Introdução à Engenharia de Software

José Mendes 107188

2023/2024

1 Maven

1.1 O que é o Maven?

É uma **ferramenta de gestão de projetos**, que inclui:

- Um **project object model** (POM) que descreve o projeto;
- Um conjunto de **standards**;
- Um **lifecycle** do projeto;
- Um sistema de gestão de **dependências**;
- Lógica para **executar plugins** em **fases** específicas do ciclo de vida.

Convenção sobre configuração (layout do projeto é padronizado).

1.2 Layout de Diretórios Padronizado

POM - Contém uma descrição completa do projeto de como construir o projeto.

src - Diretório que contém todo o código fonte para construir o projeto, o seu site, ...

target - Diretório que contém os resultados da construção, tipicamente um JAR ou WAR, juntamente com os ficheiros intermedios.

1.3 POM

Maven é baseado no conceito de um **Project Object Model** (POM). Este é um ficheiro XML, que está sempre localizado no diretório base do projeto como **pom.xml** (os users definiram POMs que estendem o Super POM).

O POM contém informação sobre o projeto e vários detalhes de configuração usados pelo Maven para construir o projeto.

O POM é declarativo, não necessita de detalhes de procedimento.

1.3.1 Estrutura do POM

O POM contém 4 categorias de descrição e configuração:

- Informação geral do projeto, isto é, informação human-readable;
- Configuração do build, que pode incluir, adicionar plugins, afixar plugins objetivo ao ciclo de vida;
- Ambiente de construção, que descreve o ambiente "familiar" em que o Maven está;
- Relações POM, isto é, coordenadas, herança, agregação, dependências.

1.4 Coordenadas Maven

As coordenadas definem o lugar único do projeto no universo Maven. São compostas por 3 partes: <**groupId**>, <**artifactId**> e <**version**> (The Maven trinity!).

As versões de um projeto são usadas para agrupar e ordenar lançamentos:

$$< major\_version > . < minor\_version > . < incremental\_version > - < qualifier >$$

Exemplo: 1.0.0-SNAPSHOT ou 1.2.3-alpha-2

Se o qualifier contiver a palavra chave SNAPSHOT, então o Maven vai expandir este token para uma data e hora convertida para o formato UTC.

- **groupId** - Nome da empresa, organização, equipa, . . . , normalmente usando a convenção de nomes de domínio invertidos (reverse URL naming, ex: org.apache.maven);
- **artifactId** - Nome único do projeto dentro do groupId;
- **version** - Versão do projeto;
- **packaging** - Tipo de empacotamento do projeto (jar (default), war, . . .);
- **classifier** - Classificador opcional para distinguir artefactos

Nota: As coordenadas Maven identificam unicamente um projeto.

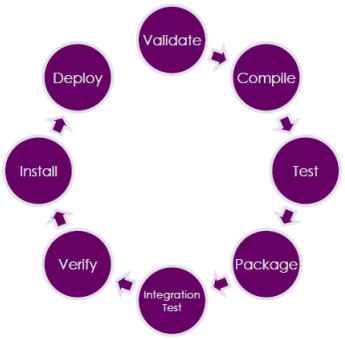
1.5 Ciclo de Vida Maven

Um ciclo de vida é uma sequência organizada de fases, que dão ordem a uma sequência de objetivos. Estes objetivos são empacotados em plugins que estão ligados as fases.

Clean Lifecyle
pre-clean
clean
post-clean

Default Lifecyle	
validate	test-compile
initialize	process-test-classes
generate-sources	test
process-sources	prepare-package
generate-resources	**package**
process-resources	pre-integration-test
compile	integration-test
process-classes	post-integration-test
generate-test-sources	verify
process-test-sources	**install**
generate-test-resources	**deploy**
processs-test-resources	

Chamar uma fase específica num ciclo de construção, vai executar todas as fases anteriores a essa fase.



1. **Validate** - Valida que a estrutura do projeto está correta. (ex: verifica se todas as dependências foram transferidas e estão disponíveis no repositório local);
2. **Compile** - Compila o código fonte, converte os ficheiros **.java** em **.class**, e armazenando-os no diretório **target/classes**;
3. **Test** - Corre testes unitários para o projeto;
4. **Package** - Empacota o código compilado num formato distribuível como **JAR** ou **WAR**;
5. **Integration Test** - Corre testes de integração para o projeto;
6. **Verify** - Corre verificações para verificar que o projeto é válido e que cumpre os critérios de qualidade;
7. **Install** - Instala o código empacotado no repositório Maven local, para uso como dependência noutros projetos locais;
8. **Deploy** - Copia o pacote final de código para o repositório remoto para partilha com outros developers e projetos.

1.6 Ciclo de Vida de Construção

O processo para contruir e distribuir um projeto. Consiste em vários passos designados por **fases**.

Algumas fases default são:

- **validate**
- **compile**
- **test**
- **package**
- **deploy**

1.7 Goals e Plugins

Os Goals são operações fornecidas pelas ferramentas Maven.

Cada fase é uma sequência de Goals, em que cada Goal é responsável por uma tarefa específica. Quando corremos uma fase, todos os Goals ligados a essa fase são executados, na ordem em que estão definidos.

Algumas Maven Plugins:

- resources
- compiler
- surefire
- jar, war

1.8 Arquétipos (Archetypes)

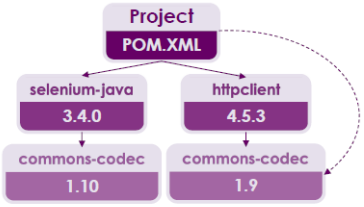
Um Archetype é um template para um projeto Maven, que pode ser usado para criar um novo projeto rapidamente.

Exemplo: *maven-archetype-quickstart* ou *maven-archetype-webapp*

Users podem criar os seus próprios Archetypes e publicá-los através de catálogos.

1.9 Gestor de Dependências

Uma **dependência** de um projeto é uma biblioteca da qual o projeto depende. Adicionar uma dependência ao projeto é simples, basta adicionar a dependência ao POM. O Maven vai automaticamente procurar a dependência no repositório local, e se não encontrar, vai procurar no repositório remoto e transferi-la.



2 Git e GitHub

2.1 Sistemas de Controlo de Versões

Um sistema de controlo de versões (também conhecido como sistema de controlo de código fonte) faz o seguinte:

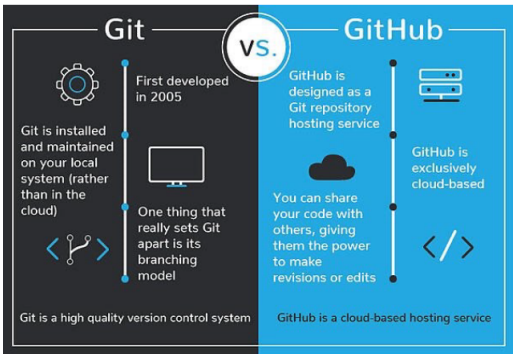
- Mantém várias (antigas e novas) versões de tudo (não só código fonte);
- Pede por comentários quando se fazem alterações;
- Permite "check-in" e "check-out" de ficheiros para saber em que ficheiros outras pessoas estão a trabalhar;
- Mostra as diferenças entre versões;

2.1.1 Vantagens

Ao trabalhar sozinho: Fornece uma "máquina do tempo" para voltar atrás para uma versão anterior, e fornece um bom suporte de diferentes versões do mesmo projeto.

Ao trabalhar em equipa: Simplifica muito trabalhar em concurrencia, dando "merge" de alterações feitas por diferentes pessoas.

2.2 o que é Git e GitHub



Quando fazemos "git init" num diretório de um projeto, ou quando fazemos "git clone" de um projeto existente, o Git cria um repositório (.git).

Em qualquer momento, podemos fazer um "snapshot" de tudo no dirétorio do projeto e guardar este no repositório. Este "snapshot" é chamado de **commit object**.

Um **commit** ocorre quando fazemos alterações que estão prontas para serem guardadas no repositório.

Quando realizamos um commit, o Git guarda um **commit object**:

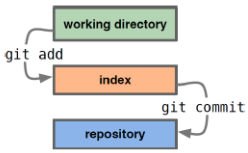
- Um estado completo do projeto, incluindo todos os ficheiros;
- O primeio não possui pai;
- Normalmente, pegamos num commit object, fazemos alterações, e criamos um novo commit object, pelo que a maior parte dos commit objects têm apenas um pai;
- Quando fazemos **merge** de dois commit objects, forma um commit object com dois pais.

Pelo que, os commit objects formam uma **DAG** (Directed Acyclic Graph). O Git é tudo sobre usar e manipular este grafo.

2.2.1 Mensagem de Commit

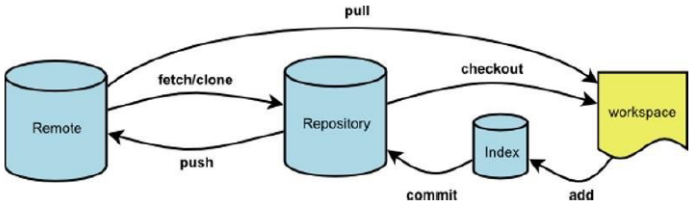
Os commits são "baratos" pelo que os devemos fazer com frequência, e com mensagens descritivas sobre o que foi alterado. Devem ter apenas uma linha.

Como não devemos dizer muito numa linha, devemos fazer vários commits.



2.3 Manter simples

- If you:
  - Make sure you are current with the central repository
  - Make some improvements to your code
  - Update the central repository before anyone else does
- Then you don't have to worry about resolving conflicts or working with multiple branches
  - All the complexity in git comes from dealing with these
- Therefore:
  - Make sure you are up-to-date before starting to work
  - Commit and update the central repository frequently

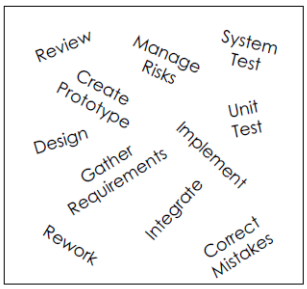


3 O Processo de Desenvolvimento de Software

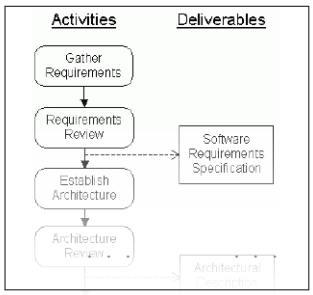
3.1 Processo

A fundação para a Engenharia de Software é a camada de processo. Um processo de Software é uma framework para as atividades, ações, e tarefas necessárias para construir software de alta qualidade. Define as técnicas e a framework de gestão para aplicação de métodos, ferramentas e pessoas ao longo do processo de desenvolvimento.

3.2 Porquê o Processo de Software?



Developing software without a defined process is chaotic and inefficient



A process makes software development more orderly and manageable

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

3.3 Processo de Software

Existem vários tipos de processos de software, no entanto, todos têm:

- **Especificação (comunicação e planeamento)** - definir o que o sistema deve fazer;
- **Design e Implementação** - definir a organização do sistema e implementar o sistema;
- **Validação** - verificar que faz aquilo que o cliente quer;
- **Evolução** - alterar o sistema em resposta a novos requisitos impostos pelo cliente.

Quando discutimos sobre o processo de software, estamos a falar sobre:

- **Atividades** - como especificar um modelo de dados, design de uma interface de utilizador, ...;
- **Ordem** a ordem destas atividades;

A descrição de processos pode também incluir, **produtos** (outcome da atividade do processo), **papéis** (roles, responsabilidades das pessoas envolvidas) e **pré-/pós-condições** (são condições que são verdadeiras antes e depois de atividade do processo ou de um produto ser produzido).

O processo de software específica:

- **O quê**
- **Quem**
- **Quando**
- **Como**

E incluí

- **Papéis** (Roles)
- **Fluxo de trabalho** (Workflow)
- **Procedimentos** (Procedures)
- **Normas** (Standards)
- **Modelos** (Templates)

3.4 Pontos Chave

O processo de Software é um guia

Não existe "um melhor processo para escrever software". Um processo que um individuo ou uma organização escolhe e segue depende de:

- das características específicas do projeto;
- da cultura da organização;
- das habilidades e preferências das pessoas envolvidas.

Um bom processo aumenta a produtividade de membros da equipa menos experientes sem impedir o trabalho/progresso de membros mais experientes.

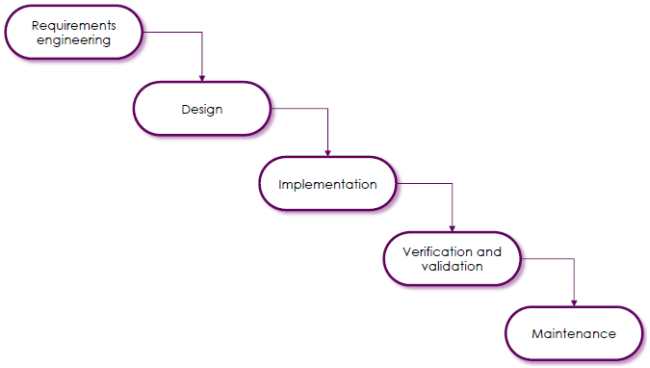
3.5 Resistência ao Processo de Software

Perceção: Algumas pessoas vêm seguir um processo como uma sobrecarga (overhead) desnecessária na produtividade.

- Interfere com a criatividade;
- Burocracia e regimento;
- Prejudica a agilidade em mercados que evoluem rapidamente.

A realidade: Grupos que não seguem um processo definido tendem a adicionar processo mais tarde no projeto, como reação a problemas que surgem. Quando o tamanho e a complexidade do projeto aumenta, a importância de seguir processos definidos aumenta proporcionalmente.

3.6 Fases de Software



3.7 Modelos de Processo de Software

Modelos abstratos que descrevem uma classe abordagens de desenvolvimento com características similares.

Alguns critérios utilizados para distinguir modelos de processos de software são:

- o tempo entre fases (timing);
- critérios de entrada e saída entre fases (entry/exit criteria);
- os artefactos criados durante cada fase;

Alguns **exemplos** incluem: Waterfall, Spiral, Rapid Prototyping, Incremental, Development, ...

3.7.1 Modelos (Tradicionais)

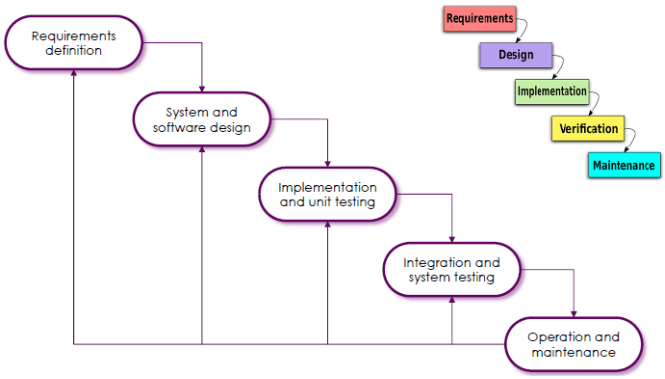
Modelo em Cascata (Waterfall): É um modelo Plan-Driven. Separa e distingue fases de especificação e desenvolvimento.

Desenvolvimento Incremental: A especificação, desenvolvimento e validação são intercalados. Pode ser Plan-Driven ou Agile.

Processos Evolucionários/Iterativos: O sistema é desenvolvido no ínicio usando uma especificação muito simples, sendo modificada e melhorada de acordo com as necessidades de software.

Muitos outros: A maior parte de sistemas grandes são desenvolvidos usando um processo que incorpora elementos de diferentes modelos.

3.7.2 O Modelo em Cascata (Waterfall)



Vantagens

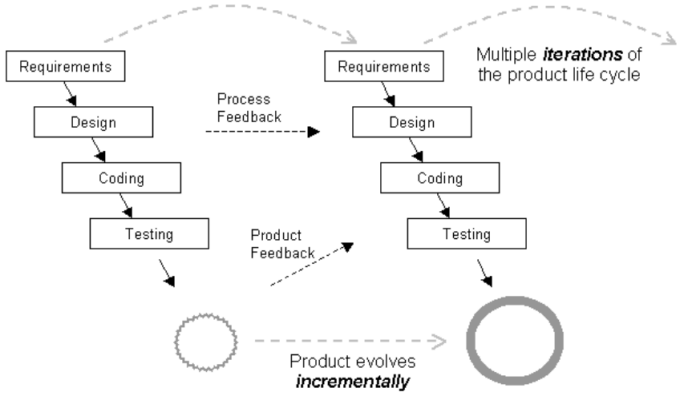
- É simples e fácil de perceber e usar;
- É fácil de planear, um schedule pode ser definido com deadlines para cada fase de desenvolvido e um produto pode ser processado através do processo de desenvolvimento como um carro numa lavagem automática, e ,teoricamente, ser entregue a tempo.
- Fácil de gerir, cada fase tem entregas específicas e um processo de revisão.
- Fases e processos são concluídos um de cada vez.
- Funciona bem onde os requisitos são bem compreendidos.

Desvantagens

- Dificuldade em acomodar mudanças após o processo começar. Em principio, uma fase deve ser concluída antes de começar a próxima. Particionamento inflexível do projeto em fases distintas torna difícil responder a mudanças nos requisitos do cliente.
- Modelo não muito bom para projetos de longa duração ou que já estão em andamento (não é produzido nenhum software funcional até mais tarde no ciclo de vida).
- Não é adequado a processos onde os requisitos são incertos ou onde há risco de serem alterados.

3.7.3 Modelo Incremental

Uma característica de modelos com ciclos de vida modernos. O produto evolui através de uma série de iterações.



Benefícios

- O custo de **acomodar mudanças de requisitos do cliente** é reduzido. A quantidade de análise e documentação que tem de ser refeita é muito menor do que no modelo em cascata.
- É mais fácil **obter feedback do cliente** sobre o desenvolvimento do trabalho que já está concluído. Clientes podem comentar sobre demonstrações do software e ver quanto foi implementado.
- **Entrega mais rápida e deployment** do software útil para o cliente é possível. Os clientes podem usar e ganhar valor do software mais cedo do que se o sistema fosse desenvolvido com o processo em cascata.

Problemas

- **Cada fase de iteração é rigida** e não se sobrepoem umas com as outras.
- O processo não é visível. Os gestores precisam de entragas regulares para medir o progresso. No entanto, se o sistema não for desenvolvido rapidamente, não é cost-effective produzir documentos que reflitam cada versão do sistema.
- A estrutura do sistema tende a degradar-se à medida que novos incrementos são adicionados. A não ser que tempo e dinheiro seja gasto na refatoração para melhorar o software, **regular mudanças tende a corromper a sua estrutura**. A medida que vamos incorporando novas mudanças de doftware, torna-se mais difícil e mais caro.

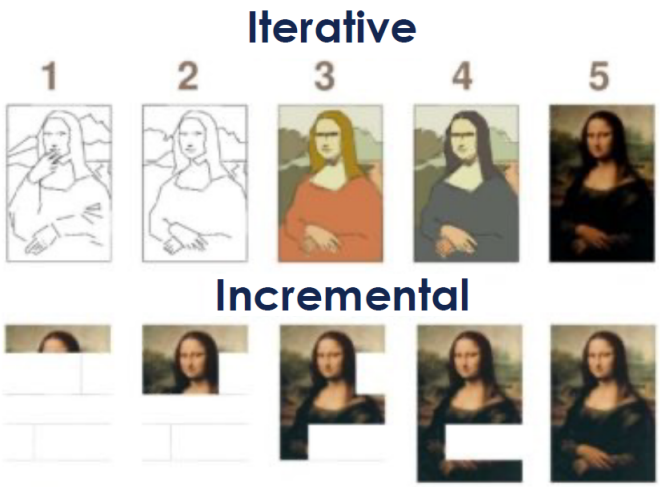
3.7.4 Modelos Evolucionários/Iterativos

Prototipagem: Geralmente, um cliente define um conjunto de objetivos gerais para o software, mas não identifica requisitos detalhados para funções e funcionalidades do sistema.

Modelo Espiral: Utilizando um modelo espiral, o software é desenvolvido numa serie de lançamentos (releases) evolutivos. Durante as primeiras iterações, o lançamento pode ser um protótipo ou um modelo.

Modelo Concurrente: Permite a uma equipa de software representar elementos iterativos e concurrentes de quaisquer modelos de processo.

3.7.5 Incremental vs Evolucionário/Iterativo



3.7.6 Exemplos

❖ **Scenario**
- You are developing a web-based e-commerce platform for a client. The client has requested **several features**, including user authentication, product catalogue, shopping cart functionality, and payment processing. They want to launch the platform as soon as possible to start generating revenue but also want to **add new features and improvements over time**.

❖ Which approach do you propose?
- Incremental
  1. Minimal viable product (MVP) – user authentication and catalogue
  2. Shopping cart
  3. Payment process

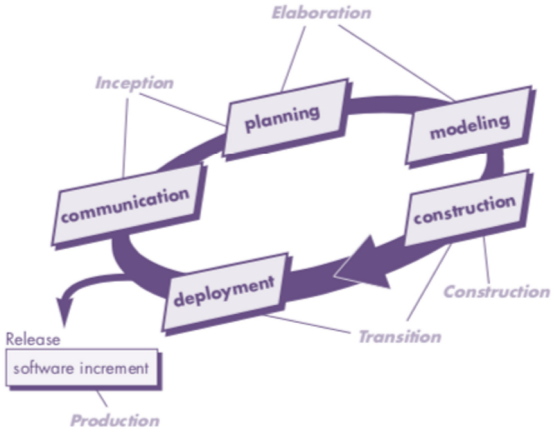
❖ **Scenario**
- You need to develop a machine learning-based recommendation system for an online streaming service. The goal is to provide personalized content recommendations to users based on their viewing history and preferences. The **requirements are complex**, and it's **essential to continually refine the recommendation** algorithms for better accuracy and user satisfaction.

❖ Which approach do you propose?
- Iterative
  1. Initial version of the system
  2. Simple recommendation algorithm
  3. Refine algorithms and models (in subsequent iterations)

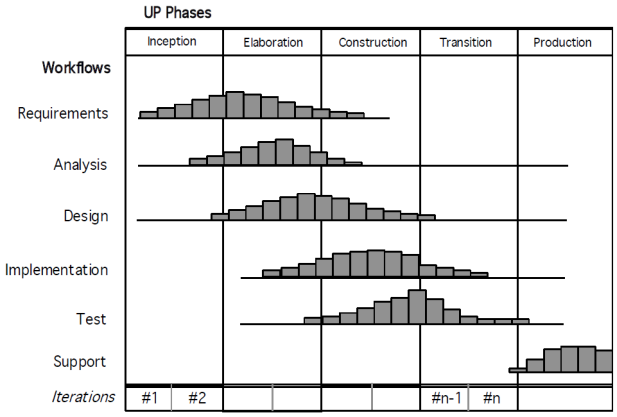
3.8 Outros Modelos de Processo

- Desenvolvimento **Component-Based** (COTS): O processo para aplicar quando reutilizar é um objetivo do desenvolvimento.
- **Métodos Formais**: Enfatiza a especificação matemática dos requisitos.
- **Processo Unificado (UP)**: Um processo de software "use-case driven, arquitetura-centric, iterativo e incremental", alinhado com o Unified Modeling Language (UML).

3.9 Processo Unificado (UP)



3.9.1 Fases



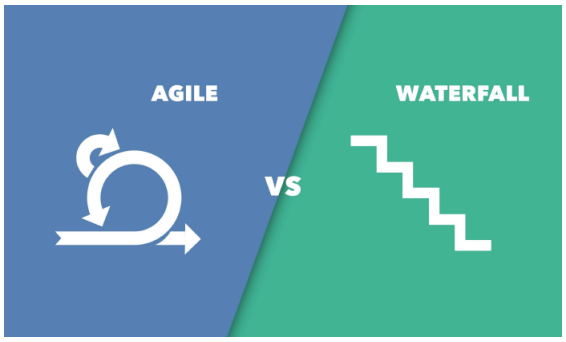
3.10 Processos Plan-Driven e Agile

Processos Plan-Driven são processos onde todas as atividades são planeadas anteciadamente e o progresso é medido contra este plano.

Em **Processos Agile**, planear é incremental e é mais fácil mudar o processo para refletir a mudança nos requisitos do cliente.

Na prática, a maior parte dos processos incluem elementos de ambos, processos plan-driven e agile. Não existem processos de software "certos" ou "errados".

3.10.1 Processos Plan-Driven vs Agile



3.10.2 Processos Agile

Desenvolvimento rápido e entragas são, geralmente, os requisitos mais importantes para sistemas de software.

- Negócios operam num ambiente **fast-changing requirements** e é praticamente impossível produzir um conjunto de requisitos de software estáveis.
- O Software deve evoluir rapidamente para refletir mudanças de negócios.

Desenvolvimento Plan-Driven é essencialmente para alguns tipos de sistemas mas não cobre as necessidades do negócio.

Métodos Agile

- Métodos Agile foram desenvolvidos num esforço para ultrapassar fraquezas percetidas e reais em engenharia de software convencional.
- **Foca no código em vez de no design**.
- Baseados em **abordagens iterativas** ao desenvolvimento de software.
- Tem a intenção de **entregar software funcional rapidamente** e evoluir rapidamente para refletir mudanças de requisitos do cliente.

3.10.3 Origem: Manifesto Agile

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

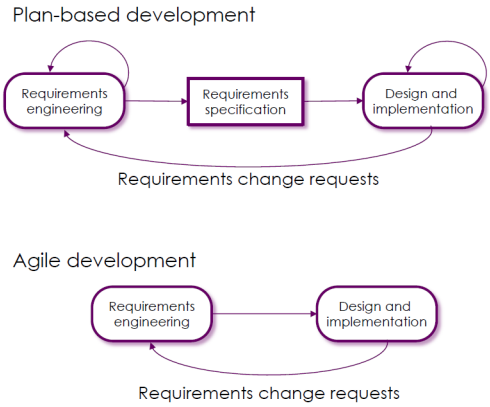
- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

3.10.4 Princípios dos Métodos Agile

Involvimento do cliente: Os clientes devem estar envolvidos durante todo o porcesso de desenvolvimento. O seu papel é fornecer e priorizar novos requisitos e avaliar as iterações do sistema.

3.10.5 Desenvolvimento Plan-Driven e Agile



Desenvolvimento Plan-Driven:

- Baseado ao redor de um desenvolvimento separado de fases, com outputs a serem produzidos em cada fase planeada anteciadamente.
- Não é necessariamente o modelo em cascata - desenvolvimento incremental, plan-driven, é possível.

Desenvolvimento Agile:

- Especificação, design, implementação e testes são intercalados.
- Os outputs do processo de desenvolvimento são decididos através de um processo de negociação durante o processo de desenvolvimento de software.

3.10.6 Métodos Agile - Benefícios

- Requisitos num modelo Agile podem ser alterados conforme os requisitos do cliente mudam. Por vezes os requisitos não são muito claros. Mudanças nos requisitos são aceites mesmo em fases avançadas do processo de desenvolvimento.
- A entrega de software é continua. Clientes podem seguir cada feature funcional do Sprint do software.
- Refatorar o código não é muito caro.

3.10.7 Métodos Agile - Desvantagens

- A documentação é escassa.
- Com requisitos pouco claros, é difícil estimar o resultado pretendido. Mais díficil de estimar o esforço necessário.
- Alguns riscos desconhecidos/imprevisiveis que podem afetar o desenvolvimento do projeto.

3.10.8 Métodos Agile - Aplicabilidade

- Desenvolvimento de produtos, onde uma empresa de software está a desenvolver um produto pequeno/médio em tamanho para venda.
- Desenvolvimento de sistemas customizados dentro de uma organização, onde existe o compromisso do cliente ficar envolvido no processo de desenvolvimento e onde existem algumas regras/regulamentos externos que afetam o software.
- Virtualmente, todos os produtos de software e aplicações são desenvolvidas usando abordagens Agile.