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.

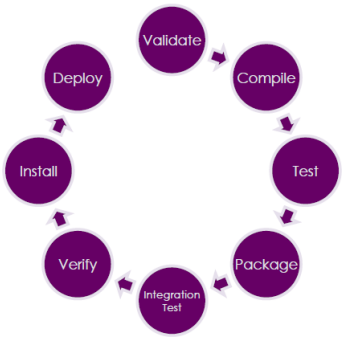
1.5 Ciclo de Vida Maven

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)

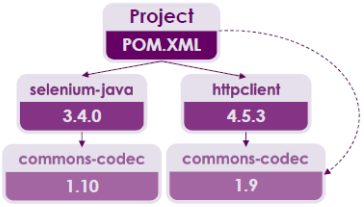
Um Archetype é um template para um projeto Maven, que pode ser usado para criar um novo projeto rapidamente.

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:

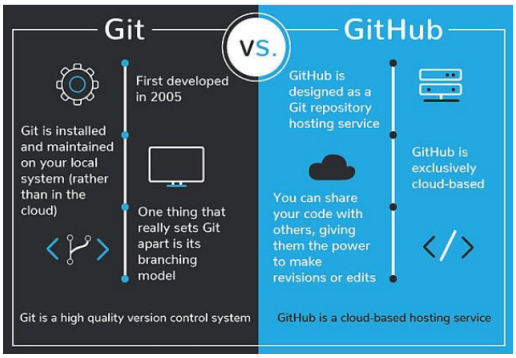
- Mantém várias (antigas e novas) versões de tudo (não só código fonte);
- Pede por comentários quando se fazem alterações;
- Permite "check-in" e "check-out" de ficheiros para saber em que ficheiros outras pessoas estão a trabalhar;
- Mostra as diferenças entre versões;

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**:

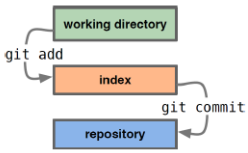
- Um estado completo do projeto, incluindo todos os ficheiros;
- O primeio não possui pai;
- Normalmente, pegamos num commit object, fazemos alterações, e criamos um novo commit object, pelo que a maior parte dos commit objects têm apenas um pai;
- Quando fazemos **merge** de dois commit objects, forma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

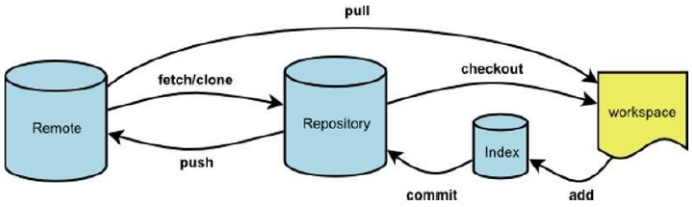
Os commits são "baratos" pelo que os devemos fazer com frequência, e com mensagens descritivas sobre o que foi alterado. Devem ter apenas uma linha.

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

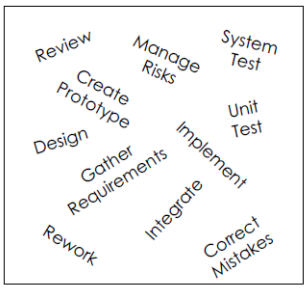


3 O Processo de Desenvolvimento de Software

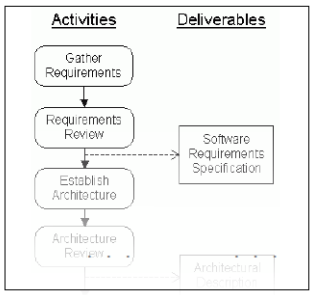
3.1 Processo

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.

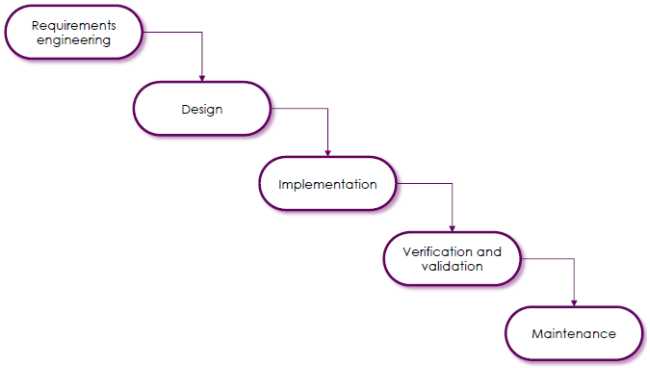
3.5 Resistência ao Processo de Software

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)

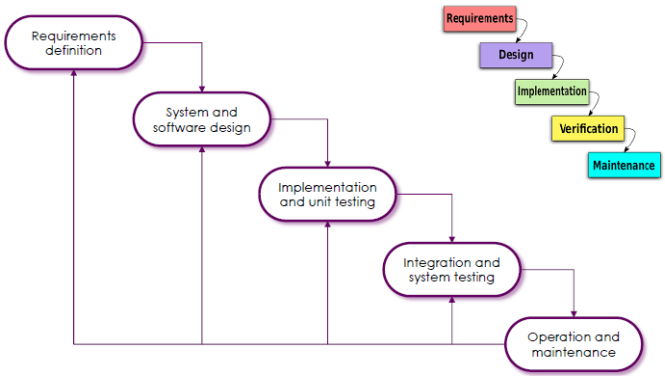
Modelo em Cascata (Waterfall): É um modelo Plan-Driven. Separa e distingue fases de especificação e desenvolvimento.

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

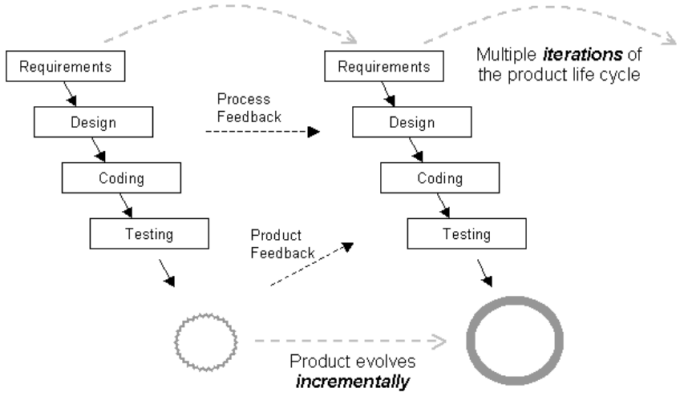
- É simples e fácil de perceber e usar;
- É fácil de planear, um schedule pode ser definido com deadlines para cada fase de desenvolvido e um produto pode ser processado através do processo de desenvolvimento como um carro numa lavagem automática, e ,teoricamente, ser entregue a tempo.
- Fácil de gerir, 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.

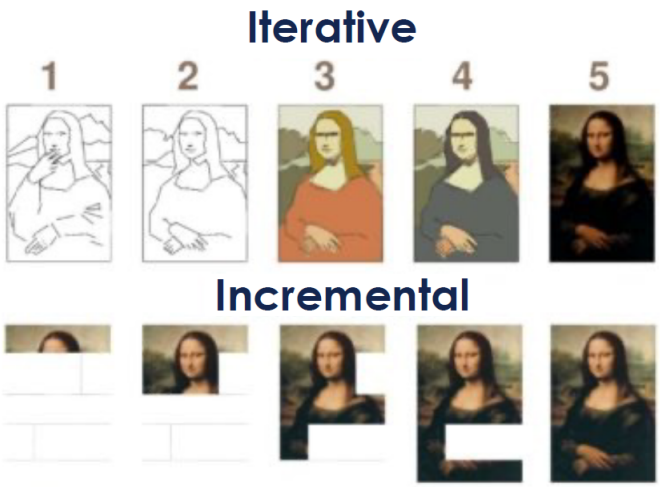
3.7.4 Modelos Evolucionários/Iterativos

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

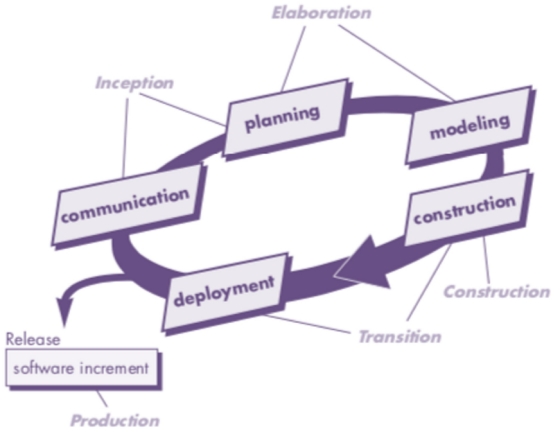
❖ **Scenario**
- You need to develop a machine learning-based recommendation system for an online streaming service. The goal is to provide personalized content recommendations to users based on their viewing history and preferences. The **requirements are complex**, and it's **essential to continually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

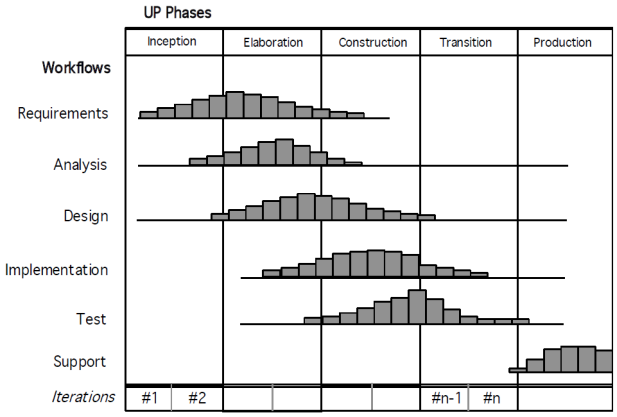
3.8 Outros Modelos de Processo

- 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



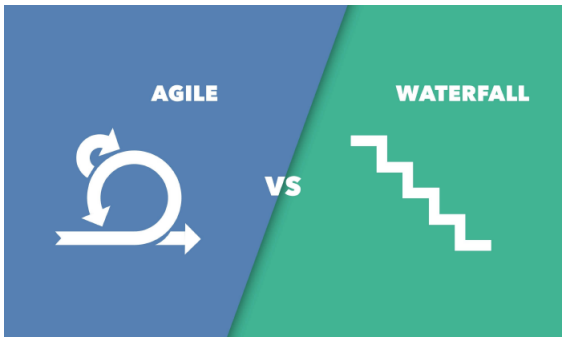
3.10 Processos Plan-Driven e Agile

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:

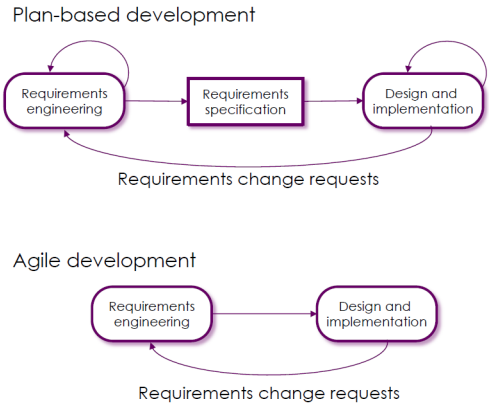
- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

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.

4 Desenvolvimento de Software Agile

4.1 Princípios Agile

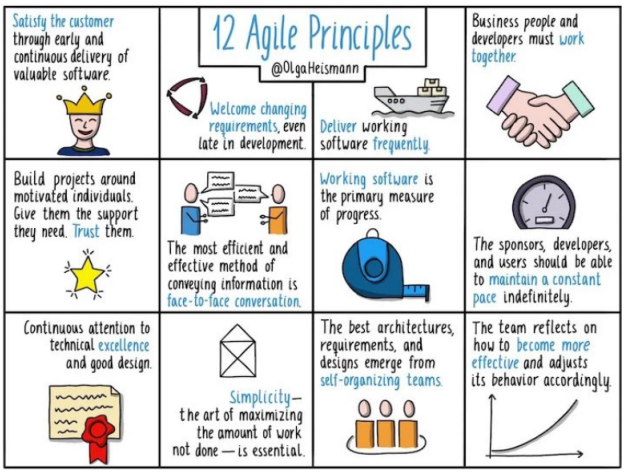
4.1.1 Porquê Agile?

O desenvolvimento rápido e entrega 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.

Princípios Agile:

- **Focada 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.



Ver analogia nos slides 6 a 15.

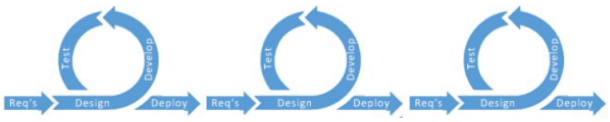
4.2 Técnicas de Desenvolvimento Agile

As fases especificação, design, implementação e avaliação são intercaladas:

- O sistema é desenvolvido como uma série de versões, envolvendo **stakeholders** na especificaçãon e avaliação;
- Entregas de novas versões frequente para avaliação;

Vasto suporte de ferramentas (ex: ferramentas de testes automáticos) usado para suportar o desenvolvimento.

Miníma documentação, uma vez que o foco é o código.



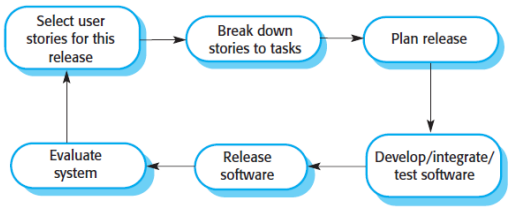
4.2.1 Extreme Programming (XP)

Extreme Programming (XP) é a abordagem mais utilizada para desenvolvimento ágil de software. **Leva uma abordagem "extrema" para o desenvolvimento iterativo**:

- Novas versões podem ser contruidas varias vezes por dia;
- Os incrementos são entregues aos clientes a cada 2 semanas;
- Todos os testes devem correr para cada build e cada buid apenas é aceite se todos os testes correrem com sucesso;

Utiliza uma abordagem orientada a objetos como o paradigma de desenvolvimento preferido. Abrange um conjunto de regras e práticas que ocorrem no contexto de quatro atividades fundamentais (framework activities): **planning, design, coding, testing**.

4.2.2 Release Cycle XP



4.2.3 Práticas XP

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Principle or practice	Description
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop, and all the developers take responsibility for all the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium-term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

4.2.4 Práticas influentes de XP

Extreme Programming tem um foco técnico e não é fácil de integrar com prática de gestão na maior parte das organizações. Consequentemente, enquanto desenvolvimento Agile utiliza práticas de XP, o método referido não é muito utilizado.

Práticas Chave:

- User stories para especificação;
- Refatoração;
- Desenvolvimento Test-First;
- Programação em pares (pair programming);

4.2.5 User Stories para Requisitos

Em XP, o cliente ou user são parte da equipa XP e são **responsáveis por tomar decisões sobre requisitos**.

Requisitos do user são expressos como **user stories** ou **scenarios**. Estes são escritos em cartões e a equipa de desenvolvimento divide-os em **tarefas de implementação**. Estas tarefas são a base das estimativas de custo e de schedule.

O cliente escolhe as **stories para inclusão** na próxima release baseado em prioridade e estimativa de schedule.

Exemplo slide 24 e 25.

4.2.6 Templates de User Stories

As a **(user)**, I **(want to)**, **(so that)**.

- **User:** Refere o user final do software;
- **Want to:** Refere à intenção do user, e não à feature que este utiliza. Temos de especificar aquilo que o user quer alcançar com a tarefa, sem mencionar a UI da aplicação;
- **So that:** Descreve a "bigger picture". Qual é o objetivo do user? O que é que ele quer alcançar com esta feature?

Exemplo: As a manager, I want to be able to understand my colleagues progress so that I can report our success and failures.

4.2.7 Organização de User Stories

- **Role-Goal-Benefit:** Força o cliente a realmente pensar quem é que vai beneficiar de uma feature, o que é que eles querem alcançar e porque é que eles querem alcançar isso;
- **Limites/Needs:** Subconjunto de situações que são de interesse para esta feature;
- **Definição de Done:** Como validar uma feature e saber que esta está concluida?
- **Tarefas de Engenharia:** Como esta feature interaje com outras features dentro do sistema ou outros subsistemas;
- **Estimativa de esforço:** Fornece uma medida concreta sobre o valor de uma feature;

4.2.8 Refatoração

Conhecimento convencional em engenharia de software é **design for change**. Compensa gastar tempo e esforçarmo-nos a anticipar mudanças, uma vez que, reduz o cussto mais à frente no ciclo de vida.

No entanto, XP mantém que este principio não compensa, uma vez que, mudanças não podem ser previstas. Em vez disso, propõe **melhorias contantes do código** (refactoring) para tornar mudanças mais fáceis quando têm de ser implementadas.

Equipas de programação procuram por possíveis **melhorias de software** e fazem-nas mesmo que não sejam necessárias naquele momento. Isto melhora a **compreensão do software** e desta maneira reduz a necessidade de documentação.

Mudanças são **fáceis de fazer** uma vez que o código é **bem estruturado** e **claro**. No entanto, algumas mudanças requerem architecture refactoring, o que é muito mais caro.

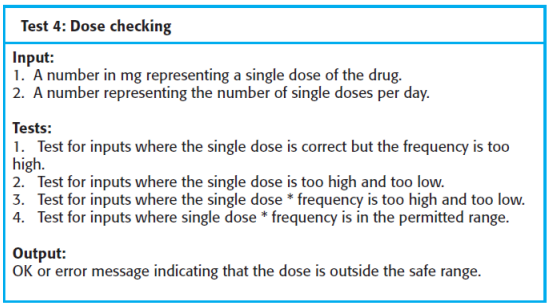
Exemplo

- **Re-organização** de uma classe hierárquica para remover duplicação de código;
- **Renomear** atributos e métodos de modo a serem mais faceis de entender;
- **Trocar inline code** com chamadas de métodos que estão incluidas em bibliotecas;

4.2.9 Desenvolvimento Test-First

Testar é uma parte central em XP. O **desenvolvimento test-first**:

- Teste incremental é desenvolvido a partir de cenários;
- Envolvimento de users no desenvolvimento de testes e validação;
- Correr todos os testes de componente de cada vez que uma nova versão é built.



Test 4: Dose checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

4.2.10 Automação de Testes

Testes são escritos como **componentes executáveis** antes da tarefa ser implementada. Estes devem:

- Ser autonomos (stand-alone);
- Simular a submissão de input para ser testado;
- Verificar que o resultado é o esperado;

Uma **automated test framework** (ex: Junit) torna mais fácil de escrever e executar testes.

Uma vez que os testes são automáticos, existe um conjunto de testes que podem ser rapidamente e facilmente executados. Quando uma nova funcionalidade é adicionada ao sistema, todos os testes podem ser executados e identificar problemas no novo código imediatamente.

4.2.11 Problemas com o desenvolvimento Test-First

- Os programadores preferem programar em vez de escrever testes, pelo que, por vezes estes fazem **short cuts ao escrever testes**. Como pro exemplo, testes incompletos que não testam todas as possibilidades.
- Alguns testes são **muito difíceis de escrever incrementalmente**. Como testes para uma UI complexa, é difícil escrever testes com display logic.
- É dificil de julgar a **cobertura** de um conjunto de testes. Podemos ter muitos testes de sistema e não cobrir tudo.

4.2.12 Programação em Pares (Pair Programming)

Um parte programadores a trabalharem juntos para desenvolver código. Os programadores sentam se juntos num mesmo computador para desenvolver software. Serve como uma forma de **revisão de código informal** uma vez que cada linha de código é olhada por mais do que uma pessoa.

Isto ajuda a desenvolver common ownership do código e compreensão coletiva pela equipa. Reduz o risco quando um programador deixa a equipa.

Em pair programming, os pares são criados dinamicamente para que todos os membros da equipa trabalhem uns com os outros durante o processo de desenvolvimento. Encoraja a refatorização, uma vez que, toda a equipa pode benificiar com a melhoria do código do sistema.

Pair programming não é necessariamente ineficiente. Estudos mostram que um par ao trabalhar junto é mais eficiente do que dois programadores a trabalhar separadamente.

4.3 Gestão de Projeto Agile

A principar responsabilidade da gestão de projeto de software é **gerir o projeto**, para que o software seja entregue a tempo e dentro do budget planeada para o projeto.

A abordagem tradicional para a gestão de projeto é **plan-driven**. Define o que deve ser entregue, quando deve ser entregue e quem vai trabalhar em cada entrega.

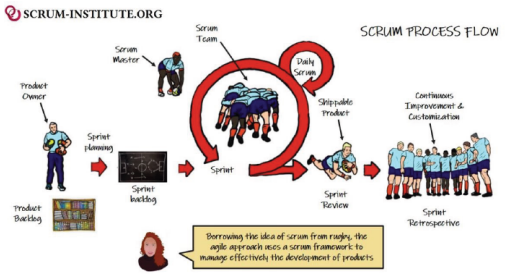
A gestão de projeto Agile requer uma abordagem diferente. É adaptada de um **desenvolvimento incremental** e as **praticas usadas em métodos agile**.

4.3.1 Scrum

É um **método agile** que foca na gestão incremental do desenvolvimento.

Existem três fases no Scrum:

- A fase inicial é o **outline planning** onde tem de se estabelecer os objetivos gerais para o projeto e fazer o design da arquitetura de software.
- Isto é seguido por uma serie de **ciclos sprint**, onde cada ciclo desenvolve um incremento do sistema.
- A fase de **encerramento do projeto**, termina o projeto, completa a documentação necessária como ajudas nos frames e manual do utilizador e acessa aquilo que foi aprendido com o projeto.

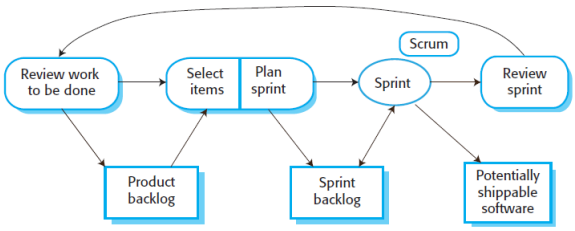


4.3.2 Scrum - Terminologia

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.

Scrum term	Definition
Scrum meetings	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

4.3.3 Scrum sprint cycle



O ponto inicial ara o planeamento é o **product backlog**. É uma lista do trabalho a ser feito no projeto.

A **fase selecionada** envolve toda a equipa. Quem trabalha com o cliente para selecionar as features e as funcionalidades do **product backlog** que vão ser desenvolvidas durante o sprint.

Durante a **fase de desenvolvimento**, a equipa está isolada do cliente e da organização. Todos os canais de comunicação passam pelo "**Scrum master**".

No **final do sprint**, o trabalho realizado é revisto e apresentado aos stakeholders.

4.3.4 Teamwork in Scrum

O papel do **Scrum master** é proteger o a equipa de desenvolvimento de distrações externas.

- Realiza reuniões diárias;
- Vai verificando o backlog para ver o trabalho a ser feito;
- Toma decisões;
- Mede o progresso usando o backlog;
- Comunica cocm os clientes e a gestão fora da equipa;

A equipa vai a reuniões diárias curtas (**Scrums**)

- Membros partilham informação, descrevem o progresso/problemas desde a última reunião, e aquilo que estava planeado para o dia;
- Isto significa que todos na equipa sabem aquilo que se passa, os porblemas que surgiram, podem re-planear num curto tempo o trabalho;

4.3.5 Scrum - Benefícios

- O produto é repartido num conjunto de pedaços **geriveis**. e **faceis de compreender**.
- **Requisitos instáveis** não prrendem o progresso;
- Toda a equipa tem visibilidade de tudo, consequentemente, a comunicação da equipa melhora;
- Os clientes vêm as entregas on-time dos incrementos, ganhando feedback em como o produto funciona;
- Confiança entre a equipa e o cliente, todos esperam que o projeto tenha sucesso;

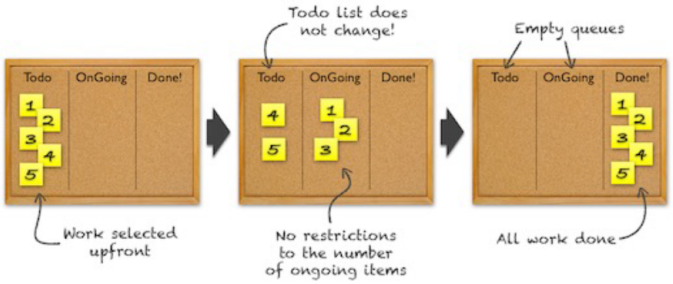
4.3.6 Kanban

É um sistema ágil que pode ser usado para melhorar qualquer desenvolvimento de software incluindo Scrum, XP ou Waterfall.

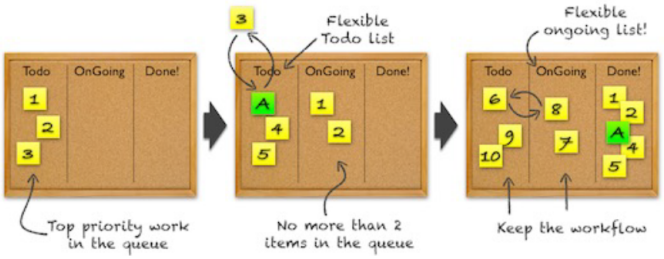
O nome "Kanban" vem do Japonês e significa "signboard" ou "billboard". Foi usado pela primeira vez pela Toyota para Just-in-time manufacturing plants, com o objetivo de limitar o trabalho em progresso (WIP - work in progress).

4.3.7 Scrum vs Kanban

Em **Scrum**, tu escolhes o trabalho que vais fazer durante o próximo sprint antes deste começar. Depois, damos lock ao sprint, fazemos o trabalho todo, e passado algumas semanas (duração normal para um sprint), a queue fica vazia.



Em **Kanban**, tudo o que é limitado é o tamanho das queues, chamado de **work in progress (WIP) limits**. Podemos alterar os items na queue a qualquer momento, não existindo um "fim de sprint". O trabalho continua a fluir.



	Scrum	Kanban
Cadence	Regular fixed length sprints (i.e., 2 weeks)	Continuous flow
Release methodology	At the end of each sprint	Continuous delivery
Roles	Product owner, scrum master, development team	No required roles
Key metrics	Velocity	Lead time, cycle time, WIP
Change philosophy	Teams should not make changes during the sprint.	Change can happen at any time

4.4 Scaling Agile Methods

Métodos Agile provaram ser muito eficazes para projetos pequenos/médios, que podem ser desenvolvidos por uma equipa pequena co-localizada. Escalar métodos Agile involve mudar estes para poderem ser utilizados em projetos cada vez maiores, onde existem múltiplas equipas de desenvolvimento, podendo trabalhar até mesmo em diferentes locais.

Quando escalamos métodos Agile, é importante manter fundamentos Agile: Planeamento flexivel, releases de sistema frequentes, integração contínua, desenvolvimento test-driven e boa comunicação entre a equipa.

4.4.1 Problemas de Sistema

- Quão grande é o sistema a ser desenvolvido? Métodos Agile são mais eficientes em equipas pequenas e co-localizadas, em que ocorre uma comunicação informal.
- Que tipo de sistema está a ser desenvolvido? Sistemas que necessitem de **muita análise antes da implementação** precisam de um design (mais) detalhado.
- Qual é o tempo de vida esperado do sistema? **Sistemas de longa duração necessitam de documentação** para comunicar as intenções do sistema com os developers da equipa de suporte.
- O sistema está sujeito a regulamentos externos? Se o sistema é regulado, muito provavelmente, é necessário **produzir documentação detalhada** como parte do caso de segurança do sistema.

4.4.2 Desenvolvimento de Sistemas Grandes

Sistemas grandes são normalmente coleções de sistemas de comunicação separados, onde **equipas separadas** desenvolvem cada sistema. Frequentemente, estas equipas trabalham em locais diferentes, por vezes até em zonas com diferentes fusos horários.

Sistemas grandes são "**brownfield systems**" isto é, incluem e interagem com vários sistemas existentes. Muitos dos requisitos do sistema estão relacionados com a interação, logo não se rendem muito à flexibilidade e desenvolvimento incremental.

Sistemas grandes e os seus processos de desenvolvimento são muitas vezes restritos por **regras e regulamentos externos**, limitando a forma como podem ser desenvolvidos.

Sistemas grandes tem uma **aquisição longa** e um tempo de desenvolvimento longo também. É dificil de manter equipas coerentes que sabem sobre o sistema durante esse periodo, uma vez que, eventualmente, os membros vão para outros projetos ou trabalhos.

Sistemas grandes normalmente tem um conjunto diverso de **stakeholders**. É praticamente impossível involver todos estes stakeholders no processo desenvolvido.

4.4.3 Agile: Scaling up em sistemas grandes

- Uma abordagem completamente incrementeal aos requisitos de engenharia é **impossível**;
- **Não pode existir** um único product owner;
- **Não é possível** focar apenas no código;
- Mecanismos de comunicação cross-team têm de ser **designed** e **utilizados**;
- Integração continua é praticamente **impossível**, no entanto, é essencial manter builds frequentes do sistema e regular releases do sistems;

4.4.4 Multi-team Scrum

- **Role replication**: Cada equipa tem um Product Owner para o seu trabalho e Scrum Master.
- **Product architects**: Cada equipa escolhe um arquiteto do produto que colaboram para desenhar e evoluir toda a arquitetura do sistema.
- **Release alignment**: As datas de releases de produtos de cada equipa são alinhadas de modo a demonstrar que um sistema completo está produzido.
- **Scrum of scrums**: Existe um Scrum of scrums diário, onde os representantes de cada equipa se encontram para discutir o progresso e o plano de trabalho a ser concluído.