

SEMANAS:

- CRITÉRIOS DE AVALIAÇÃO
- PORQUÊ AGILE?
- UM CASO DE ESTUDO
- PLANEAMENTO DE ITERAÇÃO
- TESTES UNITÁRIOS
- BOAS PRÁTICAS: O QUE TESTAR?
- TESTES DE INTEGRAÇÃO E FUNCIONAIS
- AULA DE CONSOLIDAÇÃO MATERIA
- PROVA INDIVIDUAL DE AVALIAÇÃO
- REFACTORING
- APLICAÇÃO DE PADRÓES DE REFINAMENTO
- OUTROS MÉTODOS ÁGEIS
- TESTES AUTOMAÇÃO (WEB-APPS)
- APRESENTAÇÕES 1#2
- APRESENTAÇÕES 2#2

## Sumário:

- Introdução e motivação
- Modelos clássicos de desenvolvimento de software
  - Waterfall Model
  - Incremental Model
- Obter agilidade...
- Ágeis versus Tradicionais
  - O manifesto ágil
- Modelos ágeis de desenvolvimento de software



Docente: [Ricardo Henriques](#).

## Introdução

Vamos iniciar esta disciplina com uma pequena introdução, para compreensão do contexto das necessidades e das dificuldades que motivaram o surgimento da família de metodologias designadas de ágeis.

Numa fase inicial da história da computação e da programação, a sua utilização visava, quase em exclusivo, tirar partido da rapidez na resolução de problemas matemáticos e estatísticos.

Assim numa primeira fase, na generalidade dos casos das décadas 40 e 50, os cientistas que ao mesmo tempo tinham em mãos o problema a resolver, eram os próprios que estavam afetos à programação (e muitas vezes eram também responsáveis pela construção do sistema, o computador).

Nesta fase, o desafio principal era computacional/matemático ao nível da modelação do problema/solução e da sua implementação. Apenas alguns exércitos/países detinham riqueza e conhecimento para enveredar por estas soluções.

Durante as décadas 60 e 70, universidades (grandes universidades) e instituições financeiras (banca e seguros) passaram apostar nesta vertente da tecnologia e, desde aí, ficou evidente uma separação entre quem detinha o conhecimento do negócio e quem detinha o conhecimento de desenvolvimento, das componentes computacionais e de programação.

Surgiram grandes empresas com soluções verticais - que incluíam desde a produção do hardware, produção do sistema de exploração (ou sistema operativo), programação, e comunicação (redes).

No início da década de 80, com a assinatura do contracto entre a IBM e a Microsoft, a democratização do computador pessoal permitiu a abertura da informática a pequenas e médias empresas e até aos utilizadores individuais.

Em definitivo a produção de software ficou independente do detentor do conhecimento de negócio. Se durante os anos 60 e 70 as grandes empresas podiam desenvolver sistemas para uma mesma família de empresas, ao abrir-se para a generalidade das empresas, as áreas de negócio e aplicação da informática diversificou-se de forma tão exponencial, qui ↑

foi necessário mudar o paradigma de construção de *software*.

## SW Development Life Cycle (SDLC)

De forma tradicional, e semelhante a outras engenharias, o ciclo de desenvolvimento de *software* segue um padrão representado na figura 1. Vejamos em detalhe cada uma das fases:

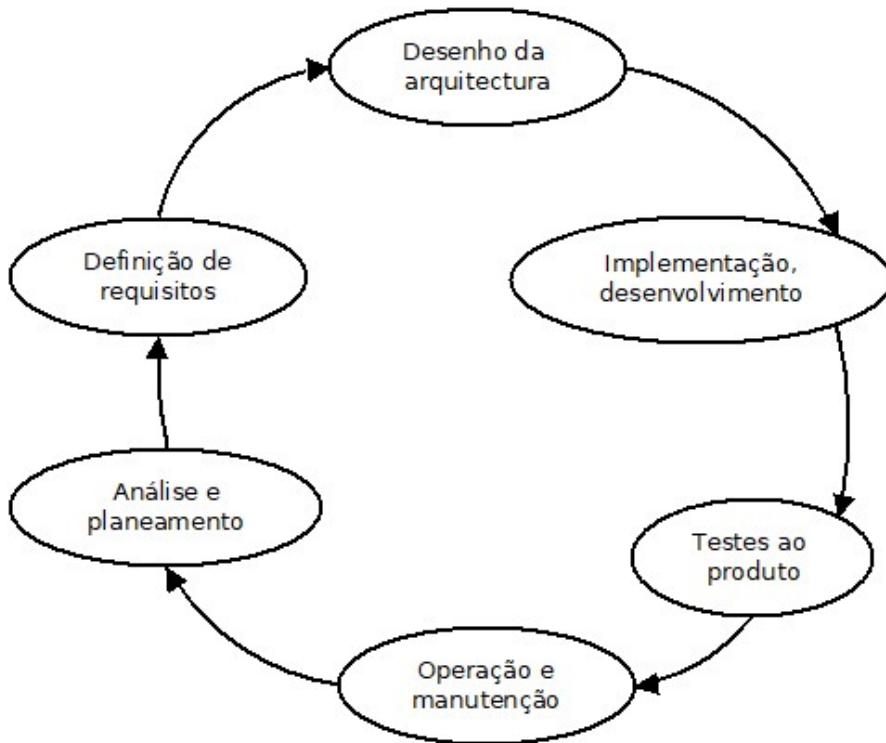


Fig. 1: ciclo de vida do desenvolvimento do SW

1. **Análise de requisitos e planeamento:** é a fase mais importante do SDLC; normalmente

executada por membros seniores da equipa, e na qual se faz uso da informação indicada pelos clientes, dos diversos departamentos da empresa. Esta informação é então usada para a definição do plano base do projeto e o estudo da viabilidade económica do mesmo. É ainda nesta fase que a equipa traça os requisitos de garantia de qualidade e identifica riscos potenciais à sua boa execução.

2. **Definição de requisitos:** após os requisitos serem analisados, os requisitos do produto ficam definidos e documentados. Devem ser aprovados pelo cliente ou pelo analista de mercado através do *Software Requirement Specification* (SRS). O SRS é um documento importante pois lista todos os requisitos do produto que deverão ser desenhados e desenvolvidos nas fases subsequentes do ciclo de desenvolvimento.

3. **Desenho da arquitetura:** o SRS é a referência de base a partir da qual, os arquitetos de sistemas, idealizam a melhor arquitetura para criação do produto. Pelo menos uma arquitetura de produto é proposta e documentada no *Design Document Specification* (DDS). Este documento é revisto por todas as partes interessadas de forma que a melhor abordagem seja seguida, baseada nalguns parâmetros tais como:

- avaliação de risco;
- robustez do produto;
- método de desenho/modelação;
- constrangimentos de tempo e orçamento.

Sugestão de leitura: [Software Requirement Specification IEEE 29148](#)

Uma abordagem de desenho deve definir todos os módulos da arquitetura do produto a par dos fluxos de comunicações e dados de e para módulos externos (de terceiros, caso existam). O desenho interno de todos os módulos deve ficar bem definido na proposta de arquitetura e apresentado de forma clara esses detalhes no DDS.

4. **Implementação, desenvolvimento:** nesta fase começa o desenvolvimento do produto. O código fonte é gerado nesta fase. Se o desenho foi tratado com detalhe e de forma organizada o código fonte pode ser desenvolvido sem complicações. Os programadores devem seguir as regras de implementação da sua organização. Para gerar código devem usar ferramentas de programação, como compiladores, interpretadores, debuggers, etc. O código fonte é escrito em linguagens alto nível, como por exemplo C/C++, C#, Delphi, Java, PHP, Python. A(s) linguagem(ns) de

programação são escolhidas em função do licenciamento disponível e do *software* a desenvolver.

5. **Testes ao produto:** esta fase é normalmente um subconjunto de todas as fases nos modelos modernos de SDLC. Ainda assim esta fase apenas envolve as situações onde são reportadas as falhas do produto: reportar, solucionar e reanalisar, até que fique de acordo com os requisitos de qualidade do SRS.
6. **Operação e manutenção:** uma vez testado, o produto fica pronto para ser lançado no mercado. Pode ser lançado, ou posto no mercado apenas num segmento limitado para testes em situações reais, até que possa ser alargado a todo o mercado, sem mais, ou contando com a incorporação de melhorias propostas pelo clientes envolvidos nos testes. Após o lançamento é efetuada a manutenção do produto.

Ao longo desta disciplina teremos oportunidade de ver como diferentes metodologias de desenvolvimento abordam a definição e estruturação dos requisitos. Porém, para já, precisamos de fazer um parentesis, para obtermos uma visão comum sobre do que são os *requisitos de software*.

## Requisitos de software

A definição de requisito de Brian Lawrence, indica que um requisito é *algo que conduz ou condiciona as escolhas do desenho da solução*. É também comum definir-se, "um requisito é uma propriedade que um produto deve ter para trazer valor ao cliente, ou a uma parte interessada (stakeholder)". A definição que vamos assumir nesta disciplina será a definição de Ian Sommerville e Pete Sawyer, que nos diz: "*um requisito é uma especificação do que deverá ser implementado. São descrições de como o sistema se deve comportar, ou a especificação da(s) propriedade(s) do sistema. O requisito pode implicar uma restrição no processo de desenvolvimento do sistema*".

Para nós, "malta do *software*" a noção de requisito não tem o mesmo significado da definição do dicionário. Desde logo, um requisito de *software* inclui uma dimensão temporal. Quando mencionado no tempo presente, descreve uma capacidade do sistema atual. Se dito para um futuro imediato, então terá alta prioridade; se num meio-termo, será moderadamente prioritário; quando mencionado de forma hipotética no futuro, assume-se tratar-se de uma implementação com baixa prioridade.

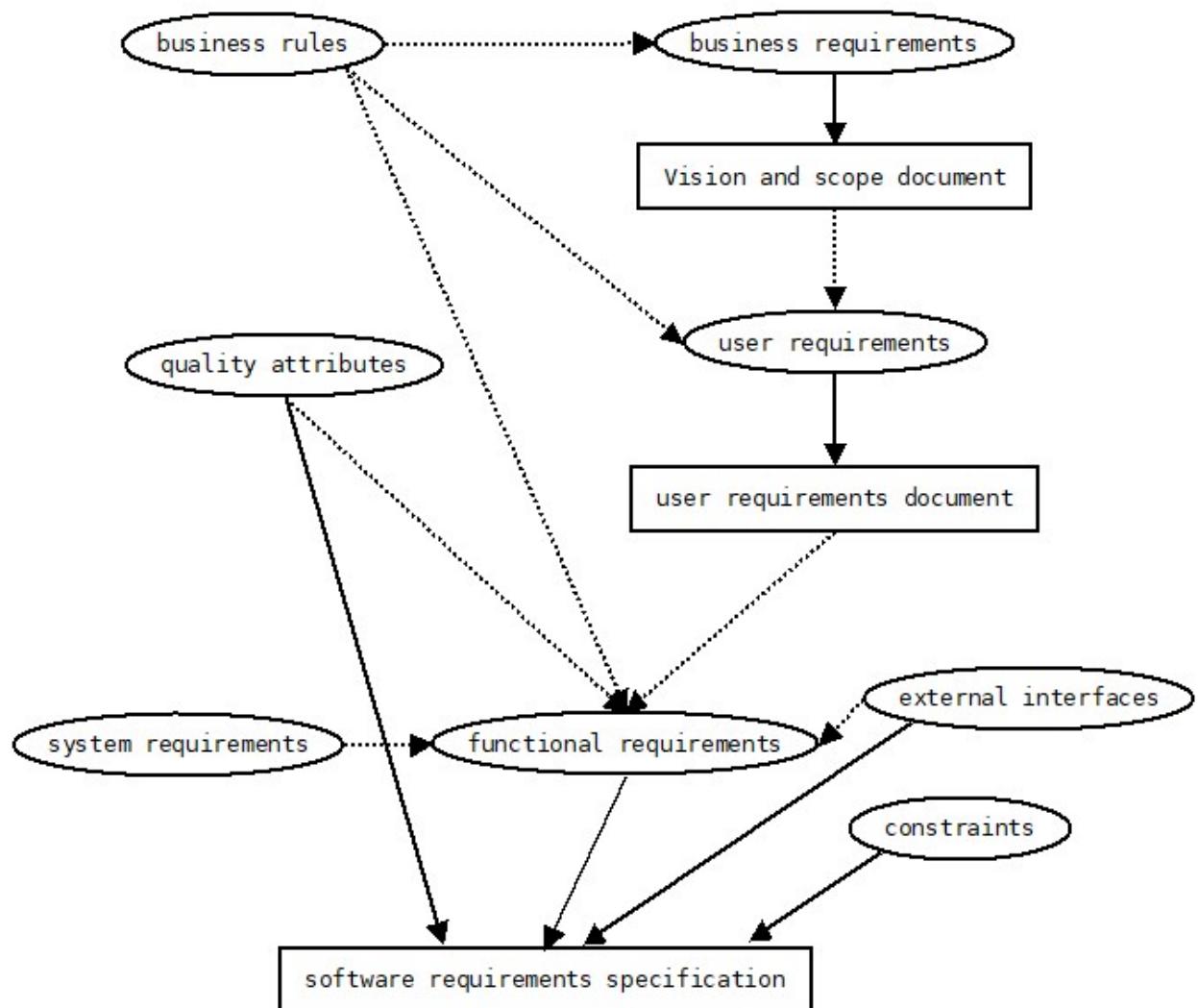
Não desperdice energia em debater se uma coisa é ou não um requisito, mesmo que à partida saiba que nunca irá ser implementado, por uma qualquer motivação negocial. Assuma tratar-se de um requisito.

Uma vez que existem muitos tipos diferentes de requisitos, é útil apresentarmos um conjunto consistente de adjetivos, em linha com os termos mais comuns, e presentes na literatura no domínio dos requisitos.

| Termo   | Definição  |
|---|--|
| Requisito de negócio<br>( <i>business requirement</i> )             | Concretiza um objetivo de alto nível, em relação à instituição que desenvolve um produto, ou de um cliente que o procura.  |
| Regra de negócio<br>( <i>business rule</i> )                        | Traduz uma política, uma diretriz, um <i>standard</i> ou regulamento que define ou impõe algum aspeto da lógica de negócio. Não sendo um requisito de <i>software</i> em si, pode dar origem a vários tipos de requisitos de <i>software</i> . |
| Restrição, ou invariante  | Limitação ao domínio ou às escolhas disponíveis para o desenho e/ou construção de uma aplicação.   |
| Requisito de interface<br>( <i>external interface requirement</i> ) | Contém a descrição quanto à ligação/conexão entre sistemas, ou entre o sistema de <i>software</i> e um utilizador, ou entre o sistema e um dispositivo.  |
| Característica ( <i>feature</i> )                                   | Aspeto do sistema com valor para o utilizador final, descrita por um conjunto de requisitos funcionais, logicamente relacionados.  |

| Termo   | Definição  |
|---|--|
| Requisito funcional<br>( <i>functional requirement</i> )        | Descreve um comportamento do sistema, no âmbito de condições específicas.  |
| Requisito não funcional<br>( <i>nonfunctional requirement</i> ) | Descreve uma propriedade ou característica que o sistema deve respeitar.   |
| Atributo de qualidade<br>( <i>quality attribute</i> )           | Descreve um serviço ou caracteriza o desempenho de um tipo de requisito não funcional.   |
| Requisito de sistema<br>( <i>system requirement</i> )           | Descrição de um requisito de alto nível, de um produto ou de um sistema, eventualmente composto por subsistemas.                       |
| Requisito de utilização<br>( <i>user requirement</i> )          | Descreve um atributo desejado para um produto, ou uma capacidade que o sistema deve proporcionar a perfis específicos de utilizadores. |

..... → é origem de / influencia  
→ fica registado em



## Exercício

Como nem sempre é claro, onde começa ou acaba a fronteira de um tipo de requisito (ver termos na tabela [acima](#)), experimente preencher, em paralelo, com exemplos fictícios de requisitos:

- Para um projeto de criação de um novo modelo rodoviário de transporte coletivo de pessoas e ↑

mercadorias.

- Para um projeto de *software* de bilhética ferroviária.

Para normas de *software* sugere-se a visita à proposta para alteração das [prescrições mínimas de segurança e saúde](#), respeitantes ao trabalho com equipamento dotados de visor.

## Modelos clássicos de desenvolvimento de *software*

Existem muitos modelos de desenvolvimento de *software* e muitas instituições criam os seus próprios modelos. A escolha do modelo tem grande impacto no sucesso do processo de desenvolvimento de *software*.

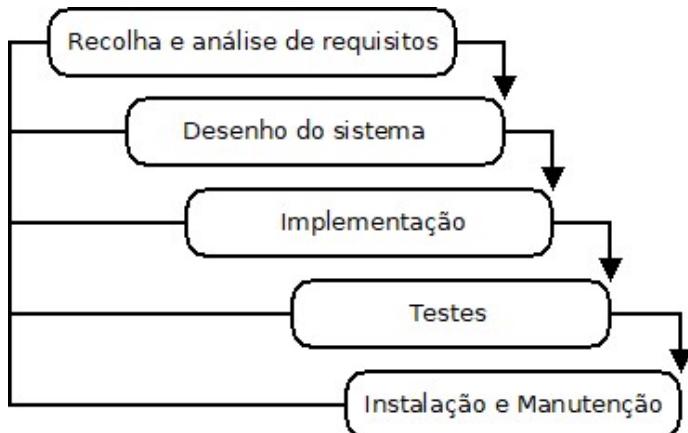
Entre os modelos mais utilizados encontram-se:

- *Waterfall Model*
- *V Model*
- *Incremental Model*
- *RAD (Rapid Application Development) Model*
- *Agile Model*
- *Iterative Model*
- *Spiral Model*

Em cada modelo podem-se apontar vantagens e desvantagens e, como tal, a escolha e aplicação de um deles em detrimento de outro deve ser uma ação refletida e ponderada: de acordo ao tipo de projeto e de acordo com as necessidades da instituição ou empresa.

### **Waterfall Model**

Este modelo foi constituído por Winston Royce em 1970 e é conhecido como um modelo linear cíclico. É fácil de compreender e aplicar: cada fase deve ficar completa antes da seguinte poder ser iniciada. No fim de cada fase é feita uma revisão para garantir a observância em relação aos requisitos do projeto.



**Fig. 2: Waterfall Model**

Algumas das **vantagens** deste modelo são:

- A existência de documentação e estrutura são uma vantagem para a rápida inclusão de novos membros na equipa;
- É um modelo linear, fácil de compreender;
- Dada a rigidez que impõe a coordenação é mais fácil - cada fase tem um resultado esperado e um processo de avaliação;
- As fases são implementadas sequencialmente e uma de cada vez.

Algumas das **desvantagens** apontadas a este modelo são:



- Caso surjam novos requisitos após a recolha de requisitos estar completa tal pode influenciar, negativamente, o desenvolvimento do produto;
- Nem todos os problemas detetados numa fase conseguem ficar tratados (ou completados) nessa mesma fase - podem obrigar a voltar à fase inicial;
- Não permite flexibilidade no particionamento do projeto em fases;
- A adição de novos requisitos pelo cliente levam ao aumento de custos, uma vez que não podem ser assimilados na edição do produto já em desenvolvimento;
- É difícil estimar o tempo e o orçamento para cada fase;
- Não se obtêm protótipos até que o ciclo de desenvolvimento fique concluído;
- Se na fase de testes se detetarem problemas é custoso regressar à fase de desenho;
- Existe um grande risco e incerteza quanto à total aceitabilidade pelo cliente;
- Não é recomendado para projetos complexos ou orientados aos objetos.

Este modelo aplica-se com maior sucesso nos seguintes casos:

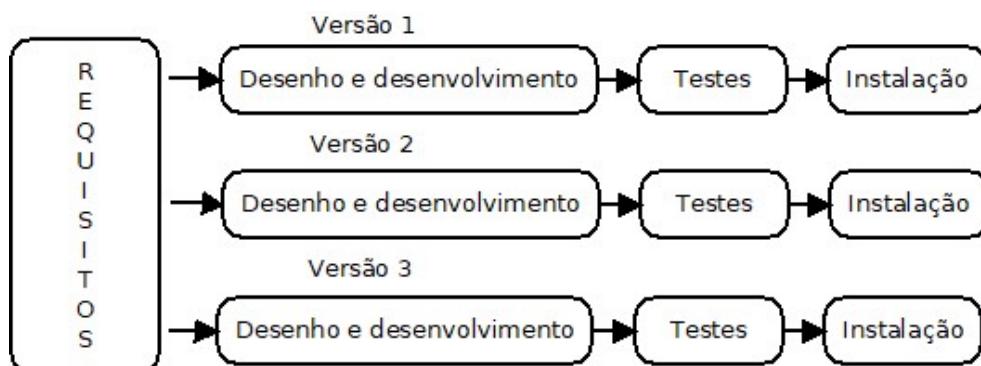
- Pequenos projetos;
- Os requisitos são bem compreendidos e mantém-se claros até ao final;
- A definição do produto é estável;
- A tecnologia é bem compreendida/conhecida pela equipa;
- Não existem requisitos ambíguos;
- Estão disponíveis na equipa os recursos que envolvam conhecimentos (*expertise*);

### **Incremental Model**

Neste modelo os requisitos estão divididos em subconjuntos e o modelo envolve múltiplos ciclos de desenvolvimento. O ciclo inteiro de desenvolvimento assemelha-se a ter *múltiplos waterfall*.

Os ciclos são novamente divididos em pequenos ciclos e os módulos são por isso mais facilmente geríveis. Cada módulo passa pelo processo de análise, desenho, implementação e testes.

Durante o primeiro módulo, é constituída uma versão funcional de *software*. É adicionada nova funcionalidade a cada versão subsequente. O processo continua até o sistema ficar completo.



**Fig. 3: Incremental Model**

Algumas das **vantagens** deste modelo são:

- Em cada fase há uma entrega funcional de um produto que cumpre alguns dos requisitos do cliente;
- Os protótipos são entregues ao cliente;
- O *feedback* do cliente fica distribuído ao longo de todo o processo de desenvolvimento;
- É mais flexível - diminui o ónus quando o propósito ou os requisitos do produto são alterados;
- É mais fácil de testar e depurar durante uma mais breve iteração.

- Diminui os custos da entrega inicial;
- O risco é mais fácil de gerir uma vez que os riscos são identificados e geridos durante a iteração;
- Quando são identificados novos requisitos eles podem ser acomodados para o próximo protótipo;

Algumas das **desvantagens** apontadas a este modelo são:

- Exige um bom planeamento e desenho;
- Exige uma definição completa e clara dos requisitos de todo o sistema antes que o mesmo possa ser dividido para construção incremental;
- Os custos totais são mais elevados do que o *waterfall model*;
- Erros de desenho são difíceis de resolver e remover;
- Pode descontrolar-se e deslizar para um modelo de "*codificação e reparação*";
- O cliente, ao ver o que está feito pode querer pedir mais (pagando o mesmo, o típico: "*já agora*");
- As abordagens orientadas aos objetos dão uma estrutura melhor para o desenvolvimento evolutivo, de uma forma iterativa.

Este modelo aplica-se com maior sucesso nos seguintes casos:

- Os requisitos do sistema estão claramente definidos e compreendidos;
- Os requisitos mais importantes não mudam; alguns detalhes podem mudar ao longo do tempo de desenvolvimento do projeto;
- Ser necessário desde cedo haver uma versão inicial disponível;
- Quando se antecipar a introdução na equipa de uma tecnologia nova;
- Quando hajam riscos elevados quanto às características ou objetivos do produto.

## **Ágeis versus Tradicionais**

As metodologias ágeis são baseadas em métodos adaptativos de desenvolvimento de *software*, ao passo que os modelos tradicionais assentam numa abordagem preditiva.

Os modelos tradicionais SDLC, as equipas trabalham sobre um plano detalhado e têm a lista completa das características e tarefas que devem completar, para um intervalo de alguns meses ou até, desde logo, para todo o ciclo completo até chegar ao produto.

Os métodos preditivos dependem da análise completa dos requisitos e do planeamento cuidadoso no início do ciclo. Qualquer requisito a ser incluído deve seguir um controlo estrito de gestão e ser devidamente definida a sua prioridade.

Os modelos ágeis utilizam uma abordagem adaptativa, substituindo o planeamento detalhado por uma visão clara, mas de curto alcance, para um futuro próximo das tarefas imediatas e das características que devem ser implementadas (agora).

A equipa adapta-se às alterações dinâmicas dos requisitos do produto. O produto é frequentemente testado, minimizando o risco de grandes desvios no futuro.

A iteração com os clientes é intensa, a comunicação deve ser franca e a documentação a mínima necessária. As equipas colaboram em grande proximidade e muitas vezes a sua localização geográfica é a mesma.

Enquanto que as metodologias ágeis de SDLC são mais usadas em projetos de média e pequena dimensão, os métodos tradicionais são muitas vezes aplicados em projetos de grande dimensão.

É por isso importante que a equipa escolha qual o SDLC que **melhor se adapta ao projeto em causa**. E a dimensão do projeto não é o único critério. Vejamos outros fatores:

- Dimensão da equipa;



- Localização geográfica (e a facilidade de comunicação);
- Tamanho e complexidade das tarefas;
- Tipo de projeto;
- Estratégia de negócio;
- Capacidade de engenharia;
- ...

Antes de tomar qualquer decisão, é importante a equipa estudar as diferenças, as vantagens e as desvantagens de cada SDLC no contexto de negócio (e se tem historial de trabalho com o cliente), dos requisitos legais e industriais.

É importante avaliar a escolha do SDLC e do processo de seleção. A escolha apropriada do SDLC é uma decisão de gestão com implicações a longo termo no sucesso do projeto.

Embora as metodologias ágeis triunfem sobre as tradicionais em vários aspectos, há muitas dificuldades a ultrapassar para que funcionem bem. Desde logo uma significativa redução da documentação e o assumir que o código deve ser a documentação é uma delas. É por isso importante que os programadores que utilizem metodologias ágeis coloquem, a par do código, comentários que clarifiquem a implementação.

É também difícil para programadores juniores ou aos novos membros da equipa completarem as tarefas, quando nem sequer compreendem totalmente o projeto. Por isso farão bastantes perguntas aos outros programadores, e tal pode atrasar a completação da iteração e, com isso, aumentar os custos de desenvolvimento. Em oposição, os métodos tradicionais, ao dedicarem mais esforço na documentação, na orientação e clarificação dos objetivos, ajudam a equipa de desenvolvimento, mesmo quando chegam novos membros.

Por isso, e de forma bem reconhecida, as metodologias ágeis dão grande importância à comunicação e ao envolvimento do cliente.

Para cada versão entregue a equipa de desenvolvimento e o cliente organizam uma reunião, onde a equipa apresenta o trabalho feito na iteração finalizada e o cliente dará o necessário *feedback* sobre o software entregue: melhoramentos sobre as funcionalidades implementadas e/ou o pedido de novas.

Muitas vezes essas reuniões periódicas (às vezes semanais) são mal geridas, tornando-se cansativas e aborrecidas, muitas vezes obrigando a equipa apresentar repetidamente módulos e funcionalidade, a membros novos ou ao cliente e seus interlocutores.

Outro aspecto bem conhecido é por vezes a dificuldade da equipa e do cliente encontrarem o tempo certo para cada iteração. Por vezes a equipa irá achar muito apertado para obter o desenvolvimento necessário, ora porque a complexidade é grande, ou a perícia e a dimensão da equipa não é suficiente. Tal levará a atraso na iteração e a mina a confiança do cliente.

Por seu turno, as metodologias tradicionais tem um modelo onde os requisitos estão bem definidos antes de se iniciar a implementação, a programação. O cliente não participa nesta fase e a codificação irá ser feita com base na documentação (boa ou má) do analista até que o sistema esteja completo e, só nessa fase, será apresentado ao cliente.

Com isto a equipa não tem de se ocupar com reuniões frequentes e tem mais tempo focado no desenvolvimento do sistema. Se tudo correr bem e de acordo com a análise, haverá a entrega de um bom produto em menor tempo.

A tabela abaixo caracteriza as principais diferenças entre os métodos de desenvolvimento tradicionais e ágeis.

|                      | <b>Desenvolvimento tradicional</b> | <b>Desenvolvimento ágil</b>                                       |
|----------------------|------------------------------------|---|
| Hipótese fundamental | Os sistemas são completamente      | SW adaptativo de alta qualidade desenvolvido por pequenas equipas |

|   | <b>Desenvolvimento tradicional</b>  | <b>Desenvolvimento ágil</b>   |
|---|---|---|
|   | específicos, previsíveis e podem ser desenvolvidos mediante um detalhado e cuidadoso planeamento. | mediante aplicação de princípios de melhoramento contínuo de desenho e teste, baseado em curtos ciclos de desenvolvimento e <i>feedback</i> e alterações. |
| Estilo de gestão                                    | Comando e controlo  | Liderança e colaboração   |
| Gestão do conhecimento                              | Explícito   | Táctico   |
| Comunicação   | Formal  | Informal  |
| Modelo de desenvolvimento                           | <i>Life cycle model (waterfall, spiral, ou semelhante).</i>                                       | Modelo entrega-evolução ( <i>evolutionary-delivery model</i> )  |
| Estrutura organizativa                              | Mecânica (burocrática, bastante formal), grandes empresas/organizações.                           | Orgânica (flexível e participativa, valoriza a cooperação social), pequenas e médias empresas/organizações.   |
| Controlo de qualidade                               | Planeamento difícil e rigoroso. Testes tardios.   | Controlo permanente dos requisitos, do desenho e das soluções. Testes contínuos.  |
| Testes  | Após a completação do código.   | A cada iteração.  |
| Requisitos do cliente final                         | Detalhados e definidos antes da implementação (codificação).                                      | Introduzidos interativamente.   |
| Acompanhamento e interação com o cliente            | Baixo.  | Elevado/Intenso.  |
| Custo de recomeçar                                  | Elevado.  | Baixo.  |
| Rumo de desenvolvimento                             | Fixo.   | Adaptável.  |
| Capacidades auxiliares da equipa de desenvolvimento | Nenhuma em particular.  | Relacionamento interpessoal e conhecimento mínimo do negócio.   |
| <i>Developers</i>                                   | Orientados ao plano, com habilitações adequadas; acesso a conhecimento exterior.                  | Ágeis, com habilidades avançadas, co-localizados e cooperativos.  |
| Requisitos  | Muito estáveis; conhecidos antecipadamente.   | Emergentes, com mudanças rápidas.   |
| Arquitetura   | Desenho para os requisitos atuais e previsíveis.  | Desenho para os requisitos atuais.  |
| Remodelação   | Dispendiosa.  | Não dispendiosa.  |
| Dimensão  | Grandes equipas e projetos.   | Equipas pequenas e pequenos e médios projetos.  |
| Objetivo primário                                   | Maior segurança.  | Valor (retorno) rápido.   |

## Detalhe da documentação

Nos projetos em *waterfall*, uma vez que a equipa de desenvolvimento passará a ter pouca interação com o cliente, assim que se inicie a construção da solução, os requisitos devem especificar o comportamento do sistema, a relação entre dados, a interação e a experiência com o utilizador, e todas as expectativas em grande detalhe. Uma colaboração mais próxima entre o cliente e a equipa de desenvolvimento, nos projetos ágeis,



geralmente significam que os requisitos podem ser documentados com menor detalhe.

Porém, não se pense que nos projetos de desenvolvimento assente em metodologias ágeis não é necessário as equipas escreverem os requisitos. Tal é impreciso. As metodologias ágeis encorajam criação de documentação mínima que permita guiar com exatidão o desenvolvimento e testes. Toda a documentação para além do que as equipas de desenvolvimento e testes - ou necessária para satisfazer regulamentação ou *standards* - representa um desperdício de esforço.

## Modelos ágeis de desenvolvimento de software

Há múltiplos métodos ágeis de desenvolvimento de *software* e, embora cada um tenha a sua abordagem particular, todos partilham os valores e as visões do manifesto ágil.

Todos valorizam a comunicação permanente, planeamento, teste e integração. Ajudam a desenvolver bom *software*, mas o que define os métodos ágeis é que encorajam a colaboração e tornam as decisões comuns (partilhadas) boas e rápidas.

Entre muitos, alguns dos mais reconhecidos são:

- *Adaptive Software Development* (ASD);
- *Feature Driven Development* (FDD);
- *Crystal Clear*;
- *Dynamic Software Development Method* (DSDM);
- SCRUM;
- *eXtreme Programming* (XP);
- *Rational Unify Process* (RUP);



Fig. 5: versão portuguesa

### Em resumo:

O desenvolvimento de aplicações de *software* é uma atividade que envolve complexos processos os quais estão predispostos a falhas, qualquer que seja o método escolhido.

Qualquer sistema de qualidade de desenvolvimento de *software* deve assegurar que a funcionalidade é a correta e implementada de acordo com as especificações; foi testada e validada antes de ser colocada em produção.