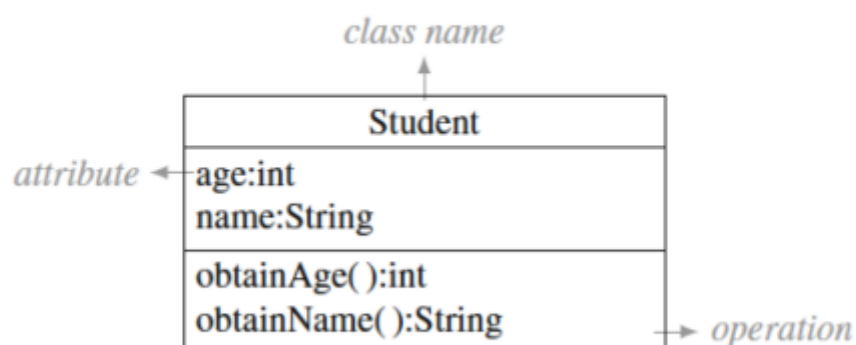
## Essential UML: Use case model

Name	Notation	Description
System		Boundaries between the system and its users
Use case		Unit of functionality of the system
Actor		Role of the users of the system

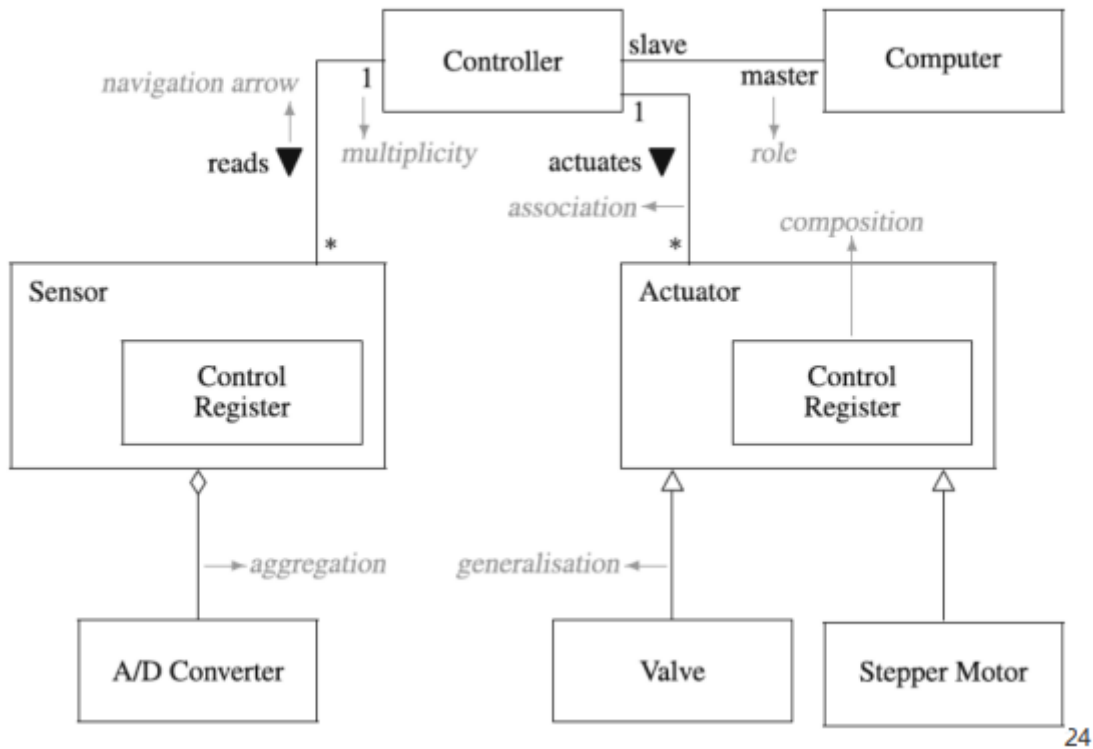
## Essential UML: Use case model

Name	Notation	Description
Association		Relationship between use cases and actors
Generalization		Inheritance relationship between actors or use cases
Extend relationship		B extends A: optional use of use case B by use case A
Include relationship		A includes B: required use of use case B by use case A

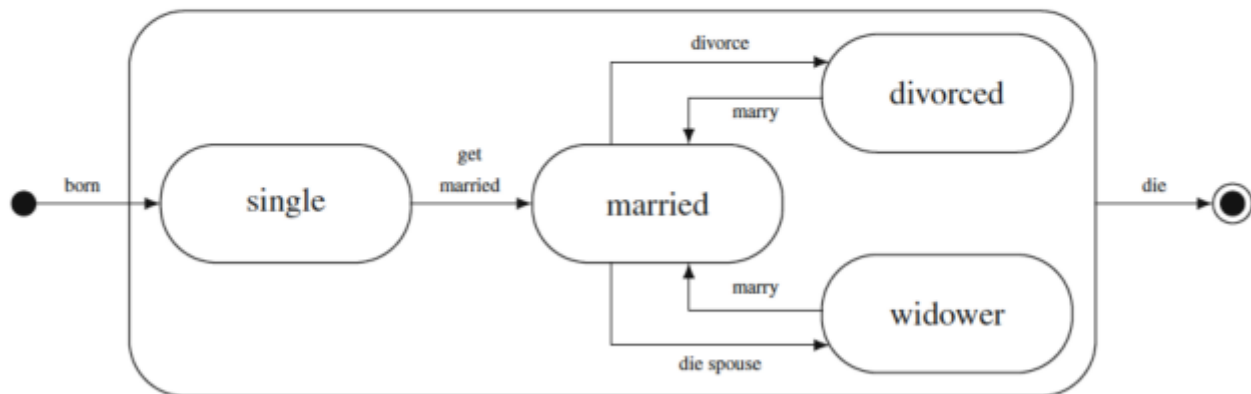
## Essential UML: Class model (1/2)



## Essential UML: Class model (2/2)



## Essential UML: State model



## Essential UML: Activity model

