universidade de aveiro

deti departamento de electrónica, telecomunicações e informática

Introdução à Engenharia de Software

Teóricas

Resumos

Gonçalo Matos, 92972

Licenciatura em Engenharia Informática

3.º Ano | 1.º Semestre | Ano letivo 2020/2021

Última atualização a 15 de janeiro de 2021

Este documento é baseado nos *slides* teóricos do professor José Luís Oliveira. Fontes adicionais são referenciadas no início dos capítulos onde foram utilizadas.

Índice

1. Introdução à Engenharia de Software

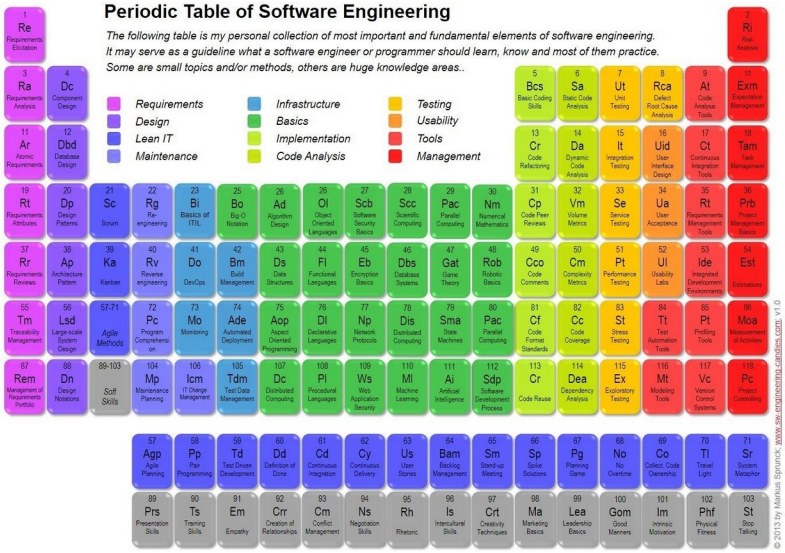
Slides teóricos e CrashCourse

Definida pelo IEEE como a aplicação de uma abordagem sistemática, disciplinada e quantificável ao desenvolvimento, operação e manutenção de *software* e por Roger S. Pressman por uma tecnologia que abrange um **processo**, um conjunto de **métodos** e uma lista de **ferramentas**.

Em suma, esta área de estudos foca-se na compreensão da organização de um projeto de *software*, procurando fornecer métodos de trabalho e de seleção de ferramentas como um processo industrial.

Uma das suas fundadoras, Margaret Hamilton, disse: "It's like preventative healthcare, but it's preventative software."

É por isso uma disciplina bastante abrangente, cobrindo matérias que vão desde análise de requisitos, passando por conhecimentos de infraestrutura ou mesmo de gestão, que em conjunto, geram valor ao projeto, em diversos aspetos.



Redução da complexidade, ao dividir os problemas, cujas partes são resolvidas de forma independente;

Minimização dos riscos custos e tempo de desenvolvimento, através do planeamento e aplicação de boas práticas;

Gestão de grandes projetos, mais uma vez possível por causa do planeamento e estratégia;

Fiabilidade do produto, devido aos testes e manutenção constantes nesta disciplina;

Processos estandardizados, críticos para o desenvolvimento de *software* eficiente.

Do ponto de vista académico a ES pode não ter grande importância uma vez que os projetos desenvolvidos focam-se mais no desenvolvimento do que no produto final, pelo que muitas vezes a UI e a documentação são menosprezados. No entanto, no mundo empresarial há utilizadores finais que não o desenvolvedor, que está a desenvolver um produto que serve de suporte ao negócio e por isso requer investimento. É aqui que a ES se torna indispensável.

A sua aplicação é fundamental desde o nascimento e ao longo de toda a vida de um projeto, uma vez que contrariamente aos mecânicos, os problemas de *software* surgem durante o desenvolvimento (uma aplicação estável terá uma baixa probabilidade de erros, mas as aplicações precisam de estar em constante atualização…).

Por isso, apesar de dar resultados positivos, a ES implica um grande investimento, principalmente em testes e na menutenção dos sistemas.

Caracteriza-se por um **processo sistemático**, que separa o processo de desenvolvimento em diferentes **fases**, cada uma focada num aspeto chave.

Arquiteto de software

Tem como função a definição das técnicas e a escolha das opções de desenho de alto nível, incluindo *standarts* de código, ferramentas e plataformas utilizadas no projeto. Deve para isto apresentar características de liderança, resolução de problemas, comunicação, pesquisa e promoção de boas práticas.

Fases do desenvolvimento

O desenvolvimento segundo a ES divide-se em várias fases, que apesar de iguais na teoria, na prática devem ser adaptadas à realidade em que se inserem.

Um arquiteto de *software* que vá trabalhar num sistema pré-existente, tem de se adaptar à sua realidade, mesmo que esta não seja perfeita do seu ponto de vista, uma vez que na maioria dos cenários será impossível fazer alterações às bases fundamentais do mesmo (por exemplo a linguagem utilizada ou a *frameowork* em que se insere).

Uma empresa que tenha uma plataforma baseada nos serviços do Windows difilmente irá mudá-los para Linux.

Todas elas envolvem a **equipa**, um conceito fundamental e indispensável à ES, que deve ser aberta à partilha e discussão de ideias. Outros conceitos fundamentais são a **gestão de tempo**, **eficiência**, **qualidade**, **ética** e **método de trabalho**.

2. Processo de software

Slides teóricos e Agility.im

O **processo de software** caracteriza-se como um **guia** para as atividades, ações e tarefas que são necessárias para a construção de um *software* com qualidade, definindo práticas e técnicas e de gestão para a aplicação de métodos, ferramentas e pessoas ao longo do desenvolvimento.

"*It is better not to procees at all, than to proceed without method.*" - Descartes

Existem vários tipos de processos, mas todos dão resposta a pontos comuns, definindo a **especificação**, **desenho e implementação**, **validação** e **evolução** através da especificação de atividades ordenadas.

O que vamos fazer?

Quem vai fazer?

Como vamos fazer?

Quando vamos entregar?

Inclui papéis, fluxos, procedimentos, padrões e modelos.

Cada projeto de *software* terá um processo adequado ao seu desenvolvimento. Um bom processo permite aumentar a produtividade dos memebros menos experientes sem prejudicar o desempenho dos mais proficientes.

Modelos de processos de software

Os modelos de processos de software descrevem diferentes abordagens ao desenvolvimento de software.

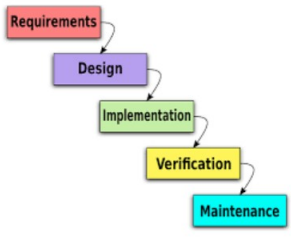
Apesar de terem características similares, distinguem-se entre si pelos intervalos de tempo entre as diferentes fases, pelos critérios que delimitam as fases e pelo resultado de cada uma.

Waterfall

Este é um modelo **preditivo**, onde a especificação está completamente separada do desenvolvimento.

Prós
Fases são indepentes e processadas separadamente;
Fácil de gerir e planear (diferentes etapas especializadas em cada etapa);

Contras
Uma vez terminada uma fase, não há volta atrás (difícil gerir mudanças);
Cliente só participa na definição dos requisitos e na aceitação (do produto final), ou seja, é excluido do desenvolvimento;
Não há um produto "utilizável" antes do fim do ciclo.



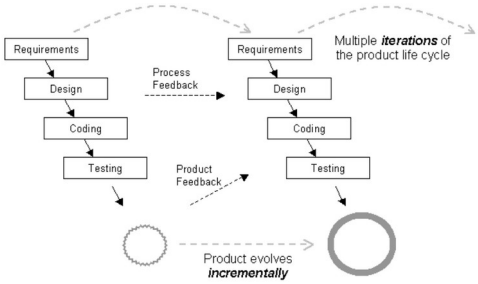
Incremental

Este é um modelo **ágil** bastante recente e focado nas necessidades atuais da industria. Caractariza-se pela evolução incremental do produto através de uma série de etapas.

Os requisitos são assim divididos e em cada interação implementados total e sucessivamente, sendo integrados de forma crescente nas entregas ao cliente.

Prós
Entrega + rápida;
Gestão de alterações + fácil;
Envolve + o cliente;

Contras
Iterações são rígidas (mini *waterfall* em cada iteração);
Iterações não se sobrepõe, pelo que em cada ciclo todas as equipas têm de entrar, o que aumenta a dificuldade de gestão dos recursos humanos;
Dada a rapidez do desenvolvimento, geralmente descora-se a documentação e o *refactoring*;
Um desenvolvimento mal gerido pode levar a iterações "infinitas", cujo custo ultrapassa o benefício.



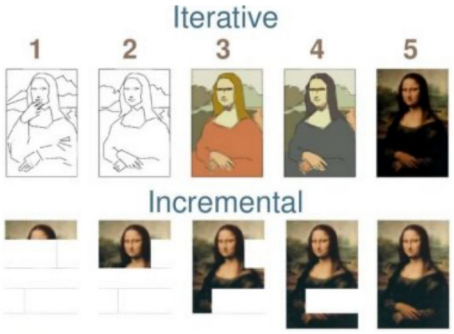
Iterativo

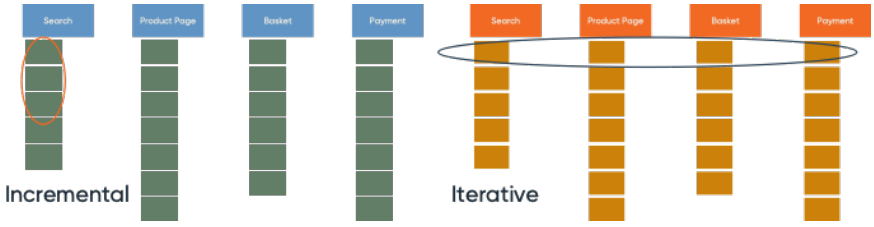
Neste modelo **ágil** os requisitos são implementados parcialmente em cada iteração, sendo aprimorados ao longo das várias etapas de desenvolvimento.

Modelo incremental vs. iterativo

O modelo **incremental** foca-se em dividir o sistema em várias partes, sendo em cada etapa desenvolvida uma dessas partes, que são entregues no final de cada etapa completas e funcionais.

Por outro lado, o **iterativo** desenvolve em cada etapa um pouco de todas as partes de forma parcial, completando-las sucessivamente em cada iteração.





Comparando estes dois modelos aplicados à construção civil, num modelo incremental teríamos o desenvolvimento em cada etapa de uma divisão da casa, totalmente mobilada, pintada e pronta a habitar, enquanto que no modelo iterativo desenvolviamos primeiro os pilares de sustenção, depois toda a estrutura, paredes, depois janelas, pavimentos, pintura, mobilar e estores, e por aí em diante.

Outros modelos

Baseado em componentes (COTS)

Baseado na reutilização de componentes já existentes (no nosso caso, código).

Métodos formais

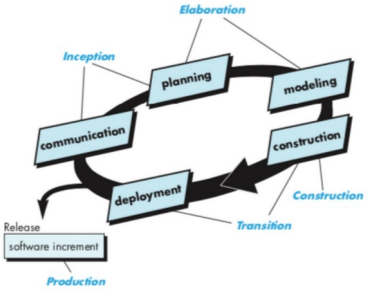
Baseado em modelos matemáticos.

Desenvolvimento de software aspect-oriented (AOSD)

Modelo focado no desenvolvimento de aspetos. Ajuda a resolver problemas de modularidade.

Unified Process

Modelo focado em casos de uso, centrado na arquitetura, iterativo e incremental, altamente ligado à UML (*Unified Modeling Language*).



O UP é ilustrado na imagem ao lado.

Tipos de modelos de processos de software

Dependendo do tipo de planeamento que requerem, os modelos de processos de *software* podem ser classficiados em dois tipos.

Modelos preditivos (plan-driven)

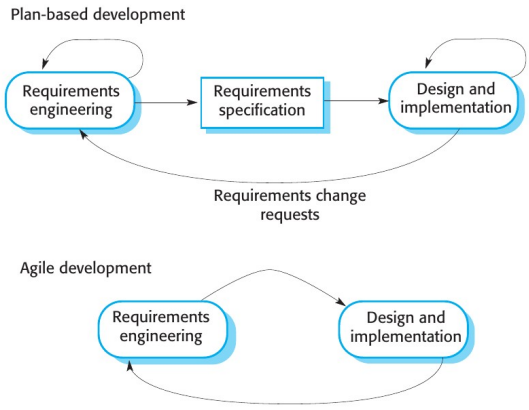
Quando as atividades são planeadas em antecipação.

Uma vez iniciadas, não é suposto voltar atrás.

Modelos ágeis

Focados num planeamento incremental.

Processo pode ser alterado durante o desenvolvimento de acordo com os requisitos do cliente (negociação durante o processo de desenvolvimento).



Na prática, são geralmente aplicados modelos que combinam um pouco dos dois.

Modelos ágeis

Houveram tempos em que o *software* era lançado na forma de executáveis que eram corridos localmente nos computadores dos utilizadores. No entanto, com a evolução da tecnologia e a sua utilização crescente, a necessidade de evolução dos produtos tornou-se constante perante um mercado dinâmico (público-alvo instável) que faz com que os requisitos alterem rapidamente.

Os **modelos ágeis** vieram dar resposta a este problema dos modelos preditivos, oferecendo uma resposta focada no código em detrimento do *design* e baseada numa abordagem iterativa, que consegue entregar *software* funcional num curto período de tempo.

Dá-se preferência a um código bem escrito e legível em detrimento da documentação.

A abordagem não é completamente iterativa. Na realidade é uma mistura entre o modelo iterativo e incremental.

Este modelo teve origem no **manifesto ágil**, que defende:

Indivíduos e interações mais que processos e ferramentas;

Software em funcionamento mais que documentação abrangente;

Colaboração com o cliente mais que negociação de contratos;

Responder a mudanças mais que seguir um plano.

Por vezes o foco no funcionamento do *software* pode levar a uma documentação dispersa e a dificuldades na estimativa de custos.

Benefícios

Facilidade na mudança dos requisitos, entrega contínua de *software* e baixo custo no *refactoring*.

Desvantagens

Documentação é esparsa, se requisitos forem pouco claros é difícil prever o resultado esperado e riscos inesperados podem afetar o desenvolvimento.

3. Desenvolvimento ágil de software

Slides teóricos, aula teórica assíncrona e Scrum.org sobre o SCRUM

Como foi abordado no último capítulo, a metodologia ágil veio oferecer uma alternativa às tecnicas de desenvolvimento mais clássicas, destacando-se pela **rapidez** com que consegue colocar um produto (ou parte dele) no mercado através de uma metodologia **iterativa** cujo **foco é o código** acima do desenho (planeamento e documentação).

Técnicas de desenvolvimento ágil

As ténicas de desenvolvimento ágil caracterizam-se por **intercalar** os processos de **especificação**, **design** e **implementação** em várias iterações que são desenvolvidas em conjunto com os *stakeholders*, originando sucessivas versões do sistema que são entregues frequentemente.

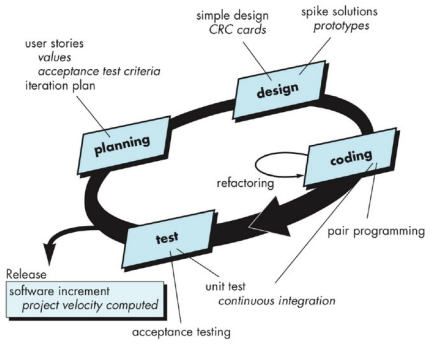
Podem ser utilizadas ferramentas de suporte para facilitar o desenvolvimento e a entrega rápida.

Extreme Programming (XP)

Esta é a corrente de desenvolvimento mais conhecida e aplicada de desenvolvimento ágil e caracteriza-se por:

- ✔ Criação de novas versões várias vezes por dia;
- ✔ Incrementos entregues a cada 2 semanas;
- ✔ Todos os testes têm de ser executados (e validados) para cada versão do produto.

Implementa quatro fases: **planeamento, design, código** e **testes**.



Geralmente utiliza uma abordagem orientada a objetos (OO).

Define ainda várias práticas de programação.

Planeamento incremental. Planeamento feito no início de cada iteração.

Entregas pequenas. Pequenas funcionalidades podem originar várias versões (internas).

Design simples. Fazer apenas o *design* estritamente necessário (sem grandes gráficos).

Desenvolvimento *test-first*. Criar testes unitários para cada funcionalidade nova antes de a implementar.

Refactoring. Apesar do produto estar a funcionar, devemos estar sempre recetivos ao *refactoring* do código escrito nas funcionalidades/iterações anteriores.

Desenvolvimento em pares. Duas pessoas a trabalhar na mesma tarefa.

Responsabilidade coletiva. Os pares de desenvolvedores trabalham em todas as áreas do projeto, podendo cada par mudar qualquer parte do projeto. Cria-se a noção de responsabilidade coletiva.

Integração contínua. Ao concluir cada tarefa, o seu código deve ser integrado de imediato no sistema para ser testado e do conhecimento dos restantes programadores.

Ritmo sustentável. Atrasos e adiamentos não são aceitáveis.

Cliente presente. Deve estar sempre disponível para consulta pela equipa de desenvolvimento um representante do utilizador final. Este membro deve estar integrado na equipa de desenvolvimento.

Algumas destas práticas são difíceis de implementar na prática do ponto de vista da **gestão**, pelo que na realidade este não costuma ser integrado 100% de acordo com o seu modelo teórico.

Por exemplo há quem defenda que o **desenvolvimento em pares** é mais dispendioso do que individual.
O **cliente presente** a 100% na equipa de desenvolvimento também é algo que geralmente não faz sentido.

Definição dos requisitos através de user-stories

As ***user-stories*** são escritas em pequenos textos, fáceis de ler e interpretar que são "partidos" pela equipa de desenvolvimento em pequenas **tarefas de implementação**, a partir das quais é atribuido um **custo** com base na complexidade/tempo de implementação.

Uma vez classificadas e com base no número limite de pontos, cabe ao cliente a escolha das histórias a implementar em cada iteração.

A sua utilização é fundamental para que as tarefas a realizar sejam claras para ambas as partes. Não só para a equipa de desenvolvimento poder gerir a carga de trabalho a desenvolver em cada iteração, mas também para o cliente ter noção dos recursos que necessita de empenhar (número de pontos) para cada história, que na sua cabeça pode parecer uma coisa simples e prioritária, mas na realidade até é bastante complexa e este chega à conclusão que afinal pode ser implementada mais tarde.

É boa prática a definição de um *template* para a criação de *user-stories*.

Papel/objetivo/benefício Obriga o cliente a considerar quem vai beneficiar, qual o objetivo e porque o quer atingir.

Limitações e necessidades Situações relevantes para a tarefa.

Definição da conclusão Definir exatemente como é validado que a tarefa está concluída.

Tarefas de engenharia Como a tarefa interage com outros serviços (internos ou externos).

Custo Medida sobre o tempo/complexidade do desenvolvimento da tarefa.

Refactoring

Geralmente, a tendência do programador é de criar um *design* preparado para a mudança, aplicando tempo e esforço na antecipação das alterações futuras de forma a poupar custo quando estas forem implemetadas.

Por exemplo se o cliente pedir um visualizador de imagens GIF, fará sentido desenvolver uma solução modular que no futuro possa ver integrados outros formatos de imagem. Mas, e se no futuro o cliente nunca pedir para alargar a solução a outros formatos? Terá sido trabalho em vão...

A XP defende que as mudanças não podem ser antecipadas e que deve ser feito apenas o código estritamente necessário à implementação, propondo alterações contínuas ao código à medida que estas são precisas.

No entanto, alterações mais profundas ao nível da arquitetura são bastante dispendiosas!!!

Dadas as características de desenvolvimento incremental da XP, o *refactoring* é frequentemente aplicado na remoção de **código duplicado**, melhoria da **legibilidade do código** e substituição de código *inline* por **novos métodos**.

Desenvolvimento test-first

A XP defende esta prática, que consiste na criação de **testes unitários** em conjunto com o cliente antes da implementação das funcionalidades, que devem ser corridos para validar cada pequena alteração.

Geralmente aplica-se a **automação de testes**, que consiste na sua execução de forma automática e sucessiva após cada alteração.

Uma ferramenta que permite realizar a automação de testes é a Junit.

No entanto, esta abordagem tem algumas desvantagens. Desde logo o **tempo dispendido** na escrita de testes, a **preferência pela programação** em detrimento dos testes pelos desenvolvedores, a **dificuldade da escrita incremental de testes** (p.e. não faz sentido chamar utilizadores humanos para testar a mesma interface a cada pequena alteração) e por fim a **dificuldade na completude dos testes** (ou seja, a abrangência de todos os cenários possíveis).

Desenvolvimento em pares

Ao implementar este desenvolvimento conjunto que é aliado a uma **rotatividade** de equipas, fomenta-se um **conhecimento mais abrangente** do sistema por parte de toda a equipa.

Este ***total ownership*** reduz o impacto da saída de um elemento da equipa.

O código torna-se mais **robusto**, uma vez que o fator do erro humano é reduzido para metade.

Encoraja o ***refactoring***, pois ao termos pares a trabalhar sobre o mesmo código torna-se mais fácil de o manter correto e elegante.

Esta prática é pouco aplicada no mundo empresarial. No entanto, muitas implementam uma variação que consiste na validação asssíncrona do código. Ou seja, alguém valida o código de outra pessoa depois de esta o ter escrito.

No entanto, há alguns estudos que defendem que a escrita de código a pares é mais eficiente do que individual, pelo que o investimento até pode ser rentável.

Gestão de projetos ágeis

Cada projeto ágil é gerido por um **gestor de projeto**, cuja responsabilidade é garantir que os prazos e orçamento definidos são cumpridos.

Esta tarefa está geralmente associada a um elevado grau de planeamento. No entanto, em metodologias ágeis, é adaptada à natureza incremental dos projetos.

Scrum

Uma das metodologias de gestão de projetos mais aplicada hoje em dia é o SCRUM. Esta define três fases de gestão.

Planificação Objetivos gerais do projeto e design da arquitetura de *software*.

Sprints Ciclos de desenvolvimento incrementais.

Término Aceitação, entrega e conclusão do projeto.

Para a compreender, é necessário saber a sua terminologia.

Equipa de desenvolvimento Grupo de no máximo 7 pessoas.

Incremento Cada *sprint* gera um incremento de *software*, que é "entregável".

Product backlog A *to-do list*. Inclui *user-stories*, requisitos de *software*…

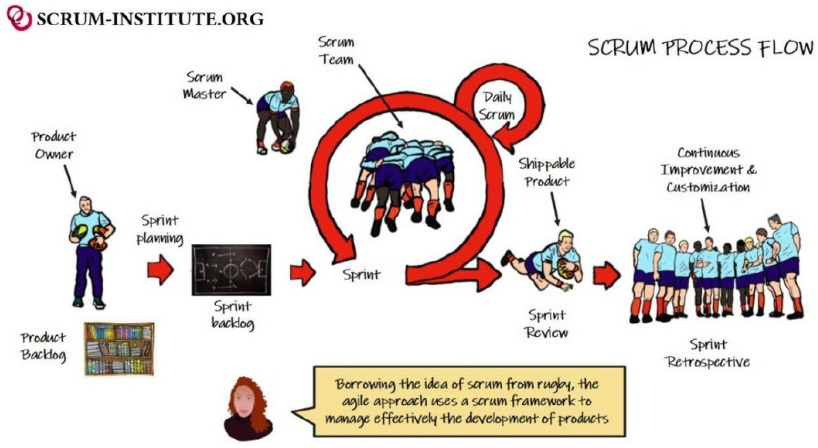
Product owner Pessoa cuja função é identificar as *features* e requisitos, prioritizando-las

Scrum meetings Reuniões diárias onde é partilhado e analisado o trabalho feito e em execução.

Scrum master Pessoa que garante que equipa está a seguir a metodologia SCRUM de forma adequada.

Sprint Iteração de desenvolvimento (entre 2 e 4 semanas).

Velocidade Quanto esforço pode ser empregue em cada iteração. Capacidade de desenvolvimento.



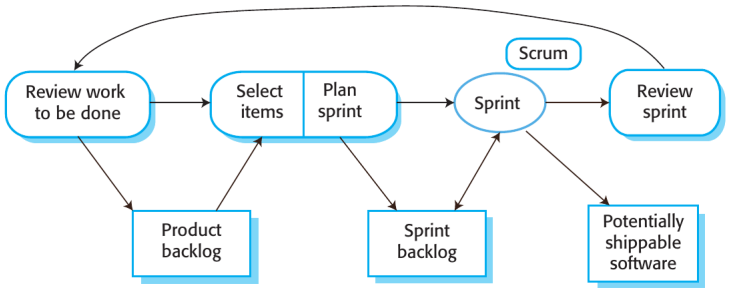
É importante perceber que no SCRUM não há um gestor de projeto. As suas funções são divididas entre a **equipa de desenvolvimento**, o **product owner** e o **scrum master**.

Ciclo Sprint

Cada sprint parte do **backlog**, a partir do qual são escolhidas as funcionalidades a implementar durante a **fase de seleção**, em conjunto com o product owner.

Na **fase de desenvolvimento** a equipa trabalha isolada do cliente, sendo quaisquer comunicações mediadas pelo scrum master.

No **final da sprint** o trabalho é analisado, revisto e entregue aos *stakeholders*.



Funcionamento da equipa

Como já vimos, a equipa junta-se diariamente nas **scrum meetings** para partilhar o estado do seu trabalho, problemas e dúvidas.

Destaca-se a função do **scrum master**, cujo papel passa não só pela marcação das **scrum meetings**, mas também pela gestão do progresso, mediação da comunicação com o cliente e entidades externas à equipa, registod e decisões e do trabalho realizado.

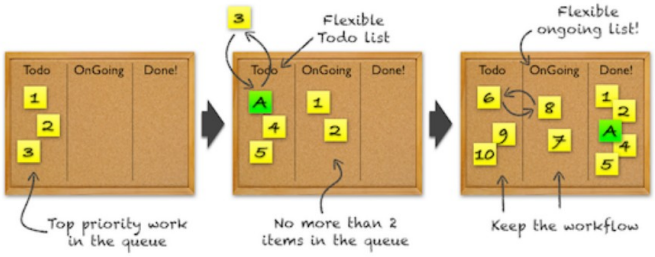
Benefícios

- ✔ Fragmentação do produto em várias peças que possam ser desenvolvidas e geridas de forma isolada;
- ✔ Requisitos instáveis não param o processo, uma vez que não são incluídas no desenvolvimento;
- ✔ Toda a equipa tem uma visão global do projeto, o que facilita a comunicação;
- ✔ No final de cada **sprint**, o cliente pode ver o produto;
- ✔ Fomentação da confiança entre todos os atores.

Kanban

Esta é outra metodologia de gestão de projetos, desenvolvido na Toyota.

É semelhante ao SCRUM no que toca à implementação de pequenas funcionalidades de forma incremental, mas difere deste no que toca à rigidez dos ciclos de desenvolvimento, não impondo quaiquer prazos.



https://project-management.com/from-scrum-to-kanban/

O único limite que impõe é o tamanho da quantidade de trabalho **em desenvolvimento**. Esta limitação tem como objetivo a **redução do tempo** aplicado em cada tarefa, garantir que as tarefas em desenvolvimento **são prioritárias**, reduzir *multi-tasking* e tempos de vantagem de forma a aumentar a qualidade.

O número de tarefas **em desenvolvimento** não deve exceder 1,5 o número de pessoas da equipa.

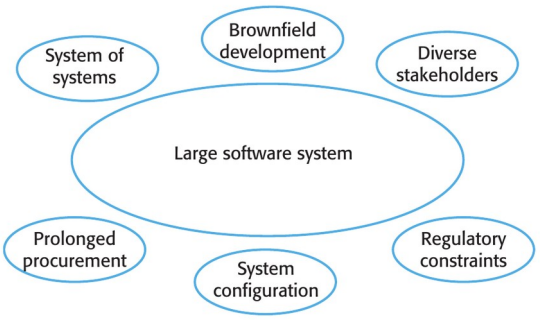
Uma equipa de 4 pessoas não deve trabalhar em mais do que 6 tarefas em simultâneo (4 * 1,5 = 6).

Escalabilidade dos métodos ágeis

Os métodos provaram ser bem sucedidos em pequenas equipas de desenvolvimento que trabalham no mesmo espaço. No entanto, em projetos de média/larga dimensão pode ser necessário incluir uma equipa maior no desenvolvimento que até pode estar espalhada por diferentes localizações geográficas.

Independentemente destas variações, há princípios dos métodos ágeis que devem ser mantidos como o **planeamento flexível**, ***releases* frequentes**, **integração contínua**, **desenvolvimento *test-driven*** e **boa comunicação** entre a equipa.

Problemas da dimensão



Estes sistemas são geralmente compostos por vários sistemas mais pequenos e independentes, sendo que cada um pode ser desenvolvido por uma pequena equipa separada ou não geograficamente das restantes. É fundamental manter a **comunicação**.

Costumam também estar assentes em sistemas pré-existentes (*brownfield systems*) e dependentes de sistemas *legacy*. Estas dependências limitam a abertura à flexibilidade do desenvolvimento.

Os sistemas estão também sujeitos a leis e regulamentações que podem limitar o desenvolvimento.

Sendo um sistema de maior dimensão, o desenvolvimento é mais longo, tornando mais difícil à equipa ter um conhecimento completo e profundo sobre a sua totalidade.

Mais um problema é a diversidade dos *stakeholders*, que vai tornar mais difícil a sua inclusão no processo de desenvolvimento.

Multi-team SCRUM (SCRUM de SCRUM)

Uma solução para a aplicação do desenvolvimento ágil nestes projetos passa pela divisão das equipas em pequenas equipas de desenvolvimento, cada uma implementando a sua metodologia

É fundamental que, apesar de equipas separadas, estas estejam em sintonia e permantente comunicação, uma vez que todas estão a trabalhar com um objetivo comum (o mesmo sistema).

Esta comunicação pode ser assegurada através de uma **scrum meeting** entre os vários **scrum masters**.

4. DevOps

Slides teóricos e aula teórica assíncrona

Como foi explorado no capítulo anterior, a metogologia ágil promove um desenvolvimento contínuo com a entrega de versões funcionais em períodos de tempo regulares. No entanto, para serem integradas na versão de produção do projeto, estas entregas têm de ser *deployed*.

Em muitas empresas, seja por **questões operacionais** ou de **ética** e **privacidade**, no caso de *outsourcing*, as equipas de desenvolvimento e operacionais são independentes. Assim, quando a primeira acaba uma iteração de desenvolvimento, entrega-la à segunda, que assume a tarefa de *deploy* de forma independente.

Equipa de desenvolvimento	Equipa de operações
Procura **inovação**	Por vezes é **conservadora** porque trabalha com versões mais antigas, mas estáveis
São chamados em caso de falha	"Empurram" a culpa das falhas para o desenvolvimento
Trabalho **contínuo**	Trabalho **não contínuo** (apenas quando há uma *release*)

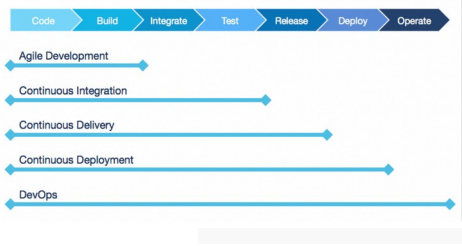
A **solução** para este tipo de cenários é criar equipas **DevOps**, que consiste na junção das duas responsabilidades de desenvolvimento e operações na mesma equipa.

Esta junção pode materializar-se em pessoas diferentes dentro da mesma equipa responsáveis por cada uma, ou em equipas mais pequenas por engenheiros que podem executar as duas tarefas.

Independentemente do modo de funcionamento da equipa, a sua relação deve estar baseada na **comunicação**, gerando ***full ownership*** dos serviços, uma vez que se quebram as barreiras e a equipa é responsável tanto pelo desenvolvimento como pela operação do sistema.

Ciclo de desenvolvimento

O trabalho de desenvolvimento das equipas DevOps caracteriza-se por várias tarefas, inicialmente focadas no **desenvolvimento** de código e depois na sua **operacionalidade**.



Desenvolvimento ágil Planeamento, escrita de código e *build*.

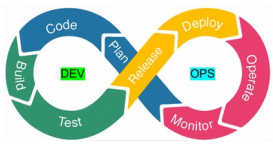
Integração contínua Compilação, revisão, testes em unidade, testes de integração.

Entrega contínua *Release* das soluções criadas em ambiente de testes para aprovação pelo cliente.

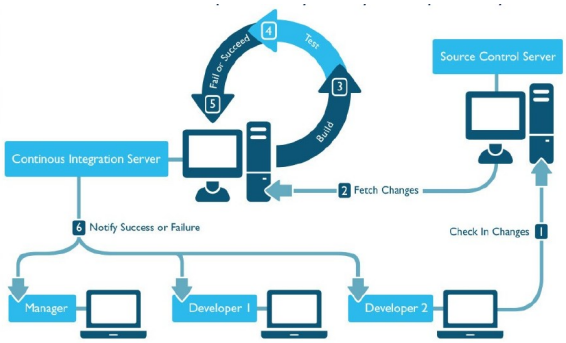
Deploy contínuo *Deploy* em produção da solução testada.

Monitorização contínua Monitorização permanente da aplicação em produção.

Estas várias tarefas são sucessivas. Tratando-se de uma metodologia ágil, no final da sua execução reinicia-se o ciclo.



Integração contínua



Esta fase do ciclo abrange quase a totalidade do **desenvolvimento** desde o planeamento, a codificação, teste do código gerado e integração do projeto, sendo o resultado uma **versão funcional do projeto**.

Ao longo da sua execução, é fundamental que as alterações geradas sejam refletidas no código da equipa o mais cedo possível. Para isto é fundamental a **sincronização constante** com repositórios de gestão de versões.

Esta prática permite a **deteção atempada de erros** e a **redução da intervenção manual**, uma vez que a validação e sincronização de código pode ser automatizada. Esta automação pode ser implementada através de várias ferramentas complementares.

O controlo de versões é uma boa prática. É fundamental a **partilha** código regular, devendo ser feita pelo menos um ***merge* diário** e trabalhar com ***branches* de curta duração**.

A configuração ao nível da construção também é importante. Deve ser garantido que os *scripts* são **independentes do IDE** onde são gerados, que é feita uma ***build* a cada alteração** e por fim que há **limites às violações** dos *standarts*, velocidade, etc.

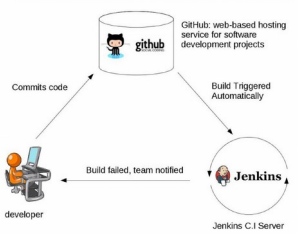
A política da equipa deve estar em sintonia com os princípios do desenvolvimento ágil.

Build script

Responsável por compilar, testar inspecionar e fazer *deploy* do projeto.

O **Maven** é um exemplo, que implementa um ciclo de validação, compilação, teste, *package*, verificação, instalação e finalmente *deploy*.

CI Server



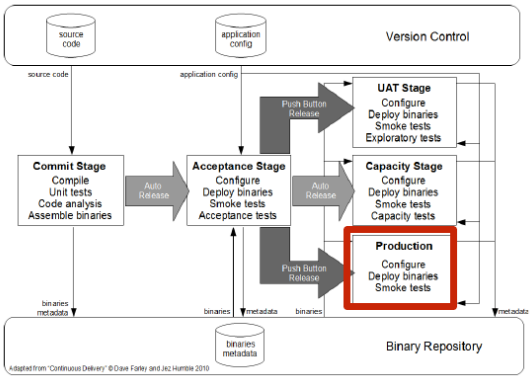
Servidores de integração contínua, que integram modificações feitas no repositório, testando a sua validade (se tem erros).

Esta integração pode ser 100% contínua, ou agendada, sendo executada diariamente à noite, por exemplo.

Entrega contínua

No final desta fase temos código pronto para *deploy*. Sendo a "ponte" entre o desenvolvimento e a operação, é ela que garante uma colaboração próxima entre todos os envolvidos no projeto.

O **Jenkins** permite efetivar esta fase através de uma linguagem declarativa e de uma interface gráfica.



Existem várias estratégias para realizar os testes de aceitação, que podem ser feitos antes ou durante a produção.

Eat your own dog food testes internos pelos funcionários da empresa

Canary releases fazer *release* da nova versão para alguns utilizadores (representativos) apenas

Dark launches semelhate à anterior, mas utilizadores não sabem que estão a ser testados

Gradual rollouts fazer *release* a um número gradualmente crescente de utilizadores

A/B testing fazer *release* para metade dos utilizadores, sem saberem que estão a ser testados

Blue/green deployments fazer *release* da nova versão para todos, mas manter a versão anterior em funcionamento para caso haja problemas na nova ser possível voltar à anterior rapidamente

Infraestrutura definida através do software

Com a criação das equipas DevOps passou a ser bastante comum a utilização de funcionalidades que permitem a gestão da infraestrutura através de código, permitindo não só a sua (re)configuração, como também análise e validação.

Assim, os engenheiros de desenvolvimento veem o seu trabalho de operacionalidade do código facilitado através destes métodos de **automatização**.

Convention over configuration

Este é um conceito bastante importante no mundo das DevOps. Consiste num paradigma de design que agrega as principais configurações de *software* em convenções que podem ser utilizadas durante o desenvolvimento, permitindo acelerar o processo de desenvolvimento ao evitar ao programador a necessidade de definir todas as preferências.

Verifica-se por exemplo quando trabalhamos com *packages*. Sabemos como funciona, mas não precisamos de configurar nada. Evitamos assim a necessidade de escrever todo o *script* que cada *package* fornece.

No Maven por exemplo corremos o comando *make* que faz *build* do projeto. Não escrevemos o código para fazer o *build*. Utilizamos *convetion over configuration*.

Benefícios das DevOps

Velocidade Maior rapidez do desenvolvimento à operação.

Fiabilidade Todo o *pipeline* está integrado e automatizado.

Escalabilidade Mudanças no sistema refletem-se em pequenas alterações nos *scripts* de configuração.

Colaboração Tal como visto na metodologia ágil, devido às responsabilidades partilhadas.

Segurança Dentro dos mecanismos de configuração podem ser garantidas regras de segurança.

Para além da técnica, DevOps é também **cultura**, **automação**, **simplicidade**, **avaliação** e **partilha**.

5. Padrões arquiteturais

Slides teóricos e aula teórica assíncrona

A **arquitetura** é o conceito de mais alto nível de desenvolvimento. Define-se como um conhecimento partilhado que descreve como o sistema se divide em componentes e como é que estes interagem através das suas interfaces.

Diz-se **conhecimento partilhado** porque todos os desenvolvedores do projeto devem ter noção da arquitetura.

Cada componente mais abstrato é composto por vários componentes mais pequenos, mas se estes não forem do conhecimento de todos os desenvolvedores não fazem parte da arquitetura.

Decisões de arquitetura

As decisões arquiteturais têm um **grande impacto** na estrutura do projeto, pelo que implicam decisões bem pensadas e que nem sempre são fáceis de tomar.

Quando vamos construir uma casa a decisão do terreno a comprar, o desenho da casa, a sua orientação são decisões com grande impacto.

Por exemplo as decisões de desenvolver uma aplicação ***web-based*** em vez de um programa de utilizador, de qual a ***framework*** a utilizar, de utilizar REST para **comunicar** entre os componentes ou até de distribuir os componentes na rede para maior **escalabilidade** terá um grande impacto no projeto.

Por isso, embora quando tomadas possam parecer demasiado óbvias e irrelevantes, devem ser **bem documentadas**, identificando as condições e restrições e analizando cada opção com base nas anteriores.

Caso contrário, no futuro podem parecer pouco sustentadas ou óbvias e levar a alterações que vão comprometer o projeto. Isto é defendido pelo **groundhog day anti-pattern**.

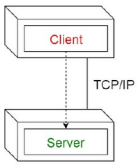
Por exemplo se no projeto da casa tivermos um pilar no meio da sala, quando começarmos a contrui-la vai parecer que não fica ali bem e vai ser ignorado. Mais tarde o primeiro piso desaba porque afinal aquele pilar sustentava grande parte deste piso superior.

A documentação deve ser **centralizada** de forma a ser facilmente consultada por todos os membros da equipa.

É fundamental que a equipa recolha o melhor do conhimento coletivo de forma a atingir uma solução comum e consensual. A arquitetura não deve por isso sem feita em pequenos grupos dentro da equipa, podendo nestes casos levar a uma mistura complexa de ideias sem uma visão clara do que deve efetivamente ser implementado.

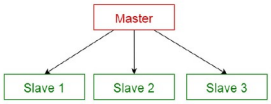
Padrões de arquitetura de software

Cliente-servidor



Esta arquitetura consiste na relação de um para um entre o cliente e o servidor.

Master-slave



Semelhante à anterior no que toca ao número de intervenientes na relação, mas disntinta porque no servidor há um relacionamento interno de master/slave.

Por exemplo aplicações *multi-threaded* ou bases de dados replicadas.

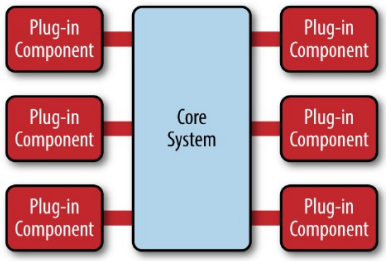
Pipe-filter

É utilizada em processos que exigem *buffering* (começamos numa fonte de dados e acabamos num consumidor de dados, podendo utilizar filros pelo meio) ou sincronismo.

Os compiladores fazem este tipo de processamento em cascata.

Microkernel

Tem por base dois componentes, o ***core system*** e os **módulos** ***plug-in***, os últimos independentes do primeiro, fornecendo extensibilidade, flexibilidade e isolamento das funcionalidades da lógica da aplicação.

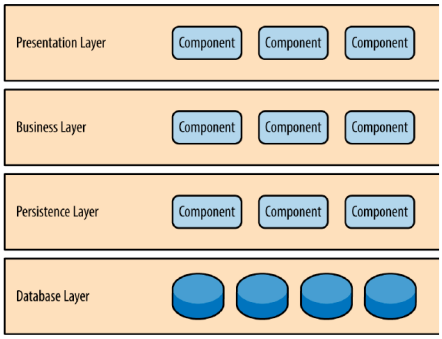


O ***core system*** geralmente providencia apenas as funcionalidades básicas e estritamente necessárias ao funcionamento do sistema. Deve ainda ter conhecimento dos *plug-ins* disponíveis e de como os obter (nome, contratos e protocolos de acesso).

Por exemplo os IDE fornecem um *core system* que depois é complementado pelos *plug-ins* que lhe atribuem novas funcionalidades.

Arquitetura em camadas

Esta é a arquitetura mais comum hoje em dia. Os seus componentes são organizados em camadas horizontais, cada uma com um papel específico na aplicação.

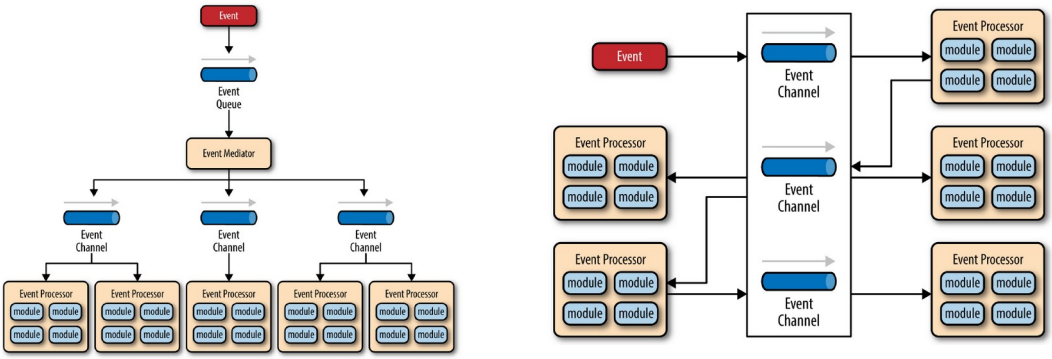


A sua característica principal é a **separação dos papéis** que cada camada desempenha, sendo este específico para cada uma, que lida apenas com a lógica adjacente a si mesma. Cada camada forma assim uma camada de **abstração**.

Por exemplo, a camada de apresentação não precisa de saber como obter os dados do utilizador, tendo como única preocupação a forma como os vai apresentar, a única lógica que implementa.

Arquitetura event-driven

Consiste num padrão de arquitetura distribuída assíncrona utilizado para criar aplicações escaláveis, também conhecido por ***message-driven*** ou ***stream processing***. É composto por componentes individuais desacoplados que recebem e processam eventos assíncronamente.

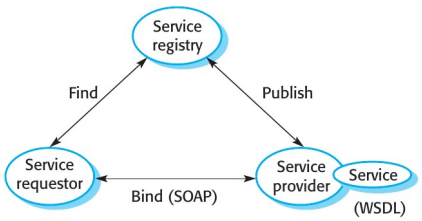


Útil em eventos com várias etapas e que necessitam de orquestração para ser processados.

Para exemplificar como funciona esta topologia, suponhamos que uma pessoa muda de casa e informa a sua seguradora, através de um evento *relocation*. Os passos envolvidos no seu processamento estão contidos num *event mediator*, como mostrado na figura acima, que por cada passo vai criar um *processing event*, que é depois enviado para o canal correspondente, onde será processado pelo seu *event processor*. O processo continua até que todos os passos estejam concluídos.

Arquitetura orientada a serviços (SOA)

Esta arquitetura tem como objetivo desenvolver serviços distribuídos cujos componentes são *standalone* e podem ser utilizados por outros como serviços prontos a consumir (não como bibliotecas).



Caracteriza-se por desenvolver sistemas **desacopaldos**, com **código independente** e utilizados através de um **contrato de utilização do serviço**, potenciando a **reutilização** e **autonomia**.

Por exemplo na UA o sistema de autenticação é utilizado por todos os serviços associados ao ensino. Foi desenvolvido como um componente *standalone* e é utilizado por outros serviços como um serviço (neste caso de auntenticação) como o PACO, o Elearning, o site do NEI…

Pode ainda ser adicionada uma camada de **abstração** quando há vários componentes para o mesmo serviço com um *service registry*. Este é contactado pelo cliente para saber qual componente está disponível.

Por exemplo na UA podiam existir vários sistemas de impressão que trabalhavam de forma complementar para aumentar a redundância. Quando queríamos imprimir contactavamos o registo dos serviços que nos dizia qual o sistema que estava disponível, para o qual era enviado o pedido de impresão.

SOAP (Simple Object Access Protocol)

Este é um protocolo baseado em XML que foi pioneiro na definição de comunicação entre aplicações, independentemente da linguagem de programação em que são desenvolvidos.

Por ser uma solução muito abrangente e demasiado estandardizada (tem estrutura pré-definida) foi considerada demasiado pesada e pouco eficiente.

REST (Representational State Transfer)

Para simplificar as comunicações foi criado o REST, que com menos estandardização torna a comunicação entre processos mais simples.

Tem por base a utilização de ***endpoints*** **HTTP**, através de pedidos GET, POST, DELETE, PUT, através dos quais são trocadas mensagens em formato JSON.

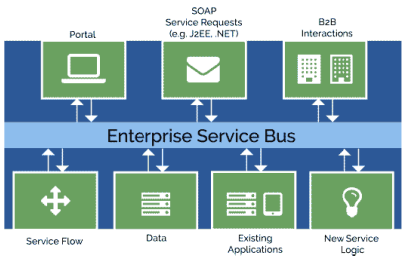
Um elemento fundamental nesta arquitetura são os **recursos** (JSON, documentos, …) sobre os quais são feitas as operações CRUD (*Create, Read, Update and Delete*).

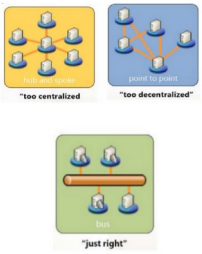
O *create* é feito pedido pedido POST, o *read* pelo GET, o *update* pelo PUT e o *delete* pelo DELETE.

Apresenta no entanto algumas desvantagens, nomeadamente a **dificuldade em representar objetos complexos**, a **necessidade de documentar** as interfaces e ainda a **necessidade de implementar sistemas adicionais** para controlar identidade, segurança, etc.

Origens

Esta arquietura era inicialmente utilizada para desenvolver aplicações monolíticas, que comunicavam entre si através de ***Enterprise Service Buses*** (*middlware* de mensagens).



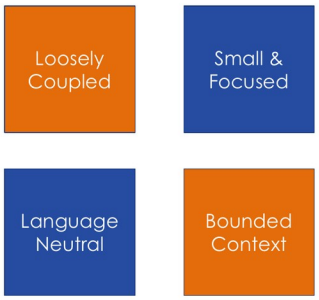


Este serviço de mensagens assumia as responsabilidades de segurança, gestão de exceções, etc.

Arquitetura de micro-serviços

Este padrão apresenta-se como uma alternativa a aplicações monolíticas (que recorrem à *layered architecture*) ou orientadas a serviços (SOA), procurando simplificar o seu desenvolvimento através da sua divisão em pequenos serviços.

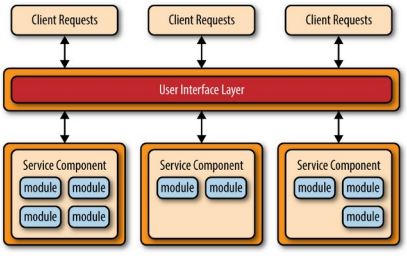
O principal conceito deste padrão é a noção de *deploy* de unidades separadas. Estas unidades são componentes que prestam um serviço e podem consistir num pequeno módulo ou numa grande porção da aplicação.

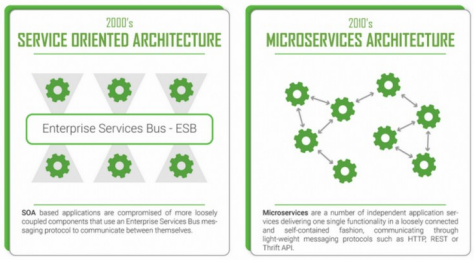


Em qualquer um dos cenários deve ser assegurada uma **arquitetura distribuída**, para a qual é fundamental o **desacoplamento** entre os módulos e a definição do seu protocolo de acesso (REST, SOAP, JMS, …).

O grande desafio é atingir o grau certo de granularidade.

Se for necessário orquestrar ou passar mensagens entre os componentes na camada da interface ou API, então o serviço está demasiado granulado. Caso esta não consiga ser reduzida (ou seja, por mais que tentemos a orquestração não deixa de ser necessária), então este não será o padrão adequado para desenvolver a nossa aplicação.





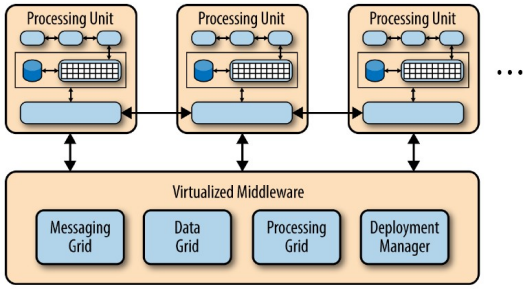
Os micro-serviços são funcionalmente mais pequenos que os componentes da SOA.

Arquiteturas space-based

A maioria das aplicações *web* gere os pedidos num fluxo **servidor > aplicação > base de dados**, fornecendo uma resposta em tempo útil para um fluxo normal de acesso.

No entanto, quando o fluxo começa a crescer começamos a assistir a um aumento do congestionamento do tráfego. Inicialmente no servidor, que pode ser expandido facilmente e sem grandes custos, mas que faz com que se comecem a sentir limitações na aplicação, esta mais difícil e cara para expandir e por fim na base de dados, que também requere um grande investimento para a sua expansão.

A ***space-based architecture***, também conhecido por ***cloud architecture*** procura então resolver os problemas de escalabilidade e concorrência, consistindo numa melhor alternativa à de tentar escalar os elementos do referido fluxo.



O nome deste padrão vem do conceito de ***tuple space***, um paradigma de memória partilhada distribuída que implementa através da remoção da base de dados central, criando uma rede de dados replicada em memória.

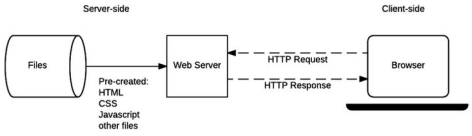
Cada **unidade de processamento** vai então ter uma cópia da base de dados, pelo que estas podem ser inicializadas e desligadas dinâmicamente de acordo com a variação do tráfego de acesso.

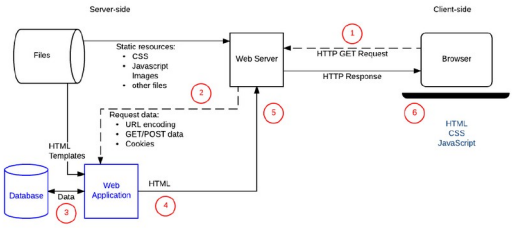
6. Web Frameworks

Slides teóricos e aula teórica assíncrona

Até há poucas décadas, os programas corriam localmente nos computadores. Hoje em dia assistimos ao inverso desta realidade, com as aplicações *web* a terem o papel central nas aplicações que utilizamos diariamente.

As páginas *web* evoluiram de versões as **estáticas**, que retornam o código originalmente armazenado no servidor, sem qualquer alterações e as **dinâmicas**, cujo conteúdo das páginas é gerado dinâmicamente, usualmente introduzindo informação de bases de dados em *templates* HTML.





O código que suporta esta contrução dinâmica de páginas é executado num servidor *web*, cuja programação é conhecida por *server-side* ou ***backend***.

Gerar páginas dinâmicas

A introdução do ***backend*** nos servidores *web* trouxe várias vantagens ao mundo das aplicações *web* para além do conteúdo **dinâmico**.

Performance

Evitam-se tarefas de computação duplicadas através da utilização da *cache*

Execução de tarefas pesadas em máquinas com mais capacidade

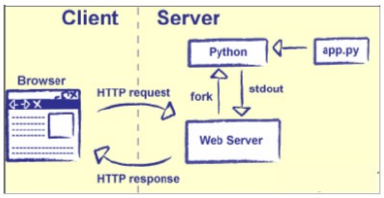
Tarefas de computação que fazem uso dos dados mais próximas dos dados

Segurança

Validações de segurança e autenticação

GCI (Common Gateway Interface)

Esta tecnologia introduziu as aplicações dinâmicas e tem um conceito baseado na especificação da aplicação a executar no servidor para gerar a página *web* através do URL. O servidor, ao receber um URL executa a aplicação e redireciona o *standart output* para o navegador que fez o pedido.



O programa faz *prints* de elementos HTML.

ASP (Active Server Page) e JSP (Java Server Packages)

Estas abordagens surgiram para simplificar o processo de geração das páginas dinâmicas proposto pelo GCI. Assim, em vez de gerarem código HTML dentro do código de uma linguagem de programação, encapsula código de uma linguagem de programação no código HTML.

O Django implementa esta abordagem através da *template tags*.

```
<a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
```

Frameworks

Nos anos 2000 começaram a surgir finalmente as *frameworks*, que quer sejam focadas no lado do servidor ou do cliente, fornecem um "esqueleto" para a criação de aplicações (código base) que pode ser costumizado pelo programador.

Por exemplo a *framework* Spring permite gerar o esqueleto para um projeto Maven e com alguns cliques no Spring Initializr ter uma aplicação a dar o *output* "Hello world!" sem ter de escrever qualquer linha de código.

Evitam assim a "reivenção da roda", evitando a escrita de código comum a vários projetos cada vez que se cria um novo.

Para além de fornecerem ferramentas e definirem regras para a sua utilização e estruturação, geralmente incluem várias livrarias. Controlam o fluxo da aplicação e invocam código da mesma.

Não devem ser confundidas com bibliotecas, que são utilizadas pelo código da aplicação para fornecer funcionalidade.

Ou seja, nós chamamos as livrarias, as *frameworks* chamam-nos a nós (código).

Esta inversão do controlo sobre o código torna as *frameworks* mais **díficeis de aprender**, **mais dependentes** (uma vez que o código segue as regras da *framework*).

No entanto, existem bastantes vantagens da sua utilização para além da velocidade e simplicidade de implementação, como o acesso a **soluções consolidadas** e amplamente utilizadas na industria, assim como a **manutenção** regular com atualizações de segurança.

Client-side frameworks

As mais utilizadas hoje em dia são o React e o AngularJS e permitem várias funcionalidades.

Manipulação do documento O DOM permite a manipulação do HTML e CSS.

Atualizar dados através de chamadas assíncronas ao servidor (p.e. AJAX).

Manipulações gráficas complexas como desenho no navegador.

Armazenamento do lado do cliente, através de "bases de dados" locais (p.e. IndexedAPI).

Server-side frameworks

Estas podem ser divididas em três grandes grupos.

Micro São focadas no encaminhamento de pedidos HTTP.

Semelhantes ao GCI

Full-stack Permitem a definição do lado do servidor de forma simplificada seja através do fornecimento do acesso aos dados, de controlo de acesso, de roteamento, templates…

O Spring facilita o acesso aos dados através dos JPARepository, que abstraem o programador da conexão à base de dados. Através do Thymeleaf podemos usar templates.

Component Livrarias especializadas e com um único propósito que podem ser utilizadas em conjunto para criar uma das *frameworks* anterior (micro ou full-stack).

O React começou por ser uma livraria *client-side.*

- Python
  - Django
- JavaScript
  - Express.js
- Ruby
  - Ruby on Rails
- Java
  - Spring
- PHP
  - Symfony
- C#
  - ASP.NET

Componentes

Todas as frameworks focadas no desenvolvimento de aplicações *web* apresentam três componentes nucleares.

Roteamento dos pedidos HTTP para o código responsável pelo seu processamento

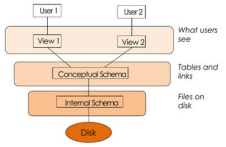
Mecanismos de Template que separam a lógica da apresentação

Acesso aos dados de forma abstrata (sem ter em conta o tipo de BD a ser utilizada)

Os **mecanismos de template** renderizam strings a partir de tokens escritos de acordo com uma estrutura pré-definida.

Por exemplo o Django faz esta renderização com base nos template tags.

O conteúdo mostrado a diferentes utilizador varia de acordo com a **view** gerada para cada um. No entanto, as várias views partem de um **esquema concetual** comum sobre o **esquema interno** que está armazenado em disco.



Há ainda outros componentes que podem fazer parte destas frameworks, nomeadamente a **segurança** contra os ataques mais comuns, **controlo de sessões**, **gestão de erros** ao nível da aplicação e **scafoolding**, que consiste em gerar de forma automática interfaces CRUD com base no modelo de dados em utilização.

Independência dos dados

Devem ser asseguradas as independências **lógica** e **física**, que permitem a independência entre a view e o modelo.

A **independência lógica** permite a alteração do esquema lógico sem ter de alterar o esquema externo.

Por exemplo adicionar um atributo a uma tabela da BD sem necessidade de alterar a view para que este seja visível.

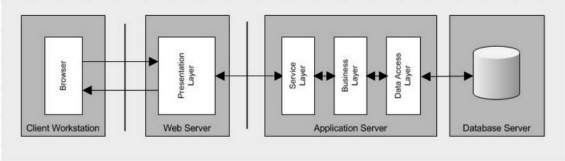
A independência **física** consiste na possibilidade de alteração o esquema físico sem alterar o esquema lógico.

Por exemplo ao alterar o SGBD utilizado não deverá de haver necessidade de alterar o código.

Arquiteturas

Arquitetura em camadas

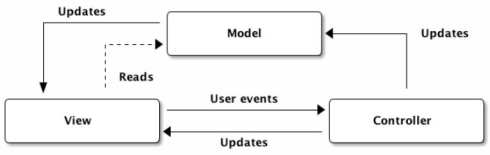
Geralmente apresentam três componentes, **apresentação**, **lógica do negócio** e **dados**, que funcionam de forma independente, podendo até ser executadas em máquinas distintas.



No entanto, apresenta alguns senãos na sua aplicação prática, a começar pelo *browser*, que pode estar num contexto *mobile* ou de *desktop*, devendo nestes dois casos ser apresentada uma apresentação distinta sobre os dados e formas de interação também distintas.

Model-View-Controller (MVC)

Uma alternativa é o modelo MVC, que trabalha com os componentes **funcionalidades nucleares** (model) e **apresentação** interligadas pela **lógica de controlo**.



Este modelo permite a geração de vistas diferentes para o mesmo modelo de dados.

O modelo poderão ser utilizadores, itens de uma loja de roupa, etc.
As vistas são as suas representações visuais, em HTML, CSS e Javascript.

7. Boas práticas no desenvolvimento de projetos

Slides teóricos e aula teórica assíncrona

Independentemente da metodologia de desenvolvimento utilizada, o planeamento do desenvolvimento de *software* engloba 4 fases.

Especificação

Design e Implementação

Validação Verificação que o sistema faz o que o cliente quer

Evolução Alterações no sistema (incrementais ou não) de acordo com as necessidades do cliente

Especificação

Nesta fase é definido o que o sistema vai fazer.

Para este efeito são geralmente utilizadas as **histórias de utilização**, um formalismo que permite ao programador perceber o cliente quer, sem este ter de entrar em detalhes técnicos, e assim o ajudam na definição dos **requisitos do sistema**.

Para facilitar esta planificação podem ser utilizadas aplicações que permitem a priorização, distribuição e acompanhamento do trabalho a ser desenvolvido.

O GitHub, o GitLab, o Jira e o Pivotal Tracker são exemplos de *softwares* que fornecem este tipo de serviços.

Design e implementação

Definição da organização do sistema e implementação (código).

Nesta etapa é bastante comum a utilização de **repositórios de código**, que permitem fazer controlo de versões.

É uma boa prática, dentro destes repositórios, utilizar o **git workflow**, um fluxo que consiste na divisão do repositório em vários *branches* para diferentes fins. Um principal, que representa o código em produção, outro para o desenvolvimento, do qual saem (*checkout*) vários para features.

Para evitar conflitos no *merge* das features com o *branch* de desenvolvimento, deve ser feito um *merge* do desenvolvimento na feature e resolver os conflitos antes de fazer *pull request* da feature no desenvolvimento.

Existem ainda os *branches* para hotfix, no qual são feitas correções a erros graves detetados, que depois são *merged* com o principal e o de desenvolvimento.

Outro bom princípio é a utilização de **pull/merge requests**, que consistem em pedidos de *merge* em *branches* protegidos, cuja gestão está tipicamente associada ao gestor de GIT (*devops master*).

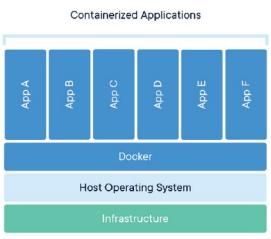
Devem ter uma descrição completa, com uma descrição, história associada e descrição dos testes efetuados.
Os labels ajudam a perceber quais os *pull requests* que estão prontos, os que precisam de revisão.

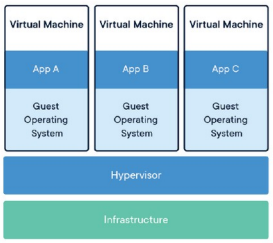
Faz sentido criar *pull requests* para *features* em desenvolvimento, porque é uma forma de documentar o *branch*. Não há outra forma de o documentar.

É também importante que os **nomes das branches** sigam uma convenção. Uma amplamente utilizada é CATEGORY/NAME.

CATEGORY	DESCRIPTION
bug	Bug fixing
imp	Improvement on already existing features
new	New features being added
wip	Works in progress - Big features that take long to implement and will probably hang there
junk	Throwaway branch created to experimentation
release	New release before merging with master

De forma a facilitar a **gestão de dependências** é bastante comum o **desenvolvimento baseado em containers**, que consiste na utilização de um ambiente de desenvolvimento virtual e semelhante por todos os membros da equipa. Facilita também o *deploy*.





Virtualization	Containerization
More secure and fully isolated	Less secure and isolated at the process level
Heavyweight, high resource usage	Lightweight, less resource usage
Hardware-level virtualization	Operating system virtualization
Each virtual machine runs in its own operating system	All containers share the host operating system
Startup time in minutes and slow provisioning	Startup time in milliseconds and quicker provisioning

Pode ser feito através de ambientes *docker* ou de virtualização.

O Docker facilita a utilização de vários containers na mesma máquina de uma forma muito mais leve. No entanto, a máquina virtual é muito mais reservada.

Containers Docker

É fundamental que as imagens de desenvolvimento e produção sejam as mesmas, variando apenas na configuração. Em produção devem ser utilizados volumes, porque os *containers* são mais propícios a "morrer".

Imagens são a base dos *containers*. Cada imagem pode ter vários *containers*, mas um *container* só pode ter uma imagem. Permitem herança entre si.

Para simplificar a integração de vários *containers* recorre-se ao **Docker Compose**, permitindo a orquestração destes com base numa certa ordem.

8. Spring Framework

Slides teóricos e aula teórica assíncrona

Uma *framework* define-se como algo que fornece uma forma estandardizada de fazer alguma coisa.

No contexto de contrução de um carro, a *framework* iria providenciar a carroçaria (esqueleto), sobre a qual o utilizador poderia trabalhar de forma a personalizar o carro ao fim que lhe pretende dar, como colocar determinadas jantes, motor, estofos…

Hoje em dia são utilizadas amplamente no desenvolvimento de *software*, potenciando a reutilização de código e ajudando os programadores a focar-se na lógica do negócio. Neste âmbito providenciam um ambiente de desenvolvimento que para além de um conjunto de livrarias, fornece um programa de base.

Um dos grandes **desafios** que advêm da sua utilização é o **(des)acoplamento** entre os componentes a serem desenvolvidos e a *framework* em si.

Estas podem ser de vários tipos.

Micro Ambientes do lado do servidor focadas na resposta a pedidos HTTP

Geralmente são utilizadas para implementar APIs. P.e. Flask, Spark, Express.js.

Full-stack Ambientes que incorporam um leque amplo de funcionalidades que criam condições para a inclusão de ferramentas mais poderosas que facilitam o desenvolvimento

Podem incluir roteamento, *templating*, acesso de dados e mapeamento.
O Spring fornece ferramentas para acesso de dados e mapeamento, abstraindo-nos do tipo de base de dados e das operações atóminas a realizar sobre ela.
Para além do Spring, temos o Django e o ASP.NET.

Component Geralmente permitem o desenvolvimento integral, desde o *backoffice* até à interface de utilizador.

Angular, React, Vue…

A *framework* **Spring** nasceu no início dos anos 2000 e desde então tem evoluído desde um *container* para injeção de dependências para um eco-sistema de *frameworks* que abranje um espetro bastante alargado do desenvolvimento de aplicações.

Coexiste com a **JEE** (*Java Enterprise Edition*), mais tarde **Jakarta EE**. Apesar de definirem especificações distintas, a Spring implementa algumas do EE.

Para cada interface o JEE define **JSR – Java Specification Request**, que estabelecem a interface para cada funcionalidades.

Algumas destas JSR são implementadas pela Spring.

Arquitetura Spring

A arquitetura Spring divide-se em vários componentes.

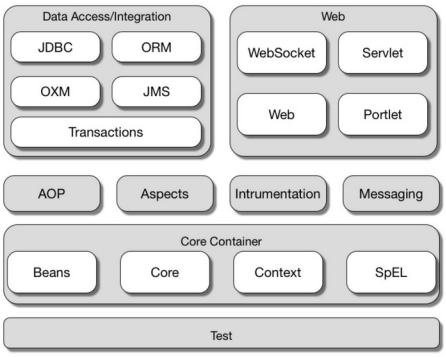
O **core container** tem quatro módulos, que são a base de qualquer aplicação.

Spring Core Fornece implementação para funcionalidades como IoC (Inversion of Control) e injeção de dependências

Spring Bean Implementa padrão de fábrica através da BeanFactory

Spring Context Com suporte nos dois anteriores, este módulo permite aceder a qualquer objeto definido e configurado

Spring Expression Languages (SpEL) Fornece uma linguagem que permite fazer *queries* e manipulação de um grafo de objetos, em tempo de execução.



Anotações

No processo de compilação de um programa é feita uma análise sintática para verificar que o código cumpre com uma determinada estrutura. No entanto, há situações em que queremos adicionar verificações adicionais, ou mesmo alterar o código de forma sistemática em determinados trechos durante o processo de compilação.

Para responder a estes cenários e de forma a evitar a complexidade da alteração do *Java Compiler*, são adicionadas anotações ao código, metadados que dão informação sobre o código ao compilador e que podem ser considerados por outro desenvolvido por nós para esse fim.

Facilitam a deteção de erros ou supressãom de avisos, por exemplo.

Alguns exemplos de anotações interpretadas pelo *Java Compiler* são o @Override, @Deprecated.

Caso num programa utilizemos um método de uma classe anotado como @Deprecated, quando compilarmos estes programa o compilador irá mostrar-nos o aviso de que estamos a utilizar um método obsoleto.

Assim, no processo de construção o código é primeiro analisado pelo nosso compilador, que para cada anotação despoleta um determinado comportamento (que pode ser uma ação, geração ou alteração de código, …), sendo o código resultante compilado pelo *Java Compiler*.

Inversão de controlo (IoC)

Quando trabalhamos com uma *framework*, como esta fornece a maioria das funcionalidades necessárias ao funcionamento de uma aplicação *web*, a *framework* passa a ser a responsável pela criação dos objetos.

Person.java

```
public class Person {
   private String id;
   private String name;
   // ...
}
```

beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
<!--  other stuffs ... -->
<beans ...>
   <bean id="person1" class="ua.ies.Person">
     <property name="id" value="12345"/>
     <property name="name" value="Leonor"/>
   </bean>
</beans>
```

Main.java

```
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    Person p1 = context.getBean("person1", Person.class);
    System.out.println(p1.getId() + ", " + p1.getName()));
}
```

O atributo name da propriedade no XML deve ser mapeado num getName e setName na classe do objeto a criar, com Name o valor do atributo.

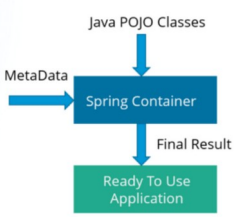
No exemplo acima para o atributos *id* e *name* devem haver os métodos *getId()* e *setId()* e *getName()* e *setName()*.

Injeção de dependências (DI)

Quando um objeto tem como atributo outro objeto, vai depender dele, sendo por isso necessário criá-lo antes do primeiro. A **injeção de dependências** é um padrão de desenho de *software* que implementa este conceito.

Assim, no caso de um objeto A depender de um B, o B é criado primeiro e injetado no A. Isto é também uma **inversão de controlo**, uma vez que o B tem como atributo A, mas não é sua responsabilidade a sua criação.

A injeção pode ser feita através de um contrutor ou de um *setter*.



Em **Spring**, as classes identificadas com a anotação @Component são geridas de acordo com estes princípios de IoC e DI pelo **Spring Container**, o ambiente que vai gerir todos os objetos e que com base em metadados (XML, anotações) e classes Java gera uma aplicação configurada e pronta a executar.

Este tem como principais tarefas a criação de objetos/beans, injeção de dependências entre eles, configurá-los e gerir todo o seu ciclo de vida.

Há vários tipos de Spring Containers, como por exemplo a BeanFactory ou o AplicationContext (feita a partir da interior). A última oferece várias interfaces para implementações distintas.

Beans

São definidas pela documentação Spring como os objetos base da aplicação, que são geridos pelo **Spring IoC Container** (desde a instanciação e construção até à gestão).

As classes destes objetos têm de cumprir com um conjunto de normas que garantem que estas não têm funcionalidades, mas apenas representação de dados, nomeadamente os atributos serem privados, todos terem gettes e setters, terem um construtor sem argumentos e os atributos só serem acessíveis ao construtor e getters e setters.

Em Spring podem ser definidos através de um ficheiro de configuração XML, pela anotação @Component (ou derivadas como @Service, @Repository, @Controller) ou escrevendo um método de fábrica anotado com @Bean.

O primeiro deixou de ser utilizado e só é mantido por questões de compatibilidade com *software* antigo.

A anotação @Component é utilizada quando temos acesso ao código fonte. Quando esta só tem um construtor, o Spring considera os seus parâmetros como a lista de dependências obrigatórias. Caso haja mais do que um, devemos anotar o que define as dependências com @Autowired.

A anotação @Autowired é utilizada para definir dependências de um bean em relação a outro sem ter de instanciar o segundo. Esta pode ser aplicada à definição do atributo, no construtor ou no *setter*.

```
@Component
class BeanWithDependency {
   private final MyBeanClass beanClass;
   BeanWithDependency(MyBeanClass beanClass) {
      this.beanClass = beanClass;
   }
}
```

```
@Configuration
class MyConfigurationClass {

   @Bean
   public static NotMyClass notMyClass() {
       return new NotMyClass();
   }
}
```

Quando não temos acesso ao código fonte da classe, definimos uma classe com um método de fábrica anotado com @Bean. Esta classe por sua vez é anotada com @Configuration, que marca a classe como sendo um *container* de definições @Bean. Podem ser definidos vários métodos de fábrica dentro da mesma classe de configuração.

Há ainda outras anotações importantes no que toca a beans.

@ComponentScan é utilizada para especificar os *packages* que contêm as classes anotadas. Por exemplo, ao fazer um bean para uma classe existente com a anotação @Configuration, esta anotação é útil para definir o *package* das classes para as quais vamos definir métodos de fábrica.

@Required Pode estar associada a construtores ou *setters* e define que determinado atributo é obrigatório para inicializar o bean.

@Scope Define a visibilidade do bean.

@Value …

Spring Expression Languages (SpEL)

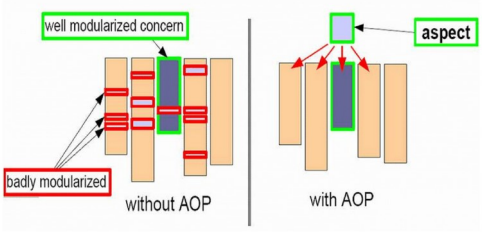
Como visto anteriormente, fornece uma linguagem que permite fazer *queries* e manipulação de um grafo de objetos, em tempo de execução.

A anotação **@Value** permite a injeção de valores nos atributos nos beans.

Aspect Oriented Progamming (AOP)

Fonte adicional: Read the docs

Este é um paradigma de programação complementar à Programação Orientada a Objetos e que tem como objetivo a separação das responsabilidades de forma a melhorar a modularidade.



Por exemplo na gestão de um fórum *online*, são várias as operações que necessitam de validar o utilizador de forma a que apenas sujeitos com determinadas permissões as podem executar. No entanto, criar código para fazer estas validações semelhantes em cada uma destas funcionalidades vai ser basicamente copiar código, que se necessitar de ser alterado no futuro vai ter de ser alterado em cada um destes componentes.

Muitas vezes até podemos estar a utilizar classes externar ao nosso domínio e não as podemos alterar para contemplar estas restrições.

A AOP permite o isolamento da segurança, ou de qualquer outro aspeto no seu próprio pacote de *software*, permitindo aos restantes que se foquem nas suas verdadeiras resposabilidades.

Pode ser visto como um **decorador dinâmico**.

Recorda que o padrão **decorator** tem como objetivo adicionar novas responsabilidades a um objeto de forma dinâmica, colocando-lo "dentro" de outros objetos que têm esse comportamento (encapsulamento).

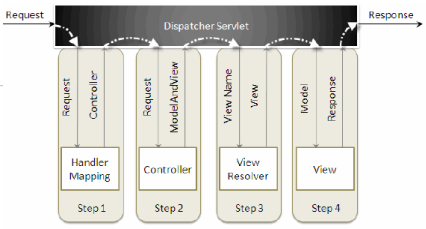
A sua definição é feita numa classe decorada com **@Aspect**, que aquando da execução de métodos (**join point**) vai realizar ações ou **advices** (@Before, @After, @AfterRunning, @AfterThrowing, …) quando se verificam determinas expressões nos join point pelos **pointcut**.

```
@Aspect
public class EmployeeAspect {
    @Before("execution(public String getName())")
    public void getNameAdvice(){
        System.out.println("Executing Advice on getName()");
    }
    @Before("execution(* ies.*.get*())")
    public void getAllAdvice(){
        System.out.println("Service method getter called");
    }
}
```

```
@Aspect
public class ProfilingAspect {
    @Around("methodsToBeProfiled()")   // action taken at the join point.
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }
    @Pointcut("execution(public * foo..*.*(..))")
    public void methodsToBeProfiled(){}
}
```

Spring Web

Este componente aplica a arquitetura Model-View-Controller (ver capítulo 6).



Quando recebe um pedido HTTP, o DispatcherServlet consulta o **HandlerMapping** para invocar o **Controlador** correto, que encaminha o pedido para o método adequado ao seu processamento (GET, POST, …).

De seguida, o DispatcherServlet pede ajuda ao **ViewResolver** para ir buscar a **View** correta, à qual vai finalmente passar o modelo de dados que é depois renderizada no navegador.

```
@Controller
@RequestMapping("/funcionarios")  // a kind of path..
public class FuncionariosController {
    @Autowired
    private FuncionarioService funcionarioService;
    @Autowired
    private Funcionarios funcionarios;

    @ResponseBody
    @GetMapping("/todos")  // curl -i http://localhost:8080/iesapp/funcionarios/todos
    public List<Funcionario> todos() {
          return funcionarios.findAll();
    }
```

Acesso a dados

Há vários módulos que permitem criar a abstração entre os dados e o seu armazenamento.

O **JBC** permite trabalhar diretamente com SQL.

Ao utilizarmos o **ORM** temos uma maior abstração, uma vez que trabalhamos sobre objetos Java, sendo responsabilidade do módulo mapear as alterações na base de dados. Para trabalhar neste paradigma sobre XML utiliza-se **OXM**.

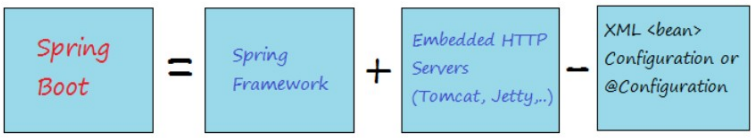
O **JMS** facilita a interação com serviços de mensagens.

Por fim, o módulo **Transaction** facilita a implementação dos conceitos de transações ao nível da organização.

9. Spring Boot

Slides teóricos e aula teórica assíncrona

O **Spring Boot** é uma extensão da framework Spring, cujo objetivo é eliminar a necessidade de definir as configurações padrão para iniciar uma aplicação Spring e fornecer uma estrutura de projeto padrão, para um desenvolvimento mais ágil e eficiente. Destaca-se ainda o servidor embutido que facilita o processo de deploy.



Starter Evita a necessidade de adicionar jars de livrarias que queremos utilizar ao projeto

São conjuntos de dependências que costumam ser utilizadas em conjunto, que podem ser utilizadas diretamente sem a necessidade das descrever. Por exemplo, para trabalhar com serviços SOAP podemos partir do spring-boot-starter-web-services, ou no caso de REST usar o spring-boot-starter-data-rest.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>
```

As versões são herdadas do spring-boot-starter-parent, de forma a garantir compatibilidade entre todas as dependências do projeto, cujos starters deixam de ver a sua versão descrita (é definida automaticamente).

AutoConfigurator Evita a configuração manual com o XML

CLI Permite correr e testar aplicações Spring Boot na linha de comandos

Actuator Permite a análise de métricas para *end points*

Os seus projetos são inicilizados em https://start.spring.io/, onde é possível selecionar várias dependências e obter um projeto pré-configurado e com a estrutura padrão.

A aplicação principal consiste num esqueleto padrão, onde a classe principal é passada à SpringApplication para esta a executar. Verifica-se a **inversão de controlo**.

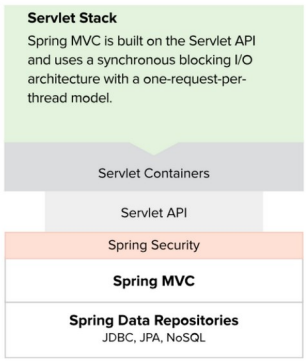
```
@SpringBootApplication
public class PayrollApplication {

    public static void main(String... args) {
        ApplicationContext ctx =
          SpringApplication.run(PayrollApplication.class, args);
    }
}
```

Enable component-scanning and auto-configuration

Bootstrap the application

@SpringBootApplication agrega três anotações:
@Configuration, que define a classe como sendo de configuração;
@ComponentScan, ativa o auto-scanning, que consiste na busca automática por controladore e outros componentes;
@EnableAutoConfiguration, ativa a auto-configuração, evitando a escrita de páginas XML para este fim.

Arquitetura Spring Web MVC (Model-View-Controller)

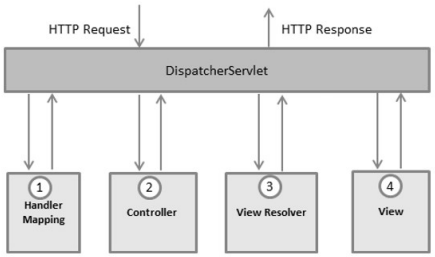


Este componente é responsável por identificar quais as classes que vão trabalhar com os dados e quais vão atendender pedidos do navegador, funcionando como a ponte que os liga.

Permite a criação de entidades @Controller, responsáveis por capturar os pedidos HTTP e encaminhá-los para as funções responsáveis por cada um.

A estrutura de um projeto MVC caracteriza-se pela divisão dos vários componentes em pastas.

/controller Classes de controlo, que capturam e tratam eventos da aplicação
/dao Data Access Object, permite lidar com o armazenamento das entidades (persistência)
/model Camada lógica. Modela dados com que aplicação vai trabalhar.
/service



O **modelo de dados** (/model) é definido através de uma típica classe Java, com atributos privados com getters e setters e um construtor vazio por defeito. Podem-lhe ser associadas anotações como @Entity, @Table(name=X) e aos seus atributos @Id ou @GeneratedValue.

```
@Entity
@Table(name = "employees")
public class Employee {

    private @Id @GeneratedValue long id;
```

Os **repositórios** (/dao) definem interfaces que fazem a ligação entre as entidades do modelo de dados e a base de dados. A sua definição é bastante simples e consiste na extensão de um **JPARepository**, a partir da qual são herdados os métodos de manipulação de dados CRUD. Podem no entanto ser definidos métodos adicionais para *queries* mais complexos.

```
@Repository
public interface EmployeeRepository
     extends JpaRepository<Employee, Long> {
        public List<Employee> findByEmail(String email);
  // ... other methods
}
```

Os métodos adicionais têm de seguir as convenções impostas pelo Spring Boot, uma vez que continuam a ser configurados automaticamente.

No exemplo acima, apesar de não ser um método fornecido por defeito, o findByEmail(String email) tem de ser declarado desta forma, uma vez que o Spring Boot vai procurar na classe Employee pelo atributo com o nome que segue 'findBy' e se esse nome não existir, dará erro.

Com base nesta terminologia 'findBy' podem ser criados vários tipos de métodos. São suportados os métodos 'findDistinctBy', 'findTop3By', 'findFirstBy'. Há ainda alguns sufixos, como 'IsNull', 'Like', 'Desc', 'IgnoreCase', 'After', 'Before', 'In', 'AndAttr2'…

O **controlador**, para além de anotado como tal através da anotação @RestController, é também mapeado para um determinado caminho na aplicação com @RequestMapping("/api").

```
@RestController
@RequestMapping("/api/v1")
public class EmployeeController {
    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping("/employees")
    public List<Employee> getAllEmployees(
            @RequestParam(required = false) String email,
            @RequestParam(required = false) String lastname) {
```

```
@PostMapping("/employees")
public Employee createEmployee(@Valid @RequestBody Employee employee) {
     return employeeRepository.save(employee);
}
```

Cada um dos seus métodos é também mapeado para um caminho relativo ao anterior, com @GetMapping("/abc"). Os seus parâmetros são anotados com @RequestParam(required=X) no caso de serem enviados através do URL ou @RequestBody no caso de serem enviados no corpo do pedido HTTP.

A anotação dos métodos varia dependendo do pedido HTTP a que respondem, podendo ser por exemplo PostMapping, PutMapping…

Definidos o modelo dos dados, a interface dos seus repositórios e o controlador que os manipula, falta apenas definir a **ligação da aplicação à base de dados**. Para este efeito, são adicionadas as dependências necessárias ao tipo de base de dados a utilizar e definidas as seus configurações no ficheiro application.properties.

In application.properties file:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=user
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

A definição dos dados de conexão à base de dados é feita no ficherio application.properties de forma a desacoplá-la do código. Assim, quando houver necessidade de alterar o *end point* da base de dados, basta editar este ficheiro, sem necessidade de fazer qualquer alteração e recompilar o código.

Por fim, a aplicação pode ser **executada** facilmente, dado o facto de incorporar um servidor HTTP embutido.

```
$ ./mvnw clean spring-boot:run
```

Spring WebFlux Framework

Em alternativa ao modelo Spring MVC, esta framework oferece um modelo assíncrono, em que o sistema não bloqueia à espera de uma resposta. Pode ser definida através de anotações, de forma análoga ao MVC, ou *hard coded* de forma funcional.

Permite que uma página seja refrescada continuamente sem ter uma resposta da base de dados para um determinado pedido que foi feito.

```
@RestController
@RequestMapping("/users")
public class MyRestController {

    @GetMapping("/{user}")
    public Mono<User> getUser(@PathVariable Long user) {
        // ...
    }

    @GetMapping("/{user}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }
```

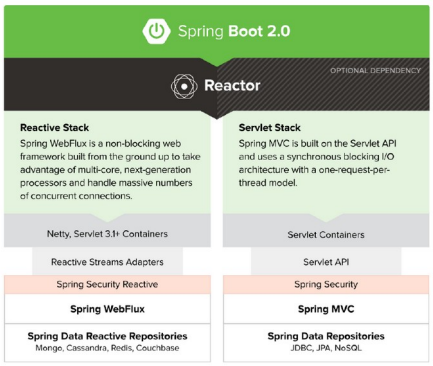
Mono<T> emits 0..1 elements

Flux<T> emits 0..n elements

O fluxo é como um stream, ao qual podemos pedir elementos continuamente. Opoem-se ao modelo em que obtemos uma resposta integral do servidor para consultas que retornam grandes volumes de dados.

MVC vs. WebFlux

O MVC é mais utilizado com bases de dados relacionais, enquanto que o WebFlux com NoSQL.



Spring Data

Este é o módulo que permite a gestão dos dados de uma forma abstrata, permitindo a utilização da grande maioria dos tipos de bases de dados, relacionais e não relacionais.

A adaptação entre os modelos de persistência (tabelas) e os modelos de representação em memória (classes) começaram por ser feitos para bases de dados relacionais pelo **Spring ORM (Object-Relational Mapping)**, que permitia relações entre tabelas e classes (one/many-to-many). Mais tarde, foi desenvolvido um mais avançado, o **Spring DAO (Data Access Object)**, um modelo mais genérico que se aplica a todos os tipos de bases de dados e utiliza tecnologias como JDBC, Hibernate ou JDO.

Com base nestes dois conceitos foram criados vários projetos, para diferentes tipos de utilização.

JPA (Java Persistence API)

Define uma norma para modelos de mapeamento entre Java e bases de dados relacionais, através do modelo **ORM**. Sendo uma especificação, é utilizado em várias implementações.

Hibernate

Esta é uma implementação do **JPA**, que fornece ferramenta **ORM** que fornece uma *framework* para o mapeamento do objetos em tabelas de BD relacionais e vice-versa.

Spring Data JPA

Este projeto não é uma atualização, mas um **utilitário**[1] **sobre o JPA**, adicionando-lhe uma nova camada, que permite a implementação de repositórios sem código e a criação de *queries* com base em nomes de métodos.

É novamente uma especificação que é implementada pelo **Hibernate**, associado ao projeto através da dependência *spring-boot-starter-jpa*.

1 Programa informático que executa tarefas auxiliares.

O Spring Boot pode fazer a configuração de bases de dados mais simples, como a H2, uma BD *in-memory*.

Para mapear uma classe na base de dados, basta anotá-la com @Entity e eventualmente @Table para definir qual a tabela onde os seus objetos serão armazenados. Cada um dos seus atributos pode ser mapeado com @Id e @GeneratedValue, para especificar uma chave primária com uma determinada estratégia de geração automática, e @Column para mapear atributos para colunas especificas da tabela.

```
@Entity public class Company {
    @Id @GeneratedValue
    private Integer id;
    private String name;
    private String address;
    @Embedded
    private ContactPerson contactPerson;
    // ...
}
```

A ligação entre classes é feita através da anotação @Embedded. Adicionalmente o atributo pode ser anotado com @AttributeOverride(s) para fazer o mapeamento entre os atributos da sua classe e as colunas para as quais estes vão ser mapeados.

As relações entre classes são anotadas com @OneToOne ou @OneToMany para referenciar entidades, @OneToMany ou @ManyToMany para referenciar conjuntos de entidades e @MappedSuperClass ou @Inheritance para herança entre entidades.

Para fazer queries personalizados, pode ser utilizada a anotação @Query, que permite a escrita de código SQL ou JPQL (JPA Query Language), com funcionalidades de ordenação e até paginação.

– SQL native – *over JDBC*

```
@Query( value = "SELECT * FROM USERS u WHERE u.status = 1",
        nativeQuery = true)
Collection<User> findAllActiveUsersNative();
```

– JPQL (*JPA Query Language*) – *over Hibernate*

```
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();

@Query(value = "SELECT u FROM User u")
List<User> findAllUsers(Sort sort);

@Query(value = "SELECT u FROM User u ORDER BY id")
Page<User> findAllUsersWithPagination(Pageable pageable);
```

Sorting

Pagination

JPA com MongoDB

Para além de suportar bases de dados relacionais, tem muitos *drivers* para outros tipos de BD, como o MongoDB. Estes por vezes apresentam ligeiras alterações no modelo, mas de um modo geral, este é bastante semelhante.

```
@Document(collections = "school")
public class Person {
  @Id
  private ObjectId id;
  private Integer ssn;
  @Indexed
  private String name;
}
```

```
@Repository
public interface PersonRepository
     extends MongoRepository<Person, String> {
        Person findByName(String name);
}
```

No exemplo acima vemos uma ligeira adaptação, com a anotação @Document em vez de @Table e a criação de um MongoRepository em vez de um JPARepository. No entanto, de resto a sintaxe é bastante semelhante e no fim vamos ter a mesma interface no repositório com os *queries* CRUD disponíveis sem necessidade de os definir.

10. Microservices

Slides teóricos e aula teórica assíncrona

Nos últimos anos, o paradigma de desenvolvimento de aplicações alterou-se de um modelo focado em armazenamento local (seja no *desktop* ou em servidores), para um baseado na nuvem, sendo hoje estes dois conceitos altamente interligados.

Esta alteração deve-se a fatores como a **alta disponibilidade** da nuvem, as **técnicas avançadas de virtualização** que estas permitem e a **facilidade de deploy**, que é quase instantânea uma vez que após a compra os serviços ficam disponíveis imediamente, em oposição à compra e configuração de servidores locais. Destaca-se ainda a **dinamicidade** destes ambientes, uma vez que é possível alocar recursos sempre que necessário.

Micro-serviços definem-se então como aplicações nativas da nuvem, compostas por peças pequenas, independentes, self-contained, substituíveis e poliglotas.

Pequenas, pois devem ser focados num único propósito. "Fazer uma coisa e fazê-la bem."

Independência e autonomia. Deve ser possível parar um componente para manutenção por exemplo sem "partir" os restantes, que podem até dar *timeouts* nas consultas caso estejamos a trabalhar com uma BD por exemplo. No entanto, quando este voltar a ser inicializado, as restantes devem voltar ao seu funcionamento normal sem necessidade de intervenção.

Substituíveis, porque deve ser possível alterar o funcionamento interno de cada componente, ou mesmo substituílo por um novo, desde que este mantenha as suas funcionalidades, ou seja, a mesma interface de acesso (API).

Poliglotas, porque dada a independência dos micro-serviços de uma aplicação, cada um destes componentes pode ser desenvolvido numa linguagem de programação diferente, sem prejuízo para as restantes, desde que todas comuniquem de acordo com um protocolo de comunicação agnóstico da linguagem-

---

Rever arquitetura de micro-serviços no capítulo 5

A sua aplicação deve assumir processos organizacionais distintos dos que se aplicam ao desenvolvimento monolítico e o seu desenvolvimento deve ser altamente coordenado entre as várias equipas.

Boas práticas no desenvolvimento

No desenvolvimento deste tipo de serviços, há um conjunto de boas práticas que deve ser seguido. São 12, mas destacam-se as principais. (3, 8 e 10 mais importantes)

1. Deve haver uma associação de um para um entre cada **serviço** e o seu repositório de **código**;
2. Os serviços devem declarar todas as suas **dependências** (não deve ser assumida a presença de ferramentas ou bibliotecas do sistema);
3. As **configurações** que variem entre ambientes de execução devem ser armazenadas (por exemplo variáveis de ambiente);

> Em Spring Boot, por exemplo, as dependências são declaradas no ficheiro pom.xml e as configurações no application.properties.

4. Todos os seviços de **armazenamento** devem ser desacoplados do serviço, sendo possível que este funcione sem o primeiro, ou que seja facilmente substituído por outro;
5. O processo de **desenvolvimento** deve **distinguir as fases** de construção, entrega e execução;
6. As aplicações devem ser executadas como um ou mais **processos independentes**;

> Por exemplo uma aplicação que seja composta por vários microserviços, deve ver cada serviços *deployed* num processo separado e independente, apesar de trabalharem em conjunto.

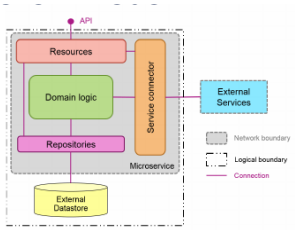
7. Serviços auto-contidos devem ser disponibilizados aos restantes através de uma **porta**;
8. A **concorrência** é executada através da escalada horizontal dos serviços (multiplicação);
9. Os processos devem ser **descartáveis**, ou seja, inicializar rapidamente e parar de forma responsável;

> Se tivermos um sistema de escrita de ficheiros, este deve ser inicializado rapidamente para ficar disponível assim que necessário, mas deve também garantir que quando é terminado guarda o *buffer* de escrita que estava foi introduzido pelo utilizador.

10. Deve haver uma **paridade**[2] **entre desenvolvimento e produção**, limitando ao máximo as diferenças entre os dois.

> Se acumularmos alterações no Dev, as operações de *merge* no Main vão ser muito mais complexas.

Estrutura interna



Um micro-serviço é como que uma pequena aplicação, uma vez que tal como estas vai trabalhar num domínio do problema, expor recursos através de uma determinada interface de acesso (API), aceder a serviços externos e aceder a bases de dados externas.

> Geralmente cada micro-serviço tem a si associada uma base de dados de uso exclusivo. Não é comum haver a partilha destes recursos por vários serviços.

2 Qualidade do igual ou semelhante.

Criar micro-serviços em Java

Plataformas Java

Até agora foi abordado o Spring Boot, um conjunto de convenções que simplificam o desenvolvimento de aplicações Spring. Este é bastante utilizado para a criação de aplicações *standalone*, que são disponibilizadas num determinado porto e funcionam de forma isolada.

Para responder à emergência dos sistemas distribuídos foi criado o **Spring Cloud**, um projeto que reduz a complexidade associada aos SD, permite a descoberta de serviços, oferece mecanismos de redundância, *load balancing*, problemas de performance e complexidade de *deploy*.

Para facilitar a gestão de configurações, mais vez recorre-se a ficheiros de configuração como o application.properties. Neste ficheiro podem ser definidas configurações como endereços de bases de dados e serviços externos com dados de acesso, mas também texto em várias línguas de forma a facilitar o suporte de diferentes idiomas pela nossa aplicação.

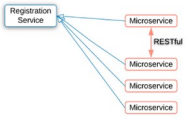
Para utilizar uma configuração numa classe da aplicação, basta anotá-la com @EnableAutoConfiguration e definir a variável que vai receber o seu valor anotada com @Value("${X}"), sendo X o nome da propriedade definida no ficheiro de configurações.

```
@Configuration @RestController
@EnableAutoConfiguration
public class Application {

  @Value("${config.name}")
  private String name;

  @RequestMapping("/")
  public String home() {  return "Hello " + name;   }

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Para facilitar a escalabilidade destaca-se ainda a o servidor de nomes/descoberta **Eureka**, que permite o registo dos vários serviços da aplicação, que deixam assim de ter necessidade de ter um endereço fixo. Os clientes que lhes queiram aceder consultam primmeiro o Eureka para resolver o seu endereço.

A aplicação de resolução de endereço é anotada com @EnableEurekaServer, o que vai disponibilizar este serviço no porto 8761. Os clientes que devam ser registados no Eureka, devem ver a sua classe principal anotada com @EnableDiscoveryClient.

```
spring-cloud-getting-started git:(master) $ ls
  hello-eureka-server
  hello-service
  hello-web-client-service
```




```
@SpringBootApplication
@EnableEurekaServer
public class HelloEurekaServerMain {
    public static void main(String[] args) {
        SpringApplication.run(HelloEurekaServerMain.class, args);
    }
}
```

```
@SpringBootApplication
@EnableDiscoveryClient
public class HelloServiceMain{

   public static void main(String[] args) {
      SpringApplication.run(HelloServiceMain.class, args);
   }
}
```

Controlo de versões e dependências

Para fazer uma boa gestão das dependências é fundamental declarar explicitamente as suas versões, de forma a garantir que a aplicação testada em ambiente de desenvolvimento vai ter o mesmo comportamento em produção.

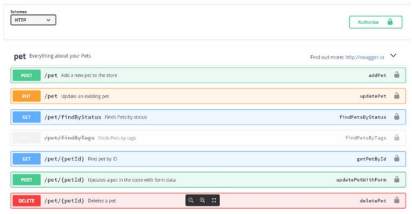
Identificação de serviços

Quando criamos *endpoints* de acesso aos serviços, é importante documentá-los. Para este efeito, existe uma norma definida pela Open API Iniciative para *web services*.

Esta norma que defina a estrutura de um ficheiro .json ou .yaml.

Pode ser feita uma conversão destes ficheiros para uma versão HTML interativa através do Swagger UI.

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
```



Não há qualquer restrição quanto à funcionalidade implementada em cada **método HTTP**. No entanto, é importante que a convenção seja seguida, de forma a facilitar a sua utilização.

Por exemplo podemos criar entidades através de um pedido GET. No entanto, a convenção é fazê-lo com POST e é assim que deve ser feito!

POST Criação de recursos. Operação não idempotente[3].

GET Consulta de informação. Operações idempotentes e nulipotentes (não alterar a base de dados).

PUT Atualização de recursos. Operação idempotente.

PATCH Atualização parcial de recursos. Pode ou não ser idempotente.

DELETE Eliminação de recursos. Operação não idempotente.

O **valor de retorno** dos pedidos deve também ser revelante e útil. Para além do código HTTP, deve também ser retornada a informação manipulada, de forma a evitar uma nova chamada à API e a tornar a comunicação mais eficiente.

Por exemplo em caso de sucesso deve ser retornado **código 200**, ou **400** em caso de um pedido mal formado (BAD REQUEST).

A forma como os **URIs[4] dos recursos** são definidos tem também definido um padrão, que define que este deve ser composto por nomes plurais e nunca por verbos. Para manipular o mesmo recurso, deve ser utilizado o mesmo URI, com /ID quando se pretende consutar ou manipular um recurso concreto. As relações são modeladas a partir do URI do recurso.

Por exemplo, para acedermos ou manipularmos as credenciais de um determinado utilizador podemos definir o URI /user/16/credenciais.

A evolução da API é também uma questão bastante importante, uma vez que as alterações à sua interface não devem implicar a alteração dos serviços que a utilizam, uma vez que iria violar o princípio da independência.

Para permitir a **evolução independente da API de um micro-serviço**, deve então ser adotada uma de três soluções: colocar a versão no URI, definir a versão no cabeçalho do pedido HTTP ou até mesmo no cabeçalho do HTTP Accept *header* e permitir a negociação.

A versão deve ser aplicada à aplicação como um todo. Por exemplo /api/v1/user e não ~~/api/user/v1~~.

3 Propriedade de uma operação que pode ser aplicada mais do que uma vez sem que o resultado se altere.

4 Universal Resource Identifier

Serviços de localização

No tópico anterior foi abordado o Eureka, um destes serviços.

Como visto anteriormente, em micro-serviços a escalada é feita horizontalmente, ou seja, através da multiplicação dos recursos. No entanto, esta traz a si associado os problemas da localização dos serviços e de distribuição do trabalho entre as várias instância. Para lhes dar resposta foram criados os **serviços de localização**.

Os micro-serviços contactam com os serviços de localização para a realização de quatro operações distintas:

Na perspetiva de disponibilizar serviços
Registo e **"des-registo"** Mostrar-se disponível/indisponível para receber pedidos.
Heartbeats Informar o serviço de que continua disponível para receber pedidos.

Na perspetiva de utilizar serviços
Descoberta de serviços Quando um serviço quiser comunicar com outro, pede ao serviço de localização pela listagem das instâncias disponíveis.

O **registo** de um micro-serviço num serviços de localização pode ser feito por ele próprio ou por terceiros.

No primeiro caso, a lógica do registo vai ser encapsulada na lógica do negócio.

A utilização de terceiros implica a inspeção constante dos serviços para determinar o seu estado atual e atualizá-lo no serviço de localização. Tem como vantagem a separação das lógicas do negócio e do registo. No entanto, implica o *deploy* de um componente extra.

Tolerância a falhas

Para garantir que os serviços disponibilizados nas nosss aplicações são tolerantes a falhas, é importante que incorporem algumas funcionalidades.

Timeout Tempo máximo para a resposta a cada pedido

Circuit breakers Contagem de *timeouts* que quando atinge um determinado patamar assume que vai dar sempre *timeout* e deixa de fazer pedidos para esse *endpoint*

Bulkheads Gestão de falhas. Como que o bloco *catch*. Garantem que as exceções não comprometem o normal funcionamento do serviço.

Consumir dados de API Os dados recebidos de qualquer API devem ser validados, mas ignorar dados que não precisamos.

Informação a mais deve ser ignorada, uma vez que não terá impacto no nosso serviço. No entanto, se faltar informação, não podemos considerar a resposta recebida ao nosso pedido.

Fornecer dados através da API Ignorar atributos não necessários no pedido. Responder apenas com informação necessária.

Do ponto de vista do consumo devemos ser tolerantes, mas do ponto de vista do fornecimento devemos ser rigorosos!

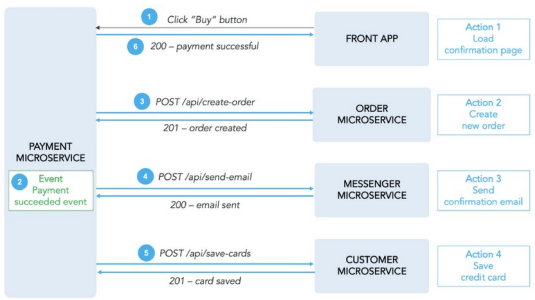
11. Sistemas baseados em eventos

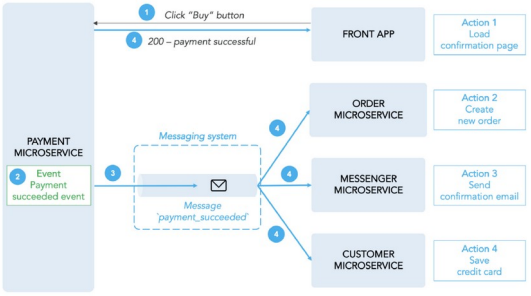
Slides teóricos e aula teórica assíncrona

Os micro serviços, abordados no capítulo anterior, têm sido o centro do desenvolvimento de software, assente em tecnologias cloud, que são características pelo seu modelo de execução serverless.

Serverless, ou computação sem servidor, é um modelo de execução de computação em núvem, característico pela alocação de recursos de forma dinâmica em nome dos seus clientes. +

No entanto, uma aplicação é composta por vários serviços e para que esta funcione eles precisam de trabalhar em conjunto. Para os conectar podemos optar por uma arquitetura **baseada em pedidos** ou por uma **baseada em eventos**.





O problema da arquitetura baseada em pedidos é que é mais demorada, uma vez que após a confirmação do pagamento vai pedir a criação do registo, que após concluída vai pedir para enviar e-mail e só no fim vai guardar os dados do cartão, depois da qual responde que o pagamento foi bem sucedido. A espera que cada um destes passos intermédios seja concluído vai atrasar imenso a resposta à aplicação que fez o pedido de pagamento.

Na arquitetura baseada em eventos, após a confirmação do pagamento o evento é gerado no sistema de mensagens e o serviço de pagamento fica imediamente livre para responder ao cliente que o pagamento foi bem sucedido, passando as restantes responsabilidades para o sistema de mensagens, que aciona as 3 ações que anteriormente eram geradas pelo sistema de pagamento.

A **arquitetura baseada em eventos** é a mais comum e pode ser implementada através de filas de mensagens como RabitMQ ou enterprise seervice buses (ESBs) como o WSO2. Mais recentemente foi introduzido o conceito de **streaming de mensagens** com o Kakfa.

As filas de mensagens e os streamings são ambos intermediários de mensagens entre produtores e consumidores, mas com um modo de funcionamento diferente. As filas de mensagens removem as mensagens da fila quando as entregam ao consumidor, eliminando o seu registo. Por outro lado, os streamings persistem as mensagens entregues ao consumidor, sendo responsabilidade do último manter um ponteiro para a mensagem a ler a seguir, libertando o código da aplicação de se preocupar com a potencial perda de transações, uma vez que pode pedir por mensagens lidas anteriormente.

As filas são mais utilizadas em arquiteturas orientadas a serviços (SOA), enquanto que os streamings nos micro serviços.

Esta arquitetura segue um **padrão** de modelação dos dados como streams de eventos, em vez de operações sobre registos estáticos.

O armazenamento está focado no armazenamento dos eventos mais do que nos objetos. Para saber o estado de uma entidade num determinado momento faz-se "play" dos eventos que ocorreram sobre essa até àquele momento.

Eventos

Define-se por **evento** algo que acontece dentro de um sistema computacional durante um determinado processo. Mimetizam a forma dos humanos organizarem o dia-a-dia e diminuem o número de relações de um-para-um nos sistemas distribuídos, aumentando o potencial dos micro serviços.

Um exemplo de uma "arquitetura" baseada em eventos no mundo real é um restaurante onde os empregados de mesa coloquem os papéis com os pedidos dos clientes por ordem num quadro, que é consultado pelo pessoal da cozinha à medida que estes têm disponibilidade para cozinhar mais uma refeição e pelos empregados do balcão para preparar as bebidas e os aperitivos. O quadro onde os pedidos são colocados é como que um sistema de stream de mensagens, que pode ser consultado tanto pelo pessoal da cozinha como do balcão, sendo responsabilidade de cada um saber a que pedidos já responderam. Eventualmente o papel de cada pedido será removido pelo empregado quando este tiver pronto e sido entregue ao cliente.

Neste caso é também visível a redução do número de relaçõs de um-para-um, uma vez que em vez de ter de se dirigir à cozinha e ao pessoal do balcão para fazer o mesmo pedido, o empregado limita-se a colocá-lo no quadro.

Não devem ser confundidos com **queries**, que são pedidos de consulta, nem **comandos**, que são pedidos de ação que alteram o estado do sistema.

	Behavior/state change	Includes a response
Command	Requested to happen	Maybe
Event	Just happened	Never
Query	None	Always

Os eventos podem ser classificados em…

Event notification Notificam outros sistemas de alterações no seu domínio;

Event-carried state transfer Evolução das anteriores, mas evento inclui dados para trabalhos adicionais;

Requerem mais recursos para ser transferidos, pelo que deve haver um compromisso entre eficiência e a utilidade dos dados incluídos.

Event-sourcing Representa todas as mudanças de estado como um evento, registados por ordem cronológica;

Permite a reconstrução da história do sistema com base nos eventos.

Vantagens sobre abordagens REST

As abordagens REST seguem uma **arquitetura baseada em pedidos**, onde o cliente espera que todas as ações despoletadas pelo seu pedido sejam executadas e apenas depois recebe a resposta. Em contraste temos a **arquitetura baseada em eventos**, que apresenta algumas vantagens sobre a anterior.

Assíncrono Permite que os recursos sejam otimizados e passem à tarefa seguinte quando o trabalho que lhes compete está completo. Previne perda de pedidos ou bloqueio dos produtores por pressão dos consumidores.

Baixo acoplamento Os serviços operam de forma independente, sem conhecimento dos restantes.

Facilmente escalável Devido ao desacoplamento, é fácil identificar os serviços mais congestionados e aumentar os seus recursos de forma independente.

Suporte à recuperação O histórico dos eventos permite a recuperação e análise do que foi feito no passado.

No entanto, há também algumas desvatangens, como o perigo de **over-engineer**, ao desenvolver soluções demasiado dispersas e complexas que seriam mais simples juntas, e a **gestão complexa de dados e transações**, uma vez que não há suporte para ACID[5].

Componentes principais

Sources e Sinks Produtores e consumidores dos eventos.

Sistema de mensagens Componente central. Gere a comunicação entre as várias entidades.

Aplicações do servidor Fazem uso do sistema de mensagens para processamento (consumo) ou geração de eventos

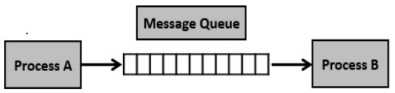
Sistemas de armazenamento e análise

Log de eventos e monitorização

5 Atomicidade, Consistência, Isolamento e Durabilidade (Persistência)

Sistemas de mensagens

A sua implementação mais simples segue o modelo **point to point**, em que as mensagens são enviadas para uma fila que estabelece uma relação de um-para-um entre o produtor e o consumidor, que é gerida pelos dois. Cada mensagem é consumida uma única vez.



A alternativa é o modelo **publish-subscribe**, onde vários produtores colocam os eventos num de vários tópicos, que são entregues a todos os consumidores que o tenham subscrito.

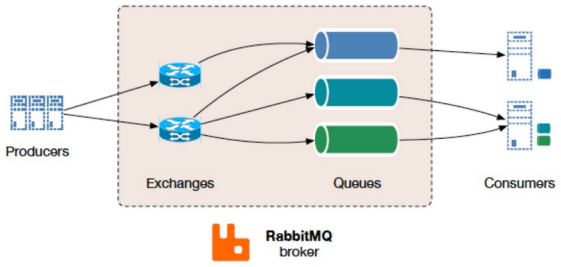
O segundo modelo permite um maior desacoplamento entre os processos e a comunicação dos eventos, cuja gestão deixa de ser sua responsabilidade e passa a ser tipicamente de uma nova aplicação, um **message broker**. Para além do processamento e encaminhamento das mensagens, destaca-se ainda a possibilidade de conversão entre protocolos de transporte.

Deve ser utilizado sempre que virmos que o código está a ficar demasiado focado na gestão de mensagens, uma vez que permite uma gestão opaca (só nos interessa a interface, não como funciona internamente).

RabbitMQ

Este é um message broker open-source que implementa o protocolo de comunicação AMQP (Advanced Message Queiuing Protocol).

As **mensagens** são constituídas por um header com os seus atributos e um payload com o conteúdo e são publicadas para uma entidade **exchange**, que faz a sua distribuição pelas **queues**, seja através de bindings (ligações diretas) ou roteamento com base nos atributos do header. Estas mensagens são por sua vez consumidas através de entrega contínua aos subscritores ou por pedidos dos clientes.



Os **exchanges** podem ser de vários tipos:

Direct Quando as mensagens são encaminhadas para a fila cuja **binding key** (nome da ligação do exchange à fila) corresponde à **routing key** (nome do tópico no cabeçalho) da mensagem.

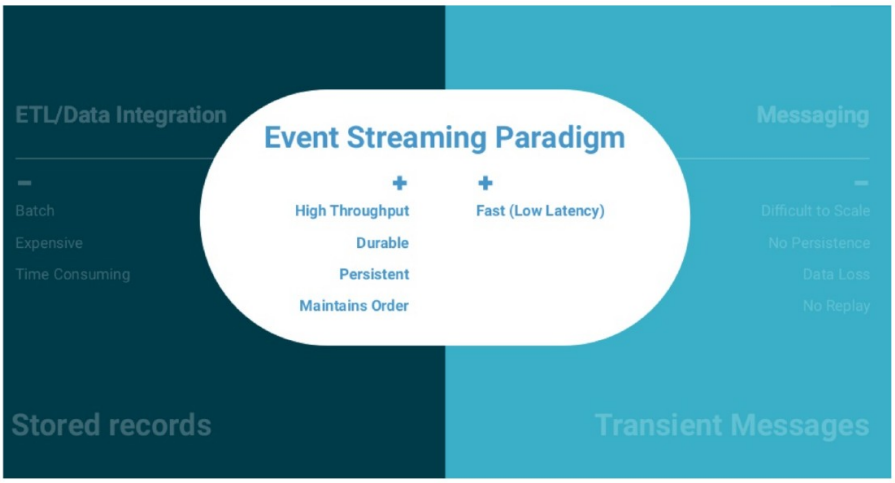
Topic Quando as mensagens são entregues às filas cuja **binding key** corresponde (através de padrões) à **routing key** da mensagem.

Fanout Ignora a **routing key** e envia a mensagem para todas as filas.

A sua utilização com o Spring é bastante simples, bastando adicionar um starter às dependências. +

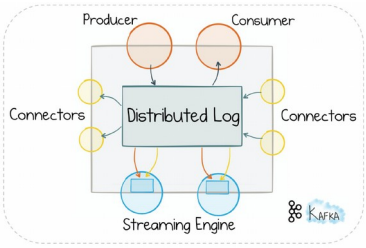
Sistemas de streaming de mensagens

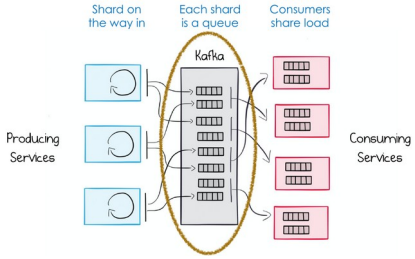
Este modelo é meio termo entre o **registo de eventos** e a **troca transiente de mensagens**, procurando tirar partido dos benefícios de ambos através da oferta de um stream de eventos atualizado continuamente.



Apache Kafka

Esta é também uma ferramenta open-source que oferece um sistema de registo de eventos sequenciais ao longo do tempo do tipo append-log[6], distribuído e imutável.





Os conectores são APIs que permitem a ligação a bases de dados. O streming engine permite a utilização do Kafka através do paradigma de streams, que permite por exemplo em Java ler os conteúdos de um tópico como se fosse um stream sobre uma estrutura de dados ou um ficheiro.

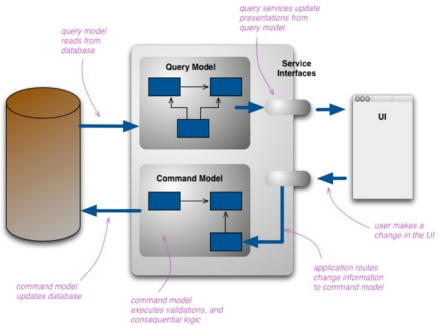
Destaca-se por estar fora da "fronteira de confiança" das aplicações, uma vez que é independentes destas. Caracteriza-se ainda por ser escalável e fazer a gestão de fluxos, ser tolerante a falhas, permitir a concorrência, manter a ordenação e ser stateful (com o "replay" dos eventos podemos saber o estado de determinada entidade).

É muito utilizado para gerir **event-sourcing**, uma vez que para este tipo de eventos a ordem e imutabilidade são cruciais.

6 Escrevemos sempre para a frente. Registos nunca são alterados nem eliminados, só acrescentados.

De forma a garantir o **statefulness** devem ser representadas todas as mudanças de estado como um evento, registados por ordem cronológica, pelo que se recorre ao modelo event-sourcing.

Neste contexto surge ainda o padrão de software **Command Query Response Segregation (CQRS)** que estabelece um desacolpamento entre a escrita e a leitura, ou seja, os processos que fazem uma e outra devem ser distintos.

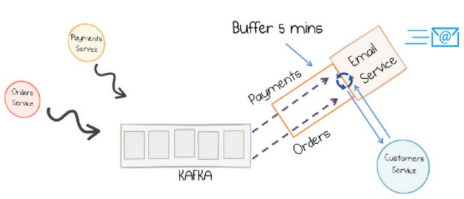


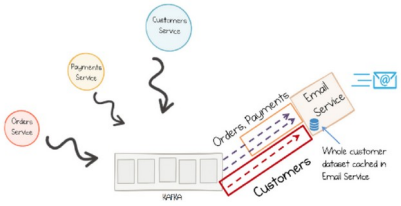
No projeto da UC foi utilizado Kafka como streaming de mensagens entre o gerador de dados, que os escreve e o Spring, que os lê. Temos um desacoplamento entre estas duas operações e por isso aplicado o padrão CQRS.

Como exemplo vamos analisar um serviço de e-mail que recebe o stream de eventos de encomendas de uma loja online e envia um e-mail ao cliente quando este completa o seu pagamento.

Uma forma de implementar este sistema em Kafka é **stateless**, através da inserção dos eventos gerados pelos serviços das encomendas e pagamentos num dos tópicos de forma cronológica e à medida que ocorrem. Neste caso, uma vez feito o pagamento, para cada encomenda, o serviço vai precisar dos dados do cliente para o poder contactar e por isso terá de fazer uma consulta ao serviço dos consumidores, o que vai criar overhead.

A forma **stateful** de o fazer é incluir os dados do cliente num evento gerado para um tópico, de forma a que o serviço de e-mail possa atuar de forma complemente independente e sem necessidade de consultar outros serviços.





A abordagem **stateful** consiste em incluir todos os elementos fundamentais para a atividade dos serviços que subscrevam ao stream de mensagens nos eventos gerados.

Tem como principal **desvantagem** a quantidade de dados transacionados que é um preço a pagar pela **vantagem** do desacoplamento e maior velocidade no processamento de eventos, uma vez que deixa de ser necessário fazer consultas adicionais.

De forma a diminuir a carga exigida por esta abordagem, o Kafka providencia um conjunto de mecanismos que o simplificam, nomeadamente a **replicação** dos estados armazenados em cada nó, **checkpoints do disco** para garantir recuperação dos nós em caso de falha e por fim **compressão dos tópicos** de forma a minimizar o tamanho dos dados.

12. Gestão de qualidade

Slides teóricos e aula teórica assíncrona

Os sistemas de qualidade são orientados ao **cliente**, que através de uma relação de **negócio** adquire um **serviço** que deve seguir um **standart** que tem um **selo de garantia**. Estes sistemas têm como objetivo acompanhar e garantir a **satisfação** do cliente através da introdução de **melhorias** no funcionamento.

Ao nível do **software**, estes sistemas focam-se em garantir que o software desenvolvido cumpre os requisitos que foram propostos com base em **políticas ao nível organizacional e dos projetos** como processos e standarts de desenvolvimento que levam ao desenvolvimento de software de qualidade. Estes partem da premissa que a qualidade do produto desenvolvido é influenciada pela qualidade do seu processo de produção.

A **gestão de qualidade** ocorre a par do processo de desenvolvimento, no final de cada iteração, e deve ser desempenhada por uma **equipa independente** da de desenvolvimento, de forma a garantir a objetividade e imparcialidade.

Cada produto desenvolvido terá a si associado um **plano de qualidade**, que define o produto, qual o seu processo de construção, a descrição do processo, os indicadores de qualidade e a gestão do risco. Este deve ser claro e sucinto.

Por exemplo uma padaria pode ter um plano de qualidade para o produto bolo rei, que tem uma receita para ser construído que descreve os ingredientes e a ordem com que estes são envolvidos. No final da sua confeção, há ainda medidas das dimensões, do nível de açúcar, do tom da cor que conferem os indicadores de qualidade. Por fim deve ser definido o que fazer em caso de uma fornada ser queimada, por exemplo, para garantir uma boa gestão do risco.

No entanto, os **requisitos de qualidade** (eficiência e fiabilidade) e a **qualidade percecionada pelos desenvolvedores** (manutenibilidade e reusabilidade) por vezes entram em conflito. É também de notar a **dificuldade em definir todos os requisitos de qualidade**, pelo que estes geralmente são incompletos e podem até vir a ser inconsistentes e ambíguos, tendo de ser redifinidos ao longo do processo de desenvolvimento.

Conclui-se então que o foco deve ser em garantir que o software cumpre o seu propósito mais do que garantir a confirmidade com as especificações.

São questões habituais perceber se foi testado, se é confiável, se tem uma boa performance, se está bem-estruturado, se seguiu práticas estandardizadas de desenvolvimento e documentação.

Sistemas de gestão de qualidade (QMS)

Estes sistemas têm por base alguns conceitos.

Qualidade é cumprir com os requisitos do cliente

São exemplos a qualidade do serviço prestado, o tempo de entrega, a gestão das reclamações, suporte, consistência dos produtos fornecidos.

Gestão de qualidade são as atividades levadas a cabo pela organização para garantir que os requisitos do cliente são cumpridos

Inspeções, recolha de feedback dos clientes, medições.

Sistema de gestão de qualidade monitoriza os processos que têm impacto na qualidade dos produtos através de métricas de forma a melhorar a sua qualidade e a satisfação dos clientes.

Deve deixar claro quem é responsável por fazer o quê, quando, como e onde.

Na realidade cada empresa pode adotar o seu próprio sistema de gestão de qualidade, o que vai dificultar ao cliente a escolha com base nestes critérios. Para uniformizar a gestão de qualidade foram criados **standarts**. Estes consistem numa agregação de **boas práticas** que definem uma **framework** de trabalho que permite uma **continuidade** das práticas aplicadas na empresa. As empresas que os cumprem são **certificadas**.

A **International Organization for Standardization (ISO)** desenvolve vários tipos de standards, desde símbolos, a materiais, práticas e formas de inspecionar que estes são cumpridos em várias áreas. Em 1987 definiu a norma **ISO 9001**, que define as regras básicas para um sistema de qualidade eficiente.

Documentar o que deve ser feito

Fazer o que é documentado

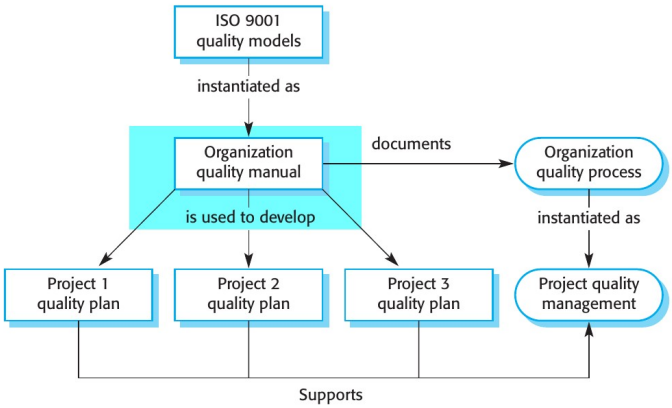
Guardar o que foi feito

Provar que foi feito

A **certificação** do cumprimento de uma determinada norma é feita através de auditorias regulares por entidades independentes. É ainda importante perceber que este não é um requisito da norma! Uma empresa pode aplicar determinada norma sem ser certificada.

As normas são revistas com frequência e periodicamente atualizadas. A versão mais recente da **ISO 9001** é de **2015** e acrescentou à anterior.

A **documentação** dos **processos** gerais de qualidade e dos **procedimentos** necessários a cada um num **manual de qualidade**.



Conclui-se então que esta norma não tem como objetivo estandardizar produtos ou serviços, mas sim **processos**, um conjunto de atividades interrelacionas com base em algum input e que produzem um determinado resultado.

Numa empresa os processos podem ser por exemplo o processo de contratação, o processo de assistência ao cliente, o processo de correção de bugs identificados, o processo de acompanhamento da satisfação do cliente…

Numa pizzaria é fundamental criar o processo de confeção da pizza, com uma receita (mesmo que genérica, sem entrar em demasiado detalhe).

Os processos permitem que haja continuidade e estabilidade, uma vez que mesmo que as pessoas responsáveis pelo processo saiam da empresa, os que vêem a seguir podem continuar a desempenhá-lo da mesma forma, uma vez que os seus procedimentos estão documentados.

No entanto, a certificação limita-se a dar uma garantia quanto ao cumprimento dos standarts, não devendo ser confundida com a qualidade percecionada pelos utilizadores do software. É por isso fundamental que o QMS seja acompanhado por **práticas de engenharia de software**.

Uma destas práticas é o **desenvolvimento ágil**, amplamente utilizado na industria de software, mas cuja base entra em conflito com os princípios do QMS, uma vez que dá preferência a um ambiente de desenvolvimento informal e pouco documentado. No entanto, este conflito está assente na falácia de que é necessária muita documentação num QMS, quando na realidade esta pode ser bastante sucinta e mesmo assim clara quanto ao que define.

O **ISO 9001 ágil** caracteriza-se então pela construção de **livrarias de processos** ricas, organizadas hierarquicamente (em pastas e subpastas) e com diferentes níveis de permissões, com **procedimentos** bem definidos e detalhados com instruções. Deve ainda ter associado um **acompanhamento** de quando os procedimentos são cumpridos, por quem e que progresso foi atingido, um **histórico** da sua evolução com todas as revisões bem fundamentadas.